Third Laboratory - Team Green

Caliguri Matteo, Cestaro Riccardo, Vaishnav Harshul,
Mari Daniele, Porro Thomas

December 2020

1 Task 1

For task one, the implementation was pretty straight forward, numpy was used to represent arrays of binary numbers. The complexity of the protocol is polynomial in l_k, l_c since the number conversion is polynomial in l_c, l_k and it is done a constant number of times, the same goes for integer sum and multiplication. The complexity of the protocol w.r.t. l_k for different values of l_c can be seen in figure 1, this was done by measuring the time before and after the execution of the protocol and subtracting the two. This was repeated multiple times and the values were averaged in order to obtain a more reliable estimation of the time needed for the execution. Still it is possible to see that the measurements were noisy. In general it is possible to see that as we increase l_k and l_c the time increases linearly or slightly super-linearly showing how the complexity requirements were met.

2 Task 2

In this task, suppose that the attacker, Eve, wants to fool Bob on the n^{th} round, then by assumption she also knows $n-25$, c_{n-25} and r_{n-25} because she observed these values during a protocol exchange between Alice and Bob. From c_{n-25} Eve can compute $s_{c,\ n-25}$ by converting it to decimal and summing all its digits, this means that she can also find $s_{t,n-25}$ by computing $s_{n-25} = decimal(r_{n-25})$ and then $s_{t,\ n-25} = \frac{s_{n-25}}{s_{c,\ n-25}}$. It is possible to see that $t_n = k+n = k+n-25+25 = t_{n-25}+25$. By calling $d_i(t)$ the digit $d_i = t//10^i \textbf{ mod } 10$ where $//$ is the integer division, it is possible to see that if $d_0(t_{n-25}) < 5$ and $d_1(t_{n-25}) < 8$ then $s_t = s_{t-25} + 7$ where 7=2+5 is the sum of the digits of 25. It is easy to see that $P(d < 5) = 0.5$ and $P(d < 8) = 0.8$ then $P(d_0(t_{n-25}) < 5, d_1(t_{n-25}) < 8) = 0.4$ since the two probabilities are independent.

One would be tempted to say that adding 7 to s_{t-25} actually yields the highest success probability, but actually this is not true. Considering the case $d_0(t_{n-25}) \geq 5$ and $d_1(t_{n-25}) < 8$ it is possible to see that almost always $s_t = s_{t-25} - 2$ so also subtracting 2 has success probability that is similar to that of

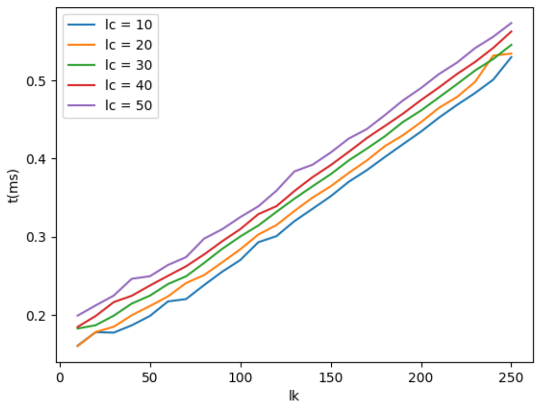

Figure 1: Computational complexity of the protocol

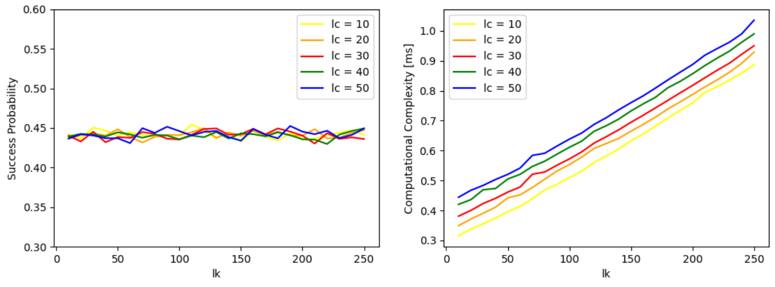

Figure 2: Success probability(left) and complexity(right) for task 2

adding 7. By running some simulations it is clear that $s_t = s_{t-25} - 2$ also in some cases when $d_0(t_{n-25}) \geq 5$ and $d_1(t_{n-25}) \geq 8$ making it a better choice wrt summing 7. As a matter of fact this approach has a success probability that is close to 44% as will be seen later in plot 2.

So during the attack Eve computes $\hat{s}_{t,n} = s_{t,n-25} - 2$, then $\hat{r} = binary(\hat{s}_{t,n} s_{c,n})$ and submits it to bob hoping that he will accept him.

From figure 2 it is possible to see that the success probability is approximately 0.44 and that it does not depend on l_k, l_c, this measure was obtained by running multiple times the attack and by finding out the fraction of successful cases. Since the attack doesn't actually do anything more than the normal protocol except from a fraction and a sum it is possible to see that the complexity trend is very similar to the one obtained with the standard protocol.

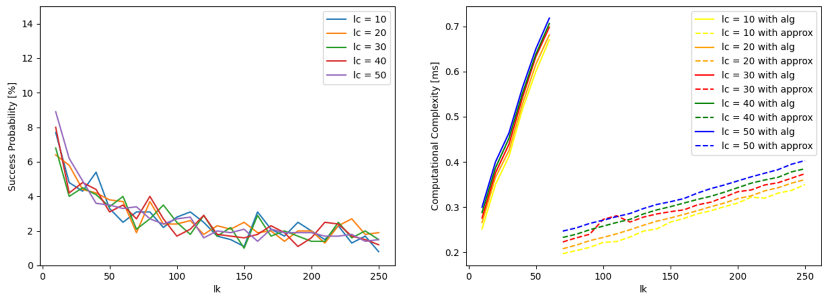

Figure 3: Success probability(left) and computational complexity(right) for task 3

3 Task 3

The weakness exploited to perform an attack on the protocol without seeing any of its previous iterations is the fact that given t uniformly distributed, S_t isn't. Because of this some values of S_t are more likely to occur. Furthermore the sum of the digits operation wastes a lot of the information carried by the bits of the key k, so even with key very big the success probability is pretty high. In order to compute this distribution it is possible to employ a dynamic programming approach. Let's call $SD(d)$ the function that computes the sum of the digits of number d and $Dist(d, c)$ the function that returns the number of values i s.t. $SD(i) = c, \ i < d$. Now it is easy to see that $Dist(1, 0) = 1, \ Dist(1, i) = 0 \ i > 0$. At this point it is possible to see that $Dist(2, j) = Dist(1, j) + Dist(1, j-1), Dist(3, j) = Dist(2, j) + Dist(1, j-2), \ldots Dist(10, j) = Dist(9, j) + Dist(1, j-9)$. And then again $Dist(20, j) = Dist(10, j) + Dist(10, j-1), Dist(30, j) = Dist(20, j) + Dist(10, j-2), \ldots, Dist(100, j) = Dist(90, j) + Dist(10, j-9)$. This can be easily generalized also for bigger multiples of power of 10. Using this procedure it is possible to see that the computational complexity needed to compute the function Dist(d, c) with d big is notably reduced since it is logarithmic in d and not linear. Given the previous property it is possible to find an algorithm to compute Dist(d, c) for a general d, see the pseudo code 1.

Now since t can take values only in $[n, 2^{l_k} + n]$ it is possible to compute the actual probability distribution of the sum of the digits of the possible values of t as $\frac{Dist(2^{l_k}+n) - Dist(n)}{sum(Dist(2^{l_k}+n) - Dist(n))}$ and the optimal t value can be computed as the argmax of this array. The complexity of the algorithm is actually linear in l_k, l_c. During the attack the length of the key l_k is known and the attacker can pose as Alice during the first two steps in order to get c, n. The transmitted values of c and n can be used to compute respectively s_c, by using the protocol, and the guess $\hat{s}_t$ through the algorithm described above. A problem encountered with this approach in the homework was that due to numpy integer being saved

Algorithm 1 Distri Algorithm

```
procedure DIST(D)
    notableDists ← [ ]
    distribution[i] ← 1 if i = 0, 0 otherwise

    // this returns and array containing the digits of d
    digits ← getDigitArray(d)

    // exponent of the highest power of 10 smaller or equal than d
    exponent ← length(digits) − 1

    for all  $i \leftarrow 0, \ldots,$ exponent + 1 do
        notableDists[i] ← distribution
        if  $i \leq$ exponent then
            limit ← 9
        else limit ← digits[0]
        for all $j \leftarrow 1, \ldots,$ limit do
            distribution[c] =  distribution[c] + notableDists[i][c - j]

    // At this point distribution[c] contains $Dist(digits[0]10^{exp}, c)$
    // Now it is important to finish computing $Dist(d, c)$,
    // this is done by repeating the previous operation but
    //in this case from the bigger digit to the smaller one
    cumulativeSum ← 0
    for all $i \leftarrow 0, \ldots,$ exponent - 1 do
        cumulativeSum ← cumulativeSum + digits[i]
        for all $j \leftarrow 0, \ldots, digits[i+1]$ do
            $distribution[c] \leftarrow distribution[c] + notableDists[exponent - i - 1][c - cumulativeSum - j]$
    return notableDists
```

with a fixed number of bits, with $l_k > 68$ some of them were overflowing making the result distribution useless. This problem could be solved easily by using other data types but in this case a different approach was employed for $l_k > 68$. In particular usually the distribution of the sums of the digits is almost symmetric, with the maximum in the center. So a simpler way to get a good estimate of the best value s_t is to compute the biggest possible sum of the digits $s_{t,max} = \lceil 9log_{10}(n + 2^{l_k}) \rceil$ and choose $\hat{s}_t = \frac{s_{t,max}}{2}$. This approach is actually faster than the previous one, but it yields a less accurate estimate of the best value for s_t, plus by computing the whole distribution the attacker could correctly compute its success probability. Because of this both approaches have their pros and cons and one can be chosen over the other depending on the necessities and capabilities of the attacker. Once $s_c, \hat{s}_t$ are computed their product $\hat{s}$ is transformed to its binary representation and submit to Bob in order to find out if the correct value of r was guessed.

The plots in figure 3 were generated using two different strategies for different values of l_k in particular:

- $l_k \leq 68$: here the algorithm 1 was used
- $l_k > 68$: here the best value was computed using the approximation described above.

The success probability looks very smooth, showing that the two approaches yield similar results, and it looks independent on l_c but actually decreases as l_k increases. For the computational complexity it is possible to notice a big difference between the two parts of the plot since obviously it is much faster to compute the approximation. Still both the approaches are linear in l_k, l_c since the trend displayed in the plot is.