# Group members

Cyril Renevey, Lukas Pestalozzi, Joey Zenhäusern, Leonardo Wirz

# Problem A

To find the solution of this problem we can use the brute force strategy, because $p = 127$ is low enough, which mean that we try all $(x, y) \in [0, 126] \times [0, 126]$ and count the ones that solves $y^2 = x^3 + 7 \; mod \; p$ including $(\infty, \infty)$. The result in $N = 127$.
Since $N = p$ then we can rewrite the Hasse's inequality as $|N - p - 1| < 2\sqrt{p} \implies |-1| < 22.5$, which is verified.
The point $\alpha = (\alpha_1, \alpha_2) = (19, 32)$ belongs to the set $E_p$ because :

$$\alpha_1^3 + 7 \; mod \; 127 = 6866 \; mod \; 127 = 8$$
$$\alpha_2^2 \; mod \; 127 = 1024 \; mod \; 127 = 8.$$

Furthermore, we can use the Bézout's identity mod $p$ to compute the division required to compute the sequence. Indeed to find the inverse $a^{-1}$ of a number $a$ in a Galois field we can use if $a$ and $p$ are co-prime (which is the case since p is prime) :

$$at = 1 \; mod \; p$$

Where t is a Bézout's coefficient, thus $a^{-1} = t$ can be used to compute a division such as $b/a = ba^{-1}$.
To compute the sequence $\beta_n = n(19, 32)$, with $n = 1, ..., 100$ we can use the given formulas to compute addition on the elliptic curve. The points of the sequence are exposed on the figure 1.

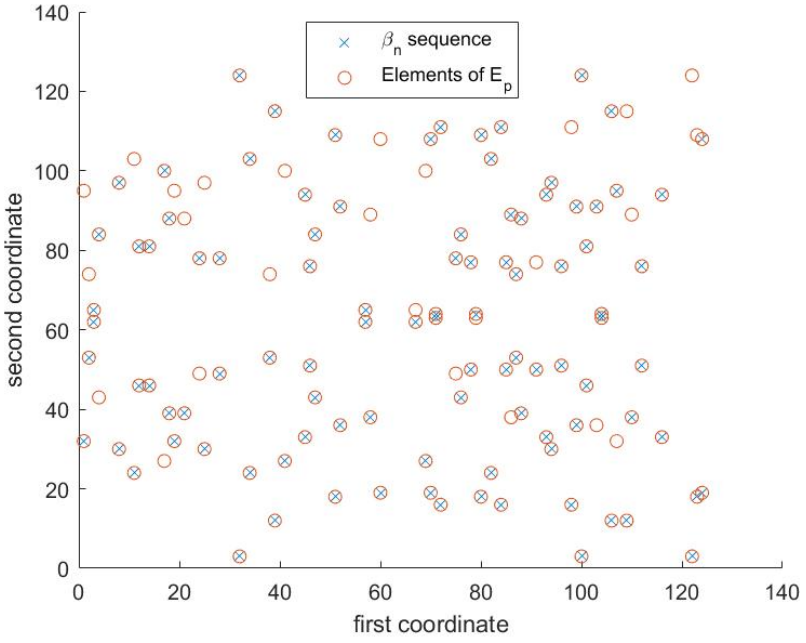


Figure 1: 2D representation of the sequence $\beta_n$ and the elements of the set $E_p$

Because of the size of the sequence, it is not suitable for encryption purposes. However if we do not take the size into account, we need to check if the sequence has twice the same points. We can do that by sorting the points of the sequence by value of x first and then value of y and check if neighbors are equals or not. By doing so we find that there are no equal points in the sequence, thus it is a suitable sequence for encryption purposes.

# Problem B

Our hash function (*MD6*) is based on a sample implementation of the MD5 algorithm from rosettacode[1]. To end up with a 32-bit digest, we changed a couple of things in the algorithm. The message is not treated in blocks of 512 bits any more, but in blocks of 128 bits instead, and the main algorithm operates on 8-bit words. We also only need to store 16 rotation values for the *left_rotate* function. We kept the first 4 entries of every line of the original rotation amounts and bounded them to the interval $\{1, 2, ..., 7\}$ by taking the values modulo 8. A few other modifications were made to ensure that all bitmasks have the correct size and that the result is formatted correctly. In the end we get a miniature version of the MD5 algorithm, though there is a problem: The rotation amounts for the original MD5 hash function were determined empirically to maximize the avalanche effect, so the new rotation amounts might not be optimal in our case, they would have to be verified.
The hash function itself should be similar in principle to MD5, but if we try to find a collision we only have to generate around $10^5$ random strings to find two with the same hash, which would indicate that the hash function is indeed not very robust in terms of collisions.

# Problem C

We build our random oracle hash using a random number generator and a python dictionary. When a new message enters the oracle, it checks if the message already exists in the dictionary, if it's the case it returns the corresponding hash, else it generates a random integer between 0 and $2^{32} - 1$ such that it can be represented with 32 bits and stores the pair (message, 32bits-integer) in the dictionary.
Finding a collision is easy and fast with brute-force.

# Problem D

## Explain how a good hash function can be used to protect users passwords.

The idea of passwords is for two parties (typically Alice and Bob) to share a secret (let's call it $Pw_A$), which is then used to correctly authenticate Alice to Bob. That is, every time Alice wants to prove to Bob that she is indeed Alice, she sends the pair [Alice, $Pw$] to Bob. If $Pw$ is the same as $Pw_A$ they agreed on earlier, Bob knows that she is indeed Alice because she is the only one that knows that password.
But how does Bob remember $Pw_A$? Easy: At the time Alice and Bob initially agree on $Pw_A$, he stores the pair [Alice, $Pw_A$] somewhere. Maybe he writes it on a sticky-note on his fridge. Later he can easily check whether the received $Pw$ is actually the password agreed on with Alice, by comparing it to the sticky-note on the fridge.
This works fine as long as he and Alice are the only persons that know the password. This becomes a problem when Bob invites Marc into his home, and Marc copies the sticky-note on the fridge. Now Marc can impersonate Alice, which is obviously very bad.
How does Bob avoid this? The crucial observation is that Bob only has to be able to *check* whether a provided string matches Alice's password. He does not have to know what the password actually is. Here is where hash-functions come into play. Bob can write the hashed password $H(Pw_A)$ on the sticky-note. Now instead of directly comparing $Pw$ and $Pw_A$, Bob compares $H(Pw)$ with the value he wrote on the sticky-note. Because of the determinism property of hash-functions, if $H(Pw)$ matches $H(Pw_A)$, it is practically certain[2] that $Pw = Pw_A$ and Bob can be sure it is indeed Alice (still assuming she is the only one that knows the password).
Now when Marc copies the sticky-note, he only gets $H(Pw_A)$. And because of the *non-invertability* of H, it is practically impossible for Marc to find $Pw_A$.

## How would you try to break it?

The naive way to break it is to try all possible values for the password and check for each whether its hash matches $H(Pw_A)$. This is of course extremely costly, because the search space is so big.

[1] https://rosettacode.org/wiki/MD5/Implementation#Python
[2] How certain depends on the hash-function.

Sidenote: let's assume $Pw_A$ has 16 characters (small and capital english letters and numbers), and assume Marc has the equivalent hash-power of the entire Bitcoin Network ($9 \times 10^{18}$ Hashes / second at time of writing). He would be expected to spend 85 years to find Alice's password by brute-force (170 years to try all 8 character passwords). And roughly $4.5 \times 10^{12}$ years if it is 22 characters long. [3]
But luckily for Marc, people don't choose long or random passwords. They tend to choose "real" words (words that have meaning) or at least passwords that are close to real words. Which leads us to the next question.

### What is a dictionary attack?

A dictionary attack is when Marc takes a list of words and tries them to find Alice's password. The trick is to create the list. As mentioned, passwords tend to be real words, so why not try a standard english (or any other language) dictionary? We can also include any lexicon or any other compilation of words. If Alice's password appears in any of those texts, Marc will find it much faster this way. However, most passwords are not exactly words from dictionaries. They have little alterations in them. For example the word "blockchain" can be used to generate "BloCkcHaIn" (capitalizing random letters), "bl0ckchain" (turn all "o" to zeros), "blockcha1n" ("i" becomes 1) etc... None of those alterations appear in a dictionary, but for a computer it is very easy to include such rules into the search.
In essence, a dictionary attack is taking a carefully chosen (and/or generated) list of words and try them. Note that this list may also include 'real' passwords that originate from previous leaks of passwords. So if Alice's password was stolen before and she uses the same password for different services[4], it is very likely that Marc will find her password this way.

## Problem E

Let us define the problem of finding a message $x$, which will produce a hash $y = h(x)$ with $n$ zeroes at the end. Prepending the message with some actual information, as is the case in bitcoin transaction blocks, does not make this problem harder to solve, due to the iterative nature of the MD5 algorithm. We have implemented a function, which randomly generates upper case strings of length 4, until it finds one that fulfills the aforementioned criterion. It is usually successful within a few ten thousand iterations.

[3] Formula for the number of years it takes to bruteforce all passwords of length $\leq L$ : $\frac{\sum_{k=1}^{L}(\text{number of possible characters})^k}{\text{number of hashes per year}} \Rightarrow \frac{\sum_{k=1}^{L}(26+26+10)^k}{10^{18}\times 60\times 60\times 24\times 365} \approx \frac{\sum_{k=1}^{L} 62^k}{2.8\times 10^{26}}$
[4] Never do that! Ever!