**Notations:** Input Vector \vec{u}*u*; Output Vector \vec{v}*v*; \vec{u},\vec{v} \in \mathbb{F}\_{128}^{8 \times 1}*u*,*v*∈F1288×1​; [E\_i],[A\_{ij}][*Ei*​],[*Aij*​] as given in question. By analysing the outputs of a couple thousand random inputs we noticed the following properties: 1. The input format is 2-letters-per-byte, i.e., the system takes blocks of multiples of 16 letters as input and outputs the same number of letters as the ciphertext. 2. The ciphertext space consists of outputs from \{ \text{ ff, fg, fh,\ldots, fu,\ldots, mf, mg,\ldots, mu } \}{ ff, fg, fh,…, fu,…, mf, mg,…, mu } only. That is, the first letter of each byte ranges from \textbf{f-m}**f-m** and the second ranges from \textbf{f-u}**f-u**. 3. Notice that this yields a set of 128 letter pairs, which can be perfectly mapped onto the bytes/elements of \mathbb{F}\_{128}F128​. Since, the DES assignment had mapping of (f-u) to (0-15) we continued with the same here as a starter. 4. Another thing to notice is that representing numbers from 0-127 requires only 7 bits, hence we guessed that the first bit of each byte should be 0. This restricts the first letter to be only from f-m, and was confirmed from the output frequency analysis. Now we started the square attack by taking various patterns as input sets (using input\_gen.py). Take \{\vec{u}^i\}{*ui*} as the set of 128 input vectors with pattern \textbf{C}**C** everywhere, but a \textbf{P}**P** at the i-th byte u\_i*ui*​. Now from a collection of such input sets with varying C*C*-byte values, we noticed the folowing: In the outputs of any input set \{\vec{u}^i\}{*ui*}, all the bytes \textbf{after}**after** the i*i*-th byte were changed. Now if we analyse the EAEAE*EAEAE* operation carefully in order, we notice that this will happen only if in A*A*, A\_{ij}=0*Aij*​=0 whenever j<i*j*<*i*, for any i*i*. Therefore, A*A* is a \textbf{lower triangular matrix}**lower triangular matrix**. Then for the next set we kept all C*C* bytes 00, (or ff in plaintext). And we calculated the theoretically expected output by applying EAEAE*EAEAE* on this set. Then, v^i\_i = (A\_{ii} \cdot (A\_{ii} \cdot (u\_i^i)^{E\_i})^{E\_i})^{E\_i} = A\_{ii}^{E\_i+E\_i^2} \cdot (u\_i^i)^{E\_i^3} \quad \forall i \text{ s.t. }1 \leq i \leq 8*vii*​=(*Aii*​⋅(*Aii*​⋅(*uii*​)*Ei*​)*Ei*​)*Ei*​=*AiiEi*​+*Ei*2​​⋅(*uii*​)*Ei*3​∀*i* s.t. 1≤*i*≤8 (\*This is clearly true for i=1*i*=1, and we can use simple induction for other i easily.\*) We wrote a code (pair\_gen.py) to iterate the values of A\_{ii} \in \{0,1,\ldots,127\}*Aii*​∈{0,1,…,127} and E\_i \in \{1,2,\ldots,126\}*Ei*​∈{1,2,…,126} and check if the expected v\_i^i*vii*​ matches the actual outputs obtained from the game for \textbf{all}**all** 128 vectors in the input set. Running this for 1 \leq i \leq 81≤*i*≤8, we obtain the following possible (E\_i,A\_{ii})(*Ei*​,*Aii*​) pairs for each i*i* satisfying the above condition: i=1 37 40 68 49 22 84 i=2 58 8 110 70 86 115 i=3 43 43 55 86 29 108 i=4 44 11 74 12 9 31 i=5 105 67 59 109 90 112 i=6 53 11 83 41 118 127 i=7 8 16 25 27 94 28 i=8 25 38 94 71 8 126 Finally, to single out the exact values of A\_{ii},E\_i*Aii*​,*Ei*​ and in the process, A\_{ij}*Aij*​ also, we will run another \sim 40∼40 input sets with similar patterns, but changing the row prefixes consisting of C*C*s from 00 to some other non-zero bytes so that the value of A\_{ij}*Aij*​ can also be found out recursively in backward order from the main diagonal. (For example, notice that we can obtain the value of A\_{i,i-1}*Ai*,*i*−1​ using the values A\_{ii}*Aii*​ and A\_{i-1,i-1}*Ai*−1,*i*−1​ and so on using a wisely chosen input set) Finally, we arrived at our final A*A* and E*E* matrices as: E= [22 \; 110 \; 43 \; 74 \; 90 \; 53 \; 25 \; 25]^T \qquad \\ A*E*=[22110437490532525]*TA*= 84 0 0 0 0 0 0 0 112 70 0 0 0 0 0 0 22 27 43 0 0 0 0 0 99 22 25 12 0 0 0 0 100 61 3 112 112 0 0 0 29 38 19 52 97 11 0 0 23 123 12 98 28 94 27 0 89 8 73 25 18 70 10 38 Thus, the last step remaining is to decode the password ciphertext by inverting all the operations. Since det(A) \neq 0*det*(*A*)​=0, it can be inverted easily. Also, the inverse exponentiation can be done using the properties of the field. \text{password} = E^{-1}A^{-1}E^{-1}A^{-1}E^{-1}(\vec{c})password=*E*−1*A*−1*E*−1*A*−1*E*−1(*c*) Our encrypted password \vec{c}*c* was \textbf{gsjmjmgtkulqlpjlgkfufmgiftgpkpfh}**gsjmjmgtkulqlpjlgkfufmgiftgpkpfh**, which on decryption yielded \textbf{mllumjmflhmpmhmlljlmifififififif}**mllumjmflhmpmhmlljlmifififififif**. Expecting a <16<16 letter password, we felt that one more step of decryption was left. Also, the tail of our password consists of "\textbf{if}**if**"s which represents 4848 in the mapping. The character with ASCII code 4848 is 00, and zeros have been used for padding as always (\textbf{votpbzrvdg000000}**votpbzrvdg000000**). Thus we converted the 3232 letters to the mapping, then to ASCII characters. Hence, our final password is \textbf{votpbzrvdg}**votpbzrvdg**. \textbf{Note 1:}**Note 1:** One noticeable thing about the plaintext space of this assignment was that the game took inputs of the form "$#", where $ ranged from (f-o) and # ranged from (f-z), else the connection was breaking. On analysing inputs outside (ff-mu), we found that the distance of the letters from f were considered for the mapping. And anything that gave the byte a value of more than 160 broke the connection. Example: the input "fzfzfzfzfzfzfzfz" gave the output "ightmkmilihlkrhf", which on decryption yielded "gigigigigigigigi". Notice that gi->20 and fz->20 also according to the mapping. This is what confuses the cryptosystem. There were some other anomalies when a byte value exceeded 127. For example, we also observed the following input-output-decrypted text tuples: (nfnfnfnfnfnfnfnf -> jhhnfumqgglfkoju; nfffffffffffffff -> jhhfmmijfrgmgfin; ffnfffffffffffff -> fffglsfrmffqjoht; ffffffffffffffnf -> fffffffffffffffu) It was clear from multiple such observations that encryption does not go inter-byte in any case, but the decimal values get affected in case the letter pairs fall out of the (ff-mu) range. \textbf{Note 2:}**Note 2:** We implemented the multiplication modulo irreducible polynomial using bitwise operations in the functions "mul" and "exp" in the file pair\_gen.py. Flow of Codes: input\_gen\_pairs.py -> inputs$i.txt -> script\_gen.cpp -> script.sh -> grep.py -> outputs$i.txt -> merge.sh -> final$i.txt -> pair\_gen.py -> pairs$i.txt ; input\_gen\_decrypt.py -> inputdir/$i/inputs$j.txt -> script\_gen\_mod.cpp -> script\_mod.sh -> grep\_mod.py -> outputdir/$i/output$j.txt -> merge\_decrypt.sh -> finaldir/$i/final$j.txt -> decrypt.py ( + \sum∑pairs$i.txt + password ciphertext) -> A, E, password plaintext

Answer Password: votpbzrvdg