I saw a video from LiveOverflow showing this cool hiring problem from Nintendo where you essentially have to reverse some output to an input. When the video went up, some commenters noticed that linear algebra was involved which really captured my attention, so I decided to tackle the problem. Man, was it a cool problem, and I think it is a great place to introduce some basic math concepts. We will build the cracking program from scratch with nothing more than Python. We will also not shy away from the math behind the solution, but don't worry. Math often is viewed as an arcane, nebulous art that is difficult. You might have this position, and it is my goal in this video to hopefully challenge that view! Math is a beautiful field and is a tool for us to reason about complex systems such as the one we are about to see! The beautiful part of this problem is that it highlights the intersection of computer science and math! Let's get started.

In this video I will introduce to you

1. Functions
2. Preimages and Inverse Functions
3. Cartesian Products
4. Modular Arithmetic
5. Matrices

These all may sound scary but in time you will see they are very approachable concepts.

Let's first split up the Forward() function into 3 visually identifiable stages.

```
void Forward(u8 c[32], u8 d[32], u8 s[512], u32 p[32]) {
    for(u32 i = 0; i < 256; i++) {
        for(u8 j = 0; j < 32; j++) {
            d[j] = s[c[j]];
            c[j] = 0;
        }

        for(u8 j = 0; j < 32; j++)
            for(u8 k = 0; k < 32; k++)
                c[j] ^= d[k] * ((p[j] >> k) & 1);
    }

    for(u8 i = 0; i < 16; i++)
        d[i] = s[c[i * 2]] ^ s[c[i * 2 + 1] + 256];
}
```

σ_1

σ_2

σ_3

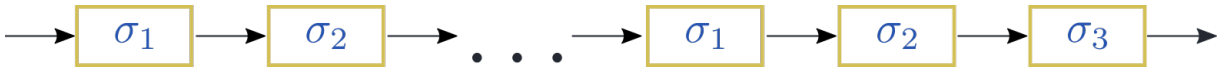
The first stage essentially takes a list of bytes called **c** and substitutes each value in **c**. This new list is now called **d**. Formally, we will call **c** and **d** "vectors" for reasons that will be more apparent later. We will also call the operation stage 1 performs on **c**: σ_1. σ_1 is what we in the math world call a "function." A function is in essence a mapping from inputs to outputs. Think back to your early schooling. You may have seen

$$f(x) = x^2$$

which means take some number x and perform a computation on it, namely square it. This f function is like a machine that consumes an x and spits out a new value. Continuing this analogy, we are abstracting away the complexities of stage 1 into this new guy σ_1 which is like a machine that performs some computation on $\mathbf{c}$. This is what mathematicians do all the time. Take complex ideas and give them names so that we can more easily talk about them. As for why I decided to call stage 1 σ_1, mathematicians typically use Greek letters which are faster to write. Also the words "stage" and "sigma" both start with "s" making it easy to remember. Additionally σ will stand out and cue to us that we are dealing with math and not code.

Continuing on, the second stage performs this "scrambling operation" which we will call σ_2. Finally the last stage which we will call σ_3 substitutes each value in $\mathbf{c}$ and compresses two consecutive elements into one element in $\mathbf{d}$.

Graphically, we can think of the input data transforming as it goes through each stage, and when put together, we get a kind of data transformation pipeline as shown. How do we reverse this pipeline whereby we can reconstruct the input given some output



at the end of the pipe? Maybe it will help to think about a smaller situation. Consider two operations A and B. We will denote performing operation B then A as

$$AB$$

and denote the "reverse" operation as

$$(AB)^{-1}.$$

This is read as "AB inverse". As for why we write A and B in the opposite order will be explained later. But first, to understand the inverse we need to understand what an "identity" operation is, which we will call I. The idea is quite simple. The identity operation is the operation that does nothing to the input. In other words it maps the input to itself. So consider

$$f(x) = x.$$

This is the identity operation as running the f machine on any input gives back the original input. Notice how using this function notation we write the input, being operated on, on the right side. This explains the opposite order for A and B as when we perform operation AB on some input x we will write it as

$$(AB)x$$

which looks like we are first executing B on x. Then we execute A on Bx.

Now given some operation A we say that A^{-1} is the operation that brings you back to original input after applying A (and vice versa). In other words

$$A^{-1}A = I = AA^{-1}$$

So imagine rotating an image 90 degrees clockwise. The inverse operation is rotating the image 90 degrees counterclockwise as performing both operations in either order will be

as if you did nothing, also known as the identity operation. Back to our original question, what is

$$(AB)^{-1}?$$

It turns out that it is

$$(AB)^{-1} = B^{-1}A^{-1}$$

Intuitively this makes sense. Consider putting on your socks (call this B) then putting on your shoes (call this A). The combination of putting on your socks then your shoes is AB. The inverse operation is thus taking off your shoes A^{-1} then taking off your socks B^{-1} which is none other than $B^{-1}A^{-1}$! All this formalism was to establish a key idea used throughout this entire video: the reverse of a sequence of the steps is the inverse of each step done in reverse order!

Looking at our pipeline from before, we can think of the entire data transformation as

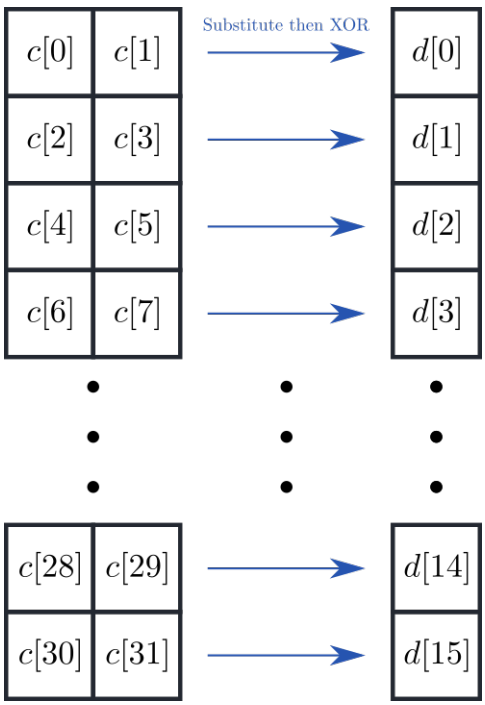
$$\sigma_3\sigma_2\sigma_1 \ldots \sigma_2\sigma_1$$

where $\sigma_2\sigma_1$ is repeated 256 times. Using the insight above, the inverse of this entire process is

$$(\sigma_3\sigma_2\sigma_1 \ldots \sigma_2\sigma_1)^{-1} = \sigma_1^{-1}\sigma_2^{-1} \ldots \sigma_1^{-1}\sigma_2^{-1}\sigma_3^{-1}.$$

Ok, we now have a place to start. We now just need to figure out σ_1^{-1},σ_2^{-1}, and σ_3^{-1} and from there we can reverse the entire process. Let's now figure out σ_3^{-1} as remember that the first operation performed is on the far right of the expression.

In a picture this is what stage 3 does. It looks at two consecutive values in **c** and



substitutes them for some value. These new values are then XORed together. How do we

reverse this process? Rather than reversing the entire vector **d**, how about reversing each entry in **d** to two consecutive values in **c**? This is actually a lot easier. Given some $d[i]$, we can just brute force all possible pairs of c values and see which turns into $d[i]$. This is not so bad as **c** is an array of 8 bit integers meaning we only need to test $256^2 = 65536$ values which can be quickly done on a computer.

```
def find_xor_match(d_value):
    res = []
    for c1 in range(256):
        for c2 in range(256):
            if confusion[c1] ^ confusion[c2 + 256] == d_value:
                res.append([c1, c2])
    return res
```

We now just have to reverse each entry in **d** and then derive some vector that when passed through stage 3 or σ_3 will output **d**. It turns out that the values of **d** we care about are

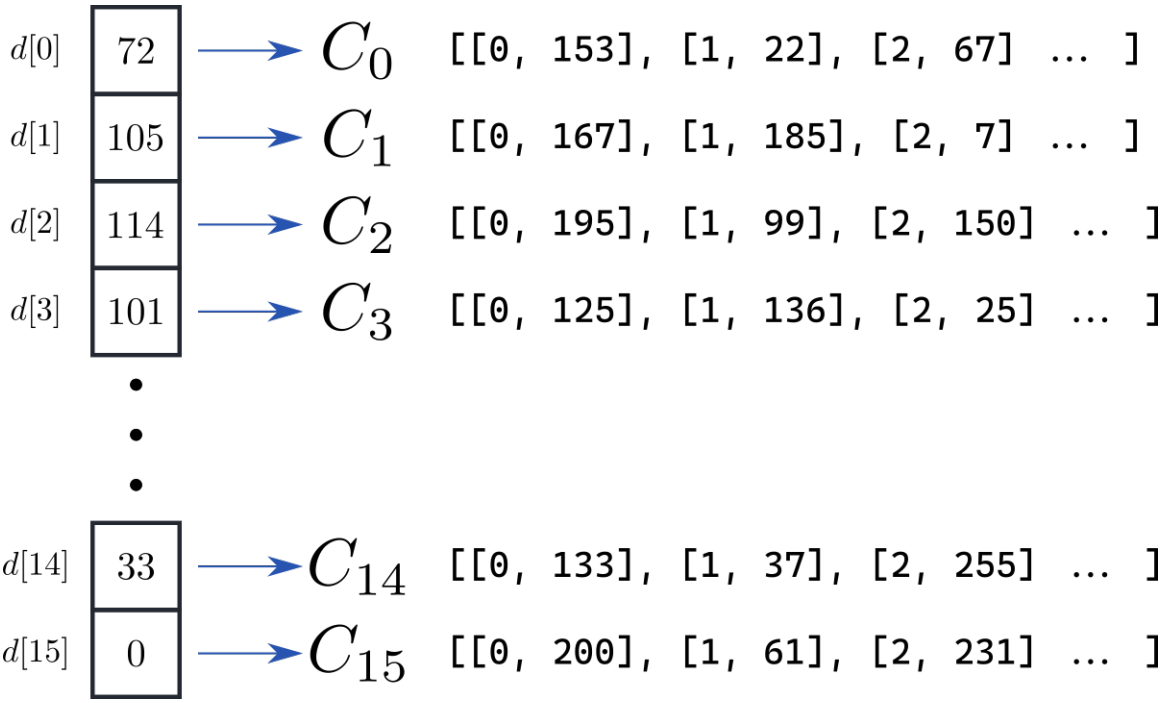
$$\text{“Hire me!!!!!!!!”} \implies [72, 105, 114, 101, 32, 109, 101, 33, 33, 33, 33, 33, 33, 33, 33]$$

which can be found by converting the string into a byte array using

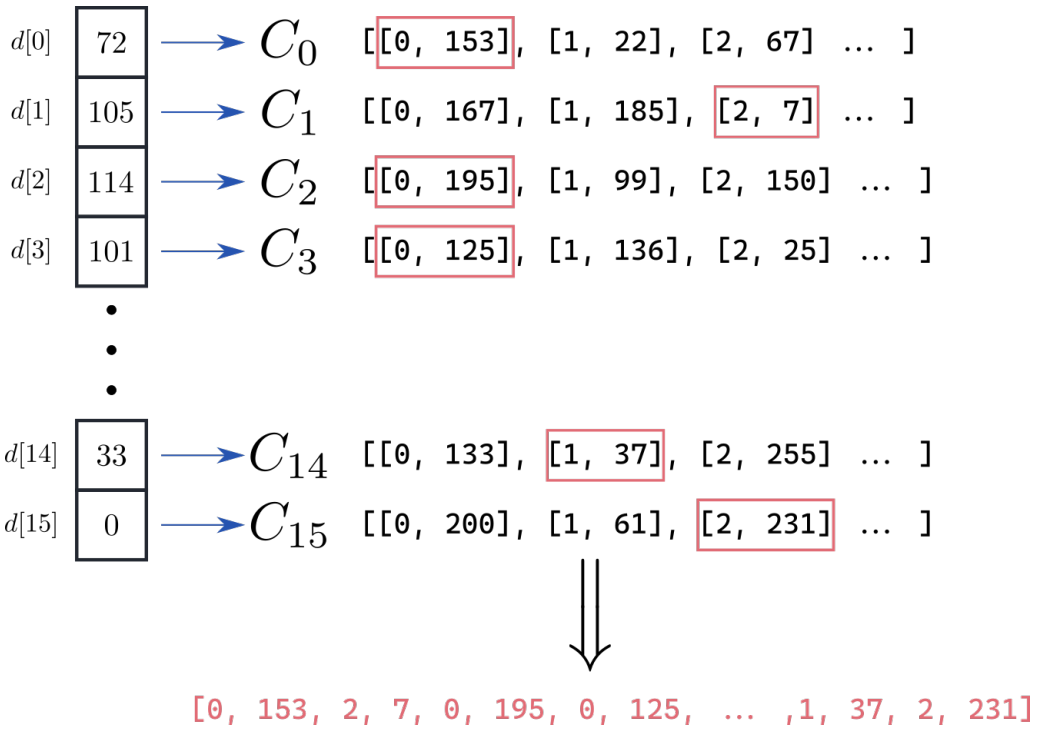
```
import array

list(array.array('B', b"Hire me!!!!!!!!"))
```

Reversing each $d[i]$ gives us a list of pairs that transform into $d[i]$. Call this list C_i. A portion of each C_i is shown. Now notice that selecting any element from each C_i



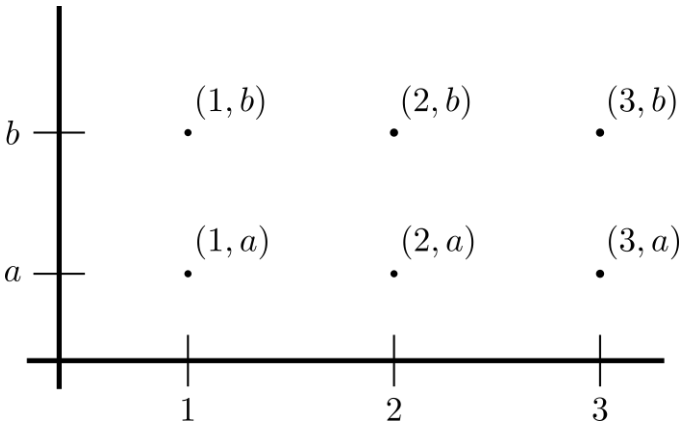
and mushing them together into one long vector will be a vector that turns into **d** after passing through stage 3 or σ_3! This idea of picking one element from a ordered list of sets and “mushing them” into one long list has a name. This is also known as the Cartesian



product. Let A and B be sets, which you can think of as a collection of elements. We say $A \times B$ is the Cartesian product of A and B representing the set of all pairs (a, b) where a is in A and b is in B. For example

$$\{1, 2, 3\} \times \{a, b\} = \{(1, a), (1, b), (2, a), (2, b), (3, a), (3, b)\}$$

Geometrically, you can think of the Cartesian product as grid points.



Do you see how reversing stage 3 involves a Cartesian product? Rather than taking the Cartesian product of 2 sets, we are taking the Cartesian product of 16 sets. Unfortunately I cannot draw a 16 dimensional space. Explaining the Cartesian produce is useful because it turns out that Python has a nice function in itertools that allows us to iterate through a Cartesian product! We will use this to bruteforce the final answer which is why knowing names and terminology comes in handy! You can easily look up how to do something when you know its proper name.

In any event, given a vector $\mathbf{d}$ we can compute all vectors $\mathbf{c}$ whose transformation under σ_3 or stage 3 is $\mathbf{d}$. We write this idea using math notation as such:

$$\sigma_3^{-1}(\mathbf{d}) = \text{flatten}(C_0 \times C_1 \times \ldots \times C_{14} \times C_{15}).$$

where flatten just takes our Cartesian product and smush it into one long vector as technically the elements in C_i are pairs and the Cartesian product results in a list of pairs as opposed to one contiguous vector. In code this is how we reverse stage 3.

```
from itertools import product

target = list(array.array('B', b"Hire me!!!!!!!!")) + [0]
individual_preimages = list(map(find_xor_match, target))
for cartesian_product in product(*individual_preimages):
    stage_3_preimage = flatten(cartesian_product)
```

I will now introduce new terminology. The equation shown is known as the "preimage (or inverse image) of $\mathbf{d}$ under σ_3." In essence, the preimage of an output value is the set of all input values whose transformation gives you your output value. Because this idea of a preimage is so close to the idea of an inverse operation, we will use the same notation. Therefore with this new idea, we can reformulate our original problem as finding the preimage of

$$\text{"Hire me!!!!!!!!"} = [72, 105, 114, 101, 32, 109, 101, 33, 33, 33, 33, 33, 33, 33, 33]$$

under the map

$$\sigma_3\sigma_2\sigma_1 \ldots \sigma_2\sigma_1!$$

This terminology will be useful as we move on to reversing stage 2 and stage 1.

The key insight to reversing stage 2 is that we are basically solving a system of equations. You may have seen linear system of equations in school looking like

$$\begin{aligned} 2x + y + \ z &= \ 7 \\ -\ y + 3z &= \ 5 \\ 4x + \ + \ z &= 10 \end{aligned}$$

We can succinctly write our system of equations in matrix form as

$$\begin{bmatrix} 2 & 1 & 1 \\ 0 & -1 & 3 \\ 4 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 7 \\ 5 \\ 10 \end{bmatrix}$$

Notice how we go from the matrix representation to the original system of equations. We do this strange left-right, up-down multiplication. This strange multiplication move will come in handy later.

The basic idea of solving these systems is that we would choose a variable we want to solve for, then eliminate other occurrences in other equations. The name for the variable chosen in one equation is called the pivot. We will box all pivots. Let's work an example. Let's pick x in the first equation and eliminate the other occurrences.

$$\begin{aligned} \boxed{2x} + y + \ z &= \ 7 \\ -\ y + 3z &= \ 5 \\ 4x + \ + \ z &= 10 \end{aligned}$$

We can do this by multiplying the first equation by -2 then adding it with equation 3 giving

$$\begin{aligned} \boxed{2x} + \ y + \ z &= \ 7 \\ - \ y + 3z &= \ 5 \\ -2y - \ z &= -4 \end{aligned}$$

In matrix form this is

$$\begin{bmatrix} 2 & 1 & 1 \\ 0 & -1 & 3 \\ 0 & -2 & -1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 7 \\ 5 \\ -4 \end{bmatrix}$$

Notice what we did. We multiplied equation 1 by -2 which is legitimate. Suppose $x = 10$ then obviously $2x = 20$ which occurs by multiplying by 2 on both sides. This special choice of -2 allowed us to cancel out the x term in equation 3 when they were added together! Adding two equations together is also legitimate. Consider if $x = 10$ and $y = 5$, then obviously $x + y = 10 + 5 = 15$.The same logic applies. We can also clearly see the elimination occurring in the matrix form where there are zeroes in the first column under the pivot. We can now continue this elimination trick by now selecting y in equation 2.

$$\begin{aligned} 2x + \ y + \ z &= \ 7 \\ - \boxed{y} + 3z &= \ 5 \\ -2y - \ z &= -4 \end{aligned}$$

We can cancel the y in equation 1 by simply adding both equation 1 and equation 2 giving

$$\begin{aligned} 2x + \quad + 4z &= \ 12 \\ - \boxed{y} + 3z &= \ 5 \\ -2y - \ z &= -4 \end{aligned}$$

We can eliminate the y term in equation 3 by multiplying equation 2 by -2 then adding both equations giving

$$\begin{aligned} 2x + \quad + 4z &= \ 12 \\ - \boxed{y} + 3z &= \ 5 \\ - 7z &= -14 \end{aligned}$$

which is the same as

$$\begin{aligned} x + \ + 2z &= \ 6 \\ y - 3z &= -5 \\ z &= \ 2 \end{aligned}$$

which we got by diving all equations by their pivots. In matrix form this is

$$\begin{bmatrix} 1 & 0 & 2 \\ 0 & 1 & -3 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 6 \\ -5 \\ 2 \end{bmatrix}$$

where we see zeroes above and below our pivot in column 2. We can now eliminate z giving

$$x = 2$$
$$y = 1$$
$$z = 2$$

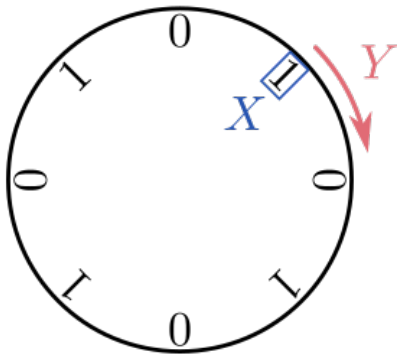
or

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 2 \\ 1 \\ 2 \end{bmatrix}$$

Do you see how in the end the final matrix is a bunch of ones down the diagonal and zeroes everywhere else? Once we have manipulated our matrix into this form, we are done and we can solve the system of equations! This special matrix is called the identity matrix because when you do the strange multiplication with this matrix you get your original input! Therefore, the essence of the elimination method is to get to the identity matrix through a series of row operations.

Now that we have a feel for how elimination works, we will use this same technique to solve a way bigger system with a twist whereby we use a different kind of addition, called XOR. Let's take a look at the XOR operation. There are a few ways of interpreting the XOR operation $\bigoplus$ which we will denote as this circle plus symbol. Typically

X \ Y	0	1
0	0	1
1	1	0



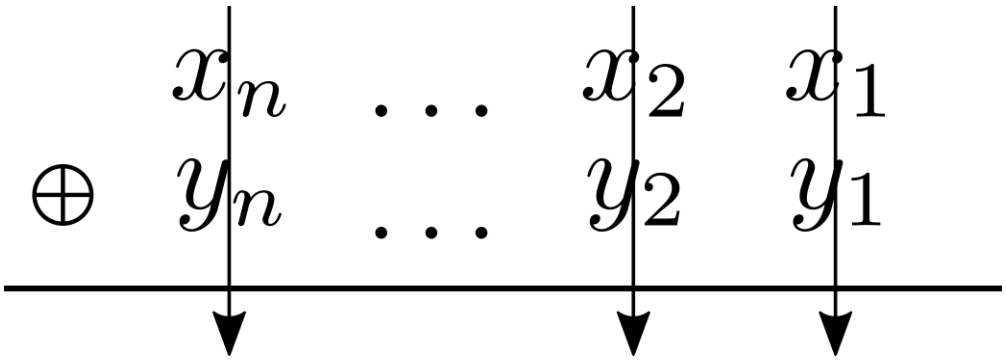
$$X \bigoplus Y$$

where X and Y are either 0 or 1 is described as 0 if both X and Y are the same and 1 if they are different as shown in the table above. Another interpretation is using modular arithmetic whereby XORing two numbers is known as "addition modulo 2". The idea is that when you add two numbers you think of a clock where once you reach past 1 you wrap back to 0 similar to a clock wrapping back to 1 after 12. What's nicer about the second interpretation is that we can easily see that all our nice properties of regular addition holds with XOR. Namely we have the associativity and commutativity property. We can also notice that 0 acts like our regular zero where any number plus 0 is the original number. 0 is also known as the additive identity element because of this property. With our identity element, we can now define an inverse element! Notice with XOR, the inverse element of any element is the element itself! Any number XORed with itself gives back the identity element 0. You can verify this using the table or the clock diagram. We

can ground these terms in our regular addition, where the additive inverse is simply the negative of our number. For example. the additive inverse of 5 is -5 as

$$5 + (-5) = 0 = (-5) + 5$$

We can extend the XOR operation to dealing with vectors where we instead apply the XOR operation on each entry in the vector.



This exposition all leads to why elimination works. We are taking our pivot and multiplying it by a value turning it into the additive inverse of the term we are eliminating so that when we add the two equations, the term we want eliminated disappears! Looking closely at stage 2, we see a very familiar picture by unwrapping the for loops. It looks just like a system of equations we saw before but using XOR as opposed to regular addition!

$$((p[0] >> 0)\&1)\mathbf{d}[\mathbf{0}] \bigoplus ((p[0] >> 1)\&1)\mathbf{d}[\mathbf{1}] \bigoplus ((p[0] >> 2)\&1)\mathbf{d}[\mathbf{2}] \ldots = c[0]$$
$$((p[1] >> 0)\&1)\mathbf{d}[\mathbf{0}] \bigoplus ((p[1] >> 1)\&1)\mathbf{d}[\mathbf{1}] \bigoplus ((p[1] >> 2)\&1)\mathbf{d}[\mathbf{2}] \ldots = c[1]$$
$$((p[2] >> 0)\&1)\mathbf{d}[\mathbf{0}] \bigoplus ((p[2] >> 1)\&1)\mathbf{d}[\mathbf{1}] \bigoplus ((p[2] >> 2)\&1)\mathbf{d}[\mathbf{2}] \ldots = c[2]$$

Substituting those values of $((p[j] >> k)\&1)$ we get the follow snapshot of the system we will solve

$$1 \cdot \mathbf{d}[\mathbf{0}] \bigoplus 0 \cdot \mathbf{d}[\mathbf{1}] \bigoplus 0 \cdot \mathbf{d}[\mathbf{2}] \ldots = c[0]$$
$$0 \cdot \mathbf{d}[\mathbf{0}] \bigoplus 1 \cdot \mathbf{d}[\mathbf{1}] \bigoplus 0 \cdot \mathbf{d}[\mathbf{2}] \ldots = c[1]$$
$$0 \cdot \mathbf{d}[\mathbf{0}] \bigoplus 0 \cdot \mathbf{d}[\mathbf{1}] \bigoplus 1 \cdot \mathbf{d}[\mathbf{2}] \ldots = c[2]$$

Using the more succinct matrix notation we end up getting

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & \ldots & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & \ldots & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & \ldots & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & \ldots & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & \ldots & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 1 & \ldots & 1 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 1 & 1 & 0 & 0 & 0 & 0 & \ldots & 1 & 0 \\ 1 & 1 & 1 & 0 & 1 & 0 & \ldots & 0 & 1 \end{bmatrix} \begin{bmatrix} \mathbf{d}[\mathbf{0}] \\ \mathbf{d}[\mathbf{1}] \\ \mathbf{d}[\mathbf{2}] \\ \mathbf{d}[\mathbf{3}] \\ \mathbf{d}[\mathbf{4}] \\ \mathbf{d}[\mathbf{5}] \\ \mathbf{d}[\mathbf{6}] \\ \vdots \\ \mathbf{d}[\mathbf{30}] \\ \mathbf{d}[\mathbf{31}] \end{bmatrix} = \begin{bmatrix} \mathbf{c}[\mathbf{0}] \\ \mathbf{c}[\mathbf{1}] \\ \mathbf{c}[\mathbf{2}] \\ \mathbf{c}[\mathbf{3}] \\ \mathbf{c}[\mathbf{4}] \\ \mathbf{c}[\mathbf{5}] \\ \mathbf{c}[\mathbf{6}] \\ \vdots \\ \mathbf{c}[\mathbf{30}] \\ \mathbf{c}[\mathbf{31}] \end{bmatrix}$$

which we can generate using

```
import numpy as np

def generate_matrix():
    matrix = np.zeros((32, 32), dtype=int)
    for j in range(32):
        for k in range(32):
            matrix[j][k] = (diff[j] >> k) & 1
    return matrix
```

The format of the matrix is that the entry at row i and column j is

$$(p[i] >> j)\&1$$

We will call the matrix A. We can reformulate the above expression as

$$A\mathbf{d} = \mathbf{c}$$

and to solve d given some c we need to perform elimination similar to what we did before. However, doing elimination every time is extremely inefficient, and it turns out we can pack all the elimination steps into a single matrix. It turns out from linear algebra that if you perform your row operations on both A and the identity matrix I simultaneously, I will turn into a new matrix. This new matrix turns out to be A^{-1}. Once we calculate A^{-1}, we can easily solve

$$\begin{aligned} A\mathbf{d} &= \mathbf{c} \\ A^{-1}A\mathbf{d} &= A^{-1}\mathbf{c} \\ I\mathbf{d} &= A^{-1}\mathbf{c} \\ \mathbf{d} &= A^{-1}\mathbf{c} \end{aligned}$$

While this is not the full proof, the idea that we can get A^{-1} by reflecting the row operations onto the identity matrix can be seen by thinking of our row operations as E_i. Performing our row operations we get

$$E_nE_{n-1}\dots E_2E_1A\mathbf{d} = E_nE_{n-1}\dots E_2E_1\mathbf{c}$$

resulting in

$$E_nE_{n-1}\dots E_2E_1A\mathbf{d} = I\mathbf{d} = \mathbf{d} = E_nE_{n-1}\dots E_2E_1\mathbf{c}$$

and from our definition of the inverse it is plausible that

$$E_nE_{n-1}\dots E_2E_1$$

is our inverse of A as it gives identity after multiplying with A. We just need to replay these elimination actions onto the identity matrix so that we get a matrix representation of

$$E_nE_{n-1}\dots E_2E_1$$

The actual proof however is much more technical but this is the high level overview.

Elimination using XOR surprisingly is easier than elimination using regular addition. We can eliminate a row by checking if it has a 1 below our pivot and simply performing

bitwise XOR on both rows. Remember that the additive inverse of a bit is itself. Let's look at an example

$$\begin{bmatrix} \boxed{1} & 0 & 0 & 0 & 0 & 0 & \dots & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & \dots & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & \dots & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & \dots & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & \dots & 0 & 0 \\ \mathbf{1} & 0 & 0 & 1 & 0 & 1 & \dots & 1 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 1 & 1 & 0 & 0 & 0 & 0 & \dots & 1 & 0 \\ 1 & 1 & 1 & 0 & 1 & 0 & \dots & 0 & 1 \end{bmatrix}$$

The boxed number will be our pivot and we will eliminate down. Notice that row 6 has a 1 (in bold) below our pivot. Adding row 1 and row 6 will eliminate that value under our pivot in row 6 giving

$$\begin{bmatrix} \boxed{1} & 0 & 0 & 0 & 0 & 0 & \dots & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & \dots & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & \dots & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & \dots & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & \dots & 0 & 0 \\ \mathbf{0} & 0 & 0 & 1 & 0 & 1 & \dots & 0 & 1 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 1 & 1 & 0 & 0 & 0 & 0 & \dots & 1 & 0 \\ 1 & 1 & 1 & 0 & 1 & 0 & \dots & 0 & 1 \end{bmatrix}$$

We then repeat this procedure for every pivot giving us the final identity matrix. In code, computing the inverse using elimination looks like

```
# Swap row i and j
def swap(matrix, i, j):
    buf = np.copy(matrix[i])
    matrix[i] = np.copy(matrix[j])
    matrix[j] = buf

# Add row i to row j and keep row i the same
def add(matrix, i, j):
    matrix[j] ^= matrix[i]

def invert(matrix):
    res = np.eye(32, dtype=int)
    current = np.copy(matrix)
    for i in range(32):
        pivot = current[i][i]
        # Perform a swap
        if pivot == 0:
            # Scan down for a row
            for j in range(i+1, 32):
```

```
            if current[j][i] == 1:
                pivot = current[j][i]
                swap(current, i, j)
                swap(res, i, j)
                break
        # Eliminate down
        for j in range(i+1, 32):
            if current[j][i] == 1:
                add(current, i, j)
                add(res, i, j)
        # Eliminate up
        for j in range(0, i):
            if current[j][i] == 1:
                add(current, i, j)
                add(res, i, j)

    return res
```

The swapping action is a technicality I have decided to avoid, but the idea is that we never want our pivot to be 0 as this would prevent us from eliminating other rows. Thus we have to swap out the offending row for a new row with a 1 in the pivot position. We are almost done reversing stage 2. We just need a way to multiply a matrix with the vector **c** so that we can recover **d**. The code below shows how we do our strange matrix multiplication.

```
def multiply(matrix, vector):
    res = [0] * 32
    for i in range(32):
        sum_entry = 0
        for j in range(32):
            sum_entry ^= matrix[i][j] * vector[j]
        res[i] = sum_entry
    return res
```

We are now done reversing stage 2 or σ_2. Therefore we can now compute the preimage of **c** under σ_2 to recover **d**.

$$\sigma_2^{-1}(\mathbf{c}) = A^{-1}\mathbf{c}$$

We are almost done. All we have left to do is reverse stage 1 which will be a lot quicker now that we have establish all the mathematical foundation. Stage 1 or σ_1 simply performs a substitution for each value in **c** and we will use a similar technique to the one done in reversing stage 3. What we will do instead is build a lookup table. This lookup table will give us a list of characters whose substitution leads to the given value.

Notice that each character is substituted for some 8 bit integer. That means our lookup table needs to have 256 slots, one for each 8 bit integer. We can populate our slots by iterating through all characters and inserting it into its corresponding slot.

```
def build_lookup_table():
    lookup_table = [[] for _ in range(256)]
    for c in range(256):
```

```
            lookup_table[confusion[c]].append(c)
    return lookup_table
```

We basically just computed the preimage of every integer under the substitution by tracking how the substitution maps each character! We could then compute the preimage of an entire vector under the substitution by using a Cartesian product as seen in stage 3. Do note that the preimage may in fact be empty as it is entirely possible that there exists no vector whose substitution gives what we need. Notice how $0xfa$ never appears in the confusion array which means $0xfa$ has no preimage under the substitution.

Ok. We have now laid the groundwork to build a program that cracks the input needed to give our hire me text.

Let's look back at the operations we need to execute.

$$\begin{aligned}(\sigma_3\sigma_2\sigma_1\ldots\sigma_2\sigma_1)^{-1} &= \sigma_1^{-1}\sigma_2^{-1}\ldots\sigma_1^{-1}\sigma_2^{-1}\sigma_3^{-1}\\ &= \left(\sigma_1^{-1}\sigma_2^{-1}\right)^{256}\sigma_3^{-1}\end{aligned}$$

We need to compute the preimage of our output under σ_3. Then taking that preimage, we need to repeatedly compute its preimage under stage 1 and 2 signified as

$$\left(\sigma_1^{-1}\sigma_2^{-1}\right)^{256}$$

which should reverse the entire process giving us our desired result.

```
def flatten(cartesian_product):
    res = []
    for pair in cartesian_product:
        res.extend(pair)
    return res

def compute_stage_1_2_preimage(lookup_table, inverse, targets):
    preimages = []
    for target in targets:
        stage_2_preimage = multiply(inverse, target)
        stage_1_cartesian_product = list(
            map(
                lambda c: lookup_table[c],
                stage_2_preimage
            )
        )
        has_stage_1_preimage = True
        for character_preimage in stage_1_cartesian_product:
            if len(character_preimage) == 0:
                has_stage_1_preimage = False
                break
        if has_stage_1_preimage:
            for stage_1_preimage in product(*stage_1_cartesian_product):
                preimages.append(list(stage_1_preimage))
    return preimages

stage_1_lookup = build_lookup_table()
```

```
stage_2_inverse = invert(generate_matrix())

target = list(array.array('B', b"Hire me!!!!!!!!")) + [0]
individual_preimages = list(map(find_xor_match, target))
for cartesian_product in product(*individual_preimages):
    stage_3_preimage = flatten(cartesian_product)
    targets = [stage_3_preimage]
    for _ in range(256):
        targets = compute_stage_1_2_preimage(stage_1_lookup, stage_2_invers

    if len(targets) > 0:
        print(targets[0])
        break
```

It will take a while to get the final answer because of how inefficiently this code is written and how slow Python is. Rewriting this program in a faster language will likely yield a much more performant program which was the case when I rewrote it in Rust. However, the goal of this video is not to solve this problem the fastest but to move through the thought process for how you could solve this problem and to showcase some cool math. Hopefully, you enjoyed this video and learned some math along the way. All code will be posted to Github along with a transcript of what was said in this video.