

Pre-Alpha Rough Draft FFE 0.0.0.8...

This tutorial will show exactly how to get us all on the same page.

Let us define a plaintext (P) where (P[n]) represents each of its (n) bytes from (0...n-1).

Here is a Python program that does this:

```
# Example 0: Plaintext
#
# P is the plaintext
P = "This is plaintext";

# n is the size of P in bytes
n = len(P);

print("P:%s" % (P));
print("n:%s" % (n));

for i in range(n):
    print("P[%s]:'%s'" % (i, P[i]));
```

The output of the code above is:

```
P:This is plaintext
n:17
P[0]:'T'
P[1]:'h'
P[2]:'i'
P[3]:'s'
P[4]:' '
P[5]:'i'
P[6]:'s'
P[7]:' '
P[8]:'p'
P[9]:'l'
P[10]:'a'
P[11]:'i'
P[12]:'n'
P[13]:'t'
P[14]:'e'
P[15]:'x'
P[16]:'t'
```

Now we create an integer based 2d plane (P_plane) around the plaintext (P) with dimensions (P_dims) equal to the ceiling of the square root of (n). We derive the integer x-coordinate by taking an index i in (P[n]) and mod it with (P_dims). The integer y-coordinate is computed by taking the floor of i divided by (P_dims). This square mapping is completely arbitrary for now. I am so used to plotting fractals on rectangles or squares for graphs, or perhaps something pretty enough to hang on a wall. I need to write something that can get everybody on the same page, then we

can really attack it. Here is some example code showing all of this using an iteration of (P):

```
# Example 1: The plaintext plane P_plane
#
import math;

# Get the dimensions of the plaintext
P_dims = int(math.ceil(math.sqrt(n)));
print("P_dims:%s" % (P_dims));

for i in range(n):
    # Get the plaintext point
    P_pt = (i % P_dims, int(i / P_dims));

    print("P[%s]:(%s, '%s') " % (i, P_pt, P[i]));
```

The output of the code above is:

```
P_dims:5
P[0]:((0, 0), 'T')
P[1]:((1, 0), 'h')
P[2]:((2, 0), 'i')
P[3]:((3, 0), 's')
P[4]:((4, 0), ' ')
P[5]:((0, 1), 'i')
P[6]:((1, 1), 's')
P[7]:((2, 1), ' ')
P[8]:((3, 1), 'p')
P[9]:((4, 1), 'l')
P[10]:((0, 2), 'a')
P[11]:((1, 2), 'i')
P[12]:((2, 2), 'n')
P[13]:((3, 2), 't')
P[14]:((4, 2), 'e')
P[15]:((0, 3), 'x')
P[16]:((1, 3), 't')
```

Okay, now we need to create the 2d complex plane (CP_plane) that we will map onto (P_plane). So, lets start off by creating some 2d axes (CP_axes) that describe the dimensions of (CP_plane) where (CP_axes[0]) and (CP_axes[1]) are the minimum and maximum values of the real axis. (CP_axes[2]) and (CP_axes[3]) are the minimum and maximum values of the imaginary axis. We will also create a 2d complex point (CP) in (CP_plane), that corresponds to a Julia set fractal. Lets go ahead and say that (CP_axes) and (CP) are part of the secret key (Secret_Key) of this cipher algorithm, where (Secret_Key[0]) is equal to (CP_axes) and (Secret_Key[1]) is equal to (CP). We also need to create (CPOW) that is the power to raise z by in the fractal function. We define (CITER) that tells us how many times we iterate the fractal. We define (CE) that defines the escape radius of the fractal iterations.

Lets create some code that shows this moment:

```
# Example 2: The Secret Key
#
CP_axes = (-2, 2, -2, 2);
CP = (-.75+.09j); # The seahorse valley
CPOW = 2
CITER = 103
CE = 2

Secret_Key = (CP_axes, CP, CPOW, CITER, CE);

print("CP_axes[0]:%s" % (CP_axes[0]));
print("CP_axes[1]:%s" % (CP_axes[1]));
print("CP_axes[2]:%s" % (CP_axes[2]));
print("CP_axes[3]:%s" % (CP_axes[3]));

print("CP:%s" % (CP));
print("CPOW:%s" % (CPOW));
print("CITER:%s" % (CITER));
print("CE:%s" % (CE));

print("Secret_Key[0]:%s" % (str(Secret_Key[0]));
print("Secret_Key[1]:%s" % (str(Secret_Key[1]));
print("Secret_Key[2]:%s" % (str(Secret_Key[2]));
print("Secret_Key[3]:%s" % (str(Secret_Key[3]));
print("Secret_Key[4]:%s" % (str(Secret_Key[4]));
```

The output of the code above is:

```
CP_axes[0]:-2
CP_axes[1]:2
CP_axes[2]:-2
CP_axes[3]:2
CP:(-0.75+0.09j)
CPOW:2
CITER:103
CE:2
Secret_Key[0]:(-2, 2, -2, 2)
Secret_Key[1]:(-0.75+0.09j)
Secret_Key[2]:2
Secret_Key[3]:103
Secret_Key[4]:2
```

So, lets assume the role of Alice and start off with creating some random axes (R_axes). These values are complete arbitrary for now. Lets assume that a random number is a double precision floating point in range of (0...1). Create a value (rmax) that represents the max width values of the axes and set it to 20. Create a value (rpad) that is the smallest width the axes can be and set it to .00001 plus a random number multiplied by .00001. Now, create a value (rxmin) that represents the minimum value of the real axis and set it to the negated value of (rmax) divided by 2, plus a random number multiplied by (rmax). Create a value (rymin) that represents the minimum value of the imaginary axis and set it to the negated value

of (rmax) divided by 2, plus a random number multiplied by (rmax). Lets create a value (rxmax) representing the maximum value of the real axis as set it to (rxmin) plus a random number multiplied by (rmax), plus (rpad). Now, we create a value (rymax) that is the maximum value of the imaginary axis as set it to (rymin) plus a random number multiplied by (rmax), plus (rpad). Finally, we set the 4 components of (R_axes) to (rxmin), (rxmax), (rymin) and (rymax) respectfully.

```
# Example 3: The Random Axes From Alice
```

```
#
```

```
import random;
```

```
rmax = 20;
```

```
rpad = .00001 + random.random() * 0.00001;
```

```
rxmin = -rmax / 2 + random.random() * rmax;
```

```
rymin = -rmax / 2 + random.random() * rmax;
```

```
rxmax = rxmin + random.random() * rmax + rpad;
```

```
rymax = rymin + random.random() * rmax + rpad;
```

```
R_axes = (rxmin, rxmax, rymin, rymax);
```

```
print("R_axes:%s" % (str(R_axes)));
```

The output I happened to get from the code above is:

```
R_axes: (-1.2870702298335752, -0.17871365701288908, -0.7914178819001128,
0.7932601328326381)
```

In order to get consistent values in the example code to come, we will assume this output value for (R_axes) will stay the same. In a real implementation, it should be different each time one runs the code.

Now, here is exactly how to get the final axes (CR_axes) that will be used for this session. Let us define |x| to mean the absolute value of x. Create a value (crsum) that is equal to the |(CP_axes[0])| plus |(CP_axes[1])| plus |(CP_axes[2])| plus |(CP_axes[3])| plus |(CP.real)| plus |(CP.imag)|. Create a value (crmod) that is equal to PI / 1.713. Create a value (crmul) that is equal to 471235630 multiplied by (crsum). Create a value (crxmin) that is equal to |(R_axes[0])| multiplied by, (CP_axes[0]) plus (CP.real), multiplied by (crmul)|, mod (crmod), using floating point modulus. If (R_axes[0]) is less than zero, then set (crxmin) equal to the negated value of (crxmin). Create a value (crymin) that is equal to |(R_axes[2])| multiplied by, (CP_axes[1]) plus (CP.imag), multiplied by (crmul)|, mod (crmod), using floating point modulus. If (R_axes[2]) is less than zero, then set (crymin) equal to the negated value of (crymin). Create a value (crxmax) that is equal to (crxmin) plus |(R_axes[1])| multiplied by, (CP_axes[2]) + (CP.real), multiplied by (crmul)|, mod (crmod), using floating point modulus. Create a value (crymax) that is equal to (crymin) plus |(R_axes[3])| multiplied by, (CP_axes[3]) + (CP.imag), multiplied by (crmul)|, mod (crmod), using floating point modulus.

I really do not have any rational explanation why I chose the constants in the verbatim description above. I am trying to implement a simple version of FFE on the fly, and document it. The goal is to end up on the same page.

Finally, create (CR_axes) whose components are equal to (crxmin), (crxmax), (crymin) and (crymax) respectfully. Here is some example code that shows how to do this:

```
*****  
  
# Example 4: The Combined Random Axes (CR_axes)  
#  
crsum = math.fabs(CP_axes[0]) + math.fabs(CP_axes[1]) + math.fabs(CP_axes[2]) +  
math.fabs(CP_axes[3]) + math.fabs(CP.real) + math.fabs(CP.imag);  
  
crmod = math.pi / 1.713;  
crmul = 471235630 * crsum;  
  
crxmin = math.fabs(R_axes[0] * (CP_axes[0] + CP.real) * crmul) % crmod;  
if (R_axes[0] < 0): crxmin = -crxmin;  
  
crymin = math.fabs(R_axes[2] * (CP_axes[1] + CP.imag) * crmul) % crmod;  
if (R_axes[2] < 0): crymin = -crymin;  
  
crxmax = crxmin + math.fabs(R_axes[1] * (CP_axes[2] + CP.real) * crmul) % crmod;  
crymax = crymin + math.fabs(R_axes[3] * (CP_axes[3] + CP.imag) * crmul) % crmod;  
  
CR_axes = (crxmin, crxmax, crymin, crymax);  
  
print("crsum:%s" % (crsum));  
print("crmod:%s" % (crmod));  
print("crmul:%s" % (crmul));  
print("CR_axes:%s" % (str(CR_axes)));
```

The output I get from the code above is:

```
crsum:8.84  
crmod:1.8339711929887874  
crmul:4165722969.2  
CR_axes:(-0.38943433546206707, 0.23697082287753957, -1.3762145808619817,  
0.3400477925872465)
```

```
*****  
  
Now these axes "seem" to be okay, because the imaginary axes is off center, and so  
are the real axes. In other words its off a center part that can exhibit some  
symmetrical behavior wrt the escape conditions of the fractal formula (F). This  
tutorial will show you exactly how to plot a rendering using (F) along with  
(CR_axes). Also, (F) will be a part of the (Secret_Key) in the form of a simple  
user-defined fractal function. The users of this raw implementation can create  
their own from scratch, or perhaps use a fractal program, such as Xaos, to find  
their own "beautiful" (F). FWIW, Xaos can be found here:
```

<http://matek.hu/xaos/doku.php>

Keep in mind, this tutorial will touch on using arbitrary precision in order to
increase the (Secret_Key) space to really __deep__ zooms wrt (CR_axes).

Now, we need to map the plaintext P to the complex plane defined by (CR_axes). Let
us define the width of (CR_axes) as its real axis maximum (CR_axes[1]) minus its

real axis minimum (CT_axes[0]). We shall do the same for the height of the imaginary axes at maximum (CR_axes[3]) minus its imaginary axis minimum (CT_axes[2]). Okay... Lets now define a value (xstep) that is the width divided by, (P_dims). (P_dims) is defined in example 1. Now, lets define a value (ystep) that is the height divided by, (P_dims). This is where we can map each byte in the plaintext to a complex number (c). So, lets iterate over (P) again, just like in example 1. Except, this time we will be generating a complex number (c). Taking (P_pt) from example 1 which is a 2d point where (P_pt[0]) is x, and (P_pt[1]) is y, in the square we put around (P), we can derive (c). The real component (c.real) is defined by taking (CR_axes[0]) plus (P_pt[0]), multiplied by (xstep). The imaginary component (c.imag) is defined by taking (CR_axes[3]) minus (P_pt[1]), multiplied by (ystep). Here is some code that shows all of this:

```
*****

# Example 5: Mapping plaintext P to complex numbers using (CR_axes)
#
xstep = (CR_axes[1] - CR_axes[0]) / P_dims;
ystep = (CR_axes[3] - CR_axes[2]) / P_dims;

for i in range(n):
    # Get the plaintext point
    P_pt = (i % P_dims, int(i / P_dims));

    # Get the complex point
    c = (CR_axes[0] + P_pt[0] * xstep, CR_axes[3] - P_pt[1] * ystep);

    print("P[%s]:(%s, '%s', %s)" % (i, P_pt, P[i], c));
```

The output I get from the code above is:

```
P[0]:((0, 0), 'T', (-0.38943433546206707, 0.3400477925872465))
P[1]:((1, 0), 'h', (-0.26415330379414576, 0.3400477925872465))
P[2]:((2, 0), 'i', (-0.13887227212622444, 0.3400477925872465))
P[3]:((3, 0), 's', (-0.013591240458303122, 0.3400477925872465))
P[4]:((4, 0), ' ', (0.1116897912096182, 0.3400477925872465))
P[5]:((0, 1), 'i', (-0.38943433546206707, -0.0032046821025991656))
P[6]:((1, 1), 's', (-0.26415330379414576, -0.0032046821025991656))
P[7]:((2, 1), ' ', (-0.13887227212622444, -0.0032046821025991656))
P[8]:((3, 1), 'p', (-0.013591240458303122, -0.0032046821025991656))
P[9]:((4, 1), 'l', (0.1116897912096182, -0.0032046821025991656))
P[10]:((0, 2), 'a', (-0.38943433546206707, -0.3464571567924448))
P[11]:((1, 2), 'i', (-0.26415330379414576, -0.3464571567924448))
P[12]:((2, 2), 'n', (-0.13887227212622444, -0.3464571567924448))
P[13]:((3, 2), 't', (-0.013591240458303122, -0.3464571567924448))
P[14]:((4, 2), 'e', (0.1116897912096182, -0.3464571567924448))
P[15]:((0, 3), 'x', (-0.38943433546206707, -0.6897096314822904))
P[16]:((1, 3), 't', (-0.26415330379414576, -0.6897096314822904))
```

Now we actually have some complex numbers mapped to the plaintext to go ahead and work with. The exercise of Funny Fractal Encryption is to try to exploit some of the entropy that arises when we expose the complex numbers in the example (5) above to a fractal function (F) in the (Secret_Key).

Okay, lets define (F) as a function. Now, this (F) can be generated with the aid of

a fractal explorer program of your choice, but lets assume for now that is defined as a function taking three parameters, (z, c, p). Where (z) equals an origin point, (c) equals a Julia set point, and (p) equals a power to raise (z) by. Here is such a function in Python:

Example 6: The fractal function (F)

#

```
def fractal(z, c, p):
```

```
    return z**p + c;
```

Now, let us define a function (nraise) that raises the absolute value of a number (z) as its only input, resulting from a fractal iteration of (c) mapped to (CR_axes) wrt (P). Set (a) equal to the absolute value of the functions only input (z). If (a) equals zero, then set (a) to (3.14159), totally arbitrary for now. Okay, now we loop on the condition, if (a) is less than (1000000). And for each time its true, we multiply (a) by (103). When the loop is completed, we take the floor of (a), mod 65536 in floating point modulus, convert it to an integer, and return the value to the caller. We mod with 65536 to bring the return value (sum) into a two byte range. Here is some example code that does this:

Example 7: The number raiser function (NR)

#

```
def nraise(z):
```

```
    a = math.fabs(z);
```

```
    if (a == 0): a = 3.14159;
```

```
    while (a < 1000000):
```

```
        a = a * 103;
```

```
    return (int)(math.floor(a) % 65536);
```

Now, we can finally get reap some cipherbytes. Lets define a function fractal_iteration that takes 5 parameters:

z = the origin point, a mapped complex number to plaintext

c = the Julia point

p = the power to raise z to

imax = the number of iterations

e = the escape radius

It iterates imax times, for each iteration it sets z to the return value of a call to the (fractal) function using z, c and p as parameters. It then checks to see if the absolute value of z, which is $\sqrt{z.\text{real}^2 + z.\text{imag}^2}$, is greater than e. If it is, it breaks out of the loop, if not, it continues on with the iteration. Now, when this loop is completed either through iterating n times, or the early break, we take an integer (rp) by assigning it to the return value of a call to (nraise) using the real part of z as its parameter. Now, we take an integer (ip) by assigning it the return value of a call to (nraise) using the imaginary part of z as its parameter. We create a variable (sum) and set it to (rp) multiplied by (ip), mod 100323. Finally, it returns (sum) mod 256 to the caller, in order to bring it

into a byte range. Here is some example code:

```
*****
```

```
# Example 8: The fractal iteration function (FI)
```

```
#
```

```
def fractal_iteration(z, c, p, imax, e):  
    for i in range(imax):  
        z = fractal(z, c, p);  
        if (math.sqrt(z.real**2+z.imag**2) > e):  
            break;  
    rp = nraise(z.real);  
    ip = nraise(z.imag);  
    sum = (rp * ip) % 100323;  
    return sum % 256;
```

```
*****
```

Okay. We can actually encrypt the damn plaintext using the (fractal_iteration) function above. The function is fed with each complex number in parameter z mapped to each plaintext byte. Its output cipherbyte is xor'ed with the plaintext byte. Lets define it...

A function called encrypt for Alice that takes 4 parameters:

```
jc = The Julia set point  
p = The power of the formula  
imax = The iteration count  
e = The escape radius for the fractal iteration.
```

This simple function returns a buffer cbytes that represents the plaintext bytes xor'ed with the cipherbytes returned from calls to fractal_iteration for each complex number mapped to each byte in P, refer to example 5 for further context. cbytes is set to a buffer with zero length. For each byte in P, with n bytes, we map the plaintext byte to integer point P_pt, refer to example 1. We create a point c that is a mapping from the current byte in P to its complex number via P_pt, refer to example 5 for further context. We create an integer ct and set it to the result of a call to the fractal_iteration function with parameters c, jc, p, imax, and e. Take the byte value (0...255) of the current byte in the plaintext and xor it with ct. Append this byte to the cbytes buffer. When we are done processing each byte in P, we return cbytes to the caller. Here is some example code:

```
*****
```

```
# Example 9: The encrypt function for Alice (CF)
```

```
#
```

```
def encrypt(jc, p, imax, e):  
    cbytes = "";  
  
    for i in range(n):  
        # Get the plaintext point  
        P_pt = (i % P_dims, int(i / P_dims));  
  
        # Get the complex point  
        c = complex(CR_axes[0] + P_pt[0] * xstep, CR_axes[3] - P_pt[1] * ystep);
```



```

    cb = fractal_iteration(c, jc, p, imax, e);

    ct = ord(P[i]) ^ cb;

    print("P[%s]:(%s, %s):%s:'%s':(%s xor %s):%s" %
          (i, P_pt[0], P_pt[1], c, [i], ord(P[i]), cb, ct));

    cbytes = cbytes + chr(ct);

return cbytes;

# The cipherbytes from Alice
cbytes = encrypt(Secret_Key[1], 2, 103, 2);

```

The output I get from the code above is:

```

P[0]:(0, 0):(-0.38943433546206707+0.3400477925872465j):'[0]':(84 xor 4):80
P[1]:(1, 0):(-0.26415330379414576+0.3400477925872465j):'[1]':(104 xor 81):57
P[2]:(2, 0):(-0.13887227212622444+0.3400477925872465j):'[2]':(105 xor 64):41
P[3]:(3, 0):(-0.013591240458303122+0.3400477925872465j):'[3]':(115 xor 117):6
P[4]:(4, 0):(0.1116897912096182+0.3400477925872465j):'[4]':(32 xor 73):105
P[5]:(0, 1):(-0.38943433546206707-0.0032046821025991656j):'[5]':(105 xor 178):219
P[6]:(1, 1):(-0.26415330379414576-0.0032046821025991656j):'[6]':(115 xor 182):197
P[7]:(2, 1):(-0.13887227212622444-0.0032046821025991656j):'[7]':(32 xor 204):236
P[8]:(3, 1):(-0.013591240458303122-0.0032046821025991656j):'[8]':(112 xor 124):12
P[9]:(4, 1):(0.1116897912096182-0.0032046821025991656j):'[9]':(108 xor 196):168
P[10]:(0, 2):(-0.38943433546206707-0.3464571567924448j):'[10]':(97 xor 59):90
P[11]:(1, 2):(-0.26415330379414576-0.3464571567924448j):'[11]':(105 xor 215):190
P[12]:(2, 2):(-0.13887227212622444-0.3464571567924448j):'[12]':(110 xor 136):230
P[13]:(3, 2):(-0.013591240458303122-0.3464571567924448j):'[13]':(116 xor 112):4
P[14]:(4, 2):(0.1116897912096182-0.3464571567924448j):'[14]':(101 xor 40):77
P[15]:(0, 3):(-0.38943433546206707-0.6897096314822904j):'[15]':(120 xor 205):181
P[16]:(1, 3):(-0.26415330379414576-0.6897096314822904j):'[16]':(116 xor 170):222

```

Alice has just encrypted the plaintext P!

R_axes is the random iv, and cbytes returned from encrypt are the bytes of the encrypted plaintext. Both of these, R_axes and cbytes, get sent to Bob.

Bob receives R_axes and cbytes from Alice... He uses Example 4 to create CR_axes for himself, and runs the cipher function, but he reverses the xor operation.

(more examples to come...)