vectorCrypt

> This set of functions is used to encrypt data. The actual encryption occurs with encrypt() and private_encrypt(). encrypt() uses a non reversible function discovered in linear algebra to create a unique vector that maps to a point in the dimension when multiplied by a square matrix of plaintext.
>
> For example, assume someone enters in a credit card number:
>
> Convert it into 4x4 matrix:
>
> 1 4 5 2
>
> 2 1 0 0
>
> 3 1 7 4
>
> 2 6 3 0
>
> Check if its independent, if not, make it independent:
>
> 1 4 5 2|0
>
> 2 1 0 0|0
>
> 3 1 7 4|0
>
> 2 6 3 0|0
>
> Set that matrix equal to a random point in R4 space:
>
> 1 4 5 2|-300912
>
> 2 1 0 0|8764
>
> 3 1 7 4|21
>
> 2 6 3 0|8449372
>
> There is a guaranteed unique linear combination of the vectors that generate the random point.

Grab the coefficients of the vectors, this is your new encrypted data:

-243 0 0 0|-300912	-243	-300912
0 2833 0 0|8764	2833	8764
0 0 2811 0|21	2811	21
0 0 0 3100|8449372	3100	8449372

Store the coefficients and the random point.

This information is now encrypted, and if a hacker gains access to both sets of data, there is no mathematical way to generate the original matrix. Thus, this is a one way encryption/hash.

To validate a password entry, encrypt the data using the previous random point.

If the encryptions match, the password is correct.

A more secure version of encryption is achieved when vectorCrypt is used in tandem with a private key.

Take a private key of 50 characters long:

{a, 6, 3, 8, 2, s, I, &, #, !, B, C........} size 50

Split that into 2 smaller keys:

Keya={a, 6, 3, 8.....} size 25

Keyb={S, 2, D, a.....} size 25

Convert each into a 5x5 matrix:

(These characters are stored as ascii values)

KeyA	KeyB
a 6 3 8 2	c 1 4 w a

```
2 s I & #            a B # ! 7

! B C q C            z B @ & s

d @ X d S            C S B A s

# s C 2 a            b 8 * % 2
```

Now add a unique combination of KeyA and KeyB to the original points:

```
1 x x x  +  (a + 6 + 3 + 8 + 2)+(c + a + z + C + b)  (Row 1 of KeyA + Column 1 of KeyB)

x x x x

x x x x

x x x x
```

Now multiply the new matrix points by the sum of the row and column:

(# is the result of the previous calculation)

```
# x x x  *  (a + 6 + 3 + 8 + 2)+(c + a + z + C + b)  (Row 1 of KeyA + Column 1 of KeyB)

x x x x

x x x x

x x x x
```

Do this for every point on the matrix, and you will have a new, privately encrypted matrix:
(Where each letter represents the previous calculations)

```
a b c d e

f g h i j

k l m n o

p q r s t

u v w x y
```

This is your new matrix, now you complete vectorCrypt on the matrix.

In terms of collision:

The operation, mathematically, generates a unique vector for every single point in Rn.

However, as the computer stores the values as a decimal, some entropy is lost and some

collision can occur on very close values. However, I have mathematically proven that the
radius of collision is directly proportional to the precision of storage the computer uses.
With the precision of a double, a collision is impossible.

In terms of security:

The highest processing GPU can generate 250 billion passwords per second, we will use this as our standard.
Assume a user entered a 16 digit credit card number.

Using vectorCrypt only:

It would take 46.296 days to generate the total password space

Using vectorCrypt with 50 digit (approximately 325 bit) Private Key:

It would take 2.88 x 10^92 years to generate every possible combination of credit card number and Key

Using vectorCrypt with 50 digit (approximately 325 bit) Private Key:

It would take 1.175 x 10^83 years to generate all possible combinations of the resultant matrix after the private key has been used
(Assuming the hacker bypasses the key and outputs all possible matrices that are a result of multiplying by a Private Key to the encryption function)

Impervious to collisions

Created in program and idea by Conor D'Arcy