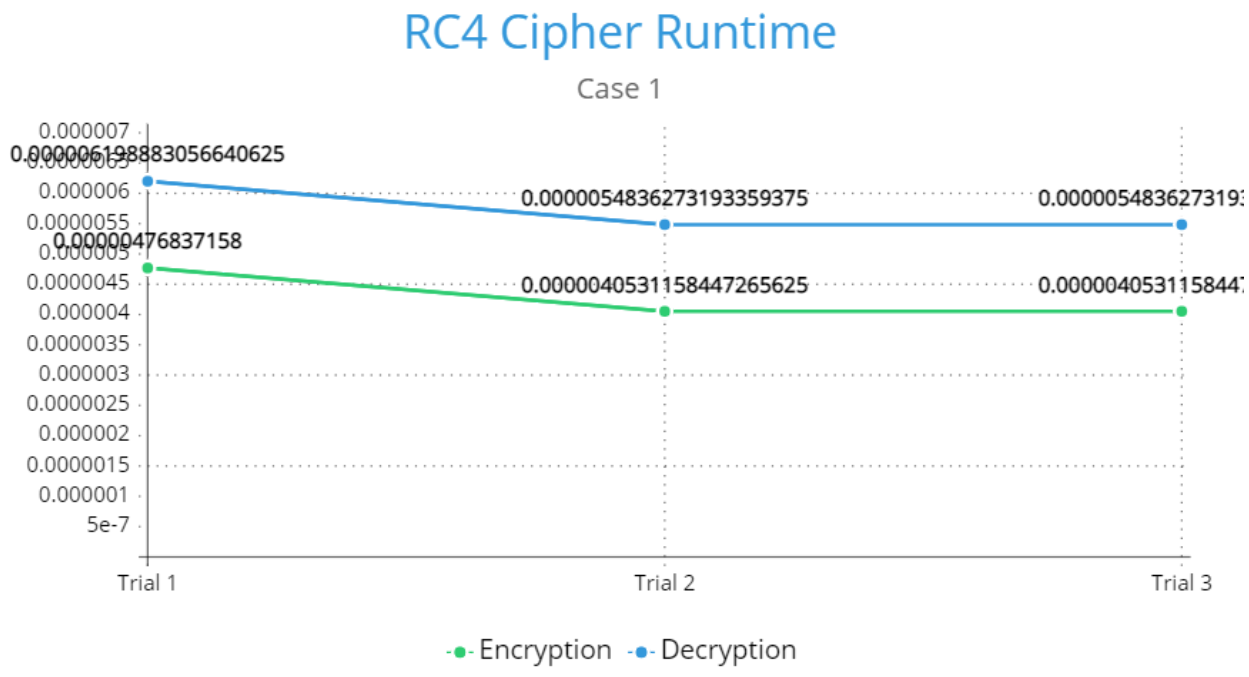
RC4

RC4 (Rivest Cipher 4 or ARC4) is a stream cipher designed in 1987 by Ron Rivest for RSA Security. The unique thing about it and why it's been used for a while ever since its debut is because it is simple to implement and use and the speed of it is impressive. However shortly after its debut, few vulnerabilities have been found which has rendered it obsolete and non-secure. As mentioned before RC4 is a stream cipher, so it is more susceptible than common block ciphers. If the message authentication code (MAC) is not strong enough, then encryption is vulnerable to a bit-flipping attack (alter the cipher text to interfere with the plain text) . The biggest reason why RC4 is not in use anymore is because unlike most stream ciphers which use nonce in conjunction with a key, RC4 doesn't create a "fresh" new key after each encryption which makes it easy for attackers to target. Since RC4 is an example of symmetric key encryption, all parties involved have to exchange the key used to encrypt the data before they can decrypt it. Another reason is because it is what is implemented in WEP which is surpassed by WPA and WPA-PSK making it obsolete. More drawbacks include one in every 256 keys can be a weak key, A particular RC4 Algorithm key can be used only once.

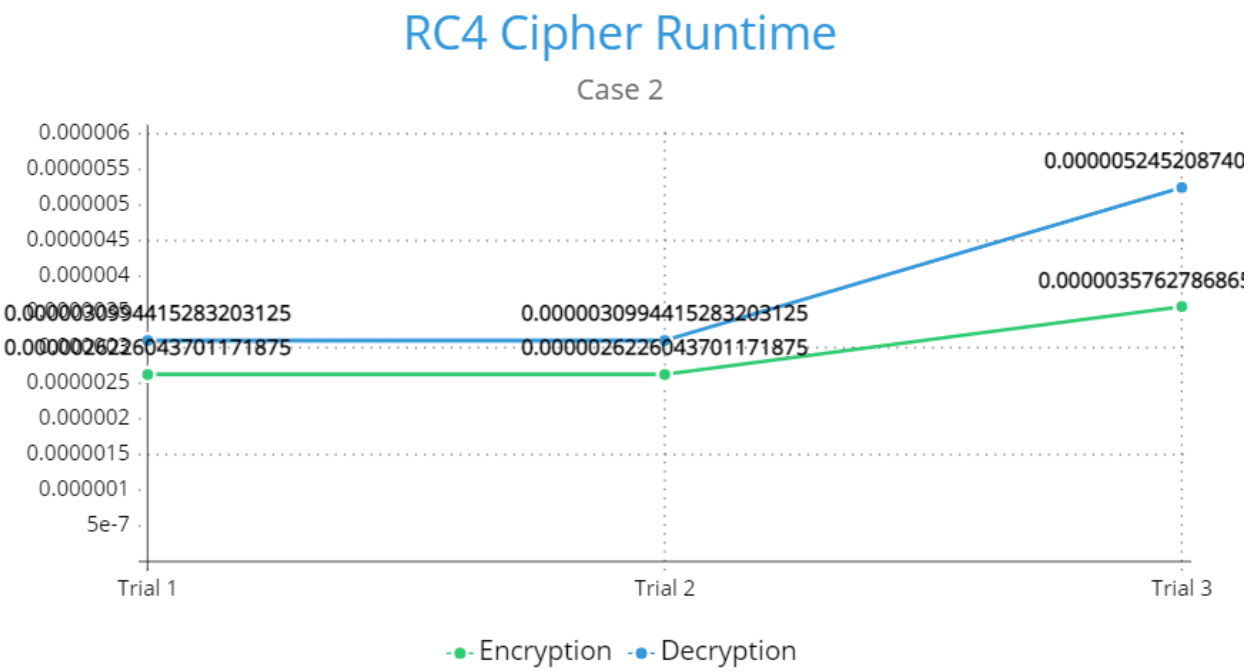
To start the encryption, we need a secret key and plaintext (for decryption we need a secret key and ciphertext). RC4 uses a key-stream (pseduo-random stream of bits) which is then encrypts the plaintext using bit-wise exclusive-or. Decryption is done using the same process since we the bit-wise exclusive-or is symmetric (can be reversed). The keystream is made up of two things: a permutation of all 256 bits which will be stored in an array called S and we use two 8-bit index pointers which are called i and j. The key can have a length between 40 bits and 256 bits (we implemented it using a mod function) and which is done using the key-scheduling algorithm (KSA). The output byte is selected by looking up the values of S(i) and S(j), adding them together modulo 256, and then looking up the sum in S; S(S(i) + S(j)) is used as a byte of the key-stream, K. First we use KSA and we put the key inside the function. We make sure the length of the key is at most 256 byte because we use a 256 bit key implementation (typically the key is around 5 bytes in length). We then create an array called S which is used as a swap method where we initialize two variables called i and j. Using these variables allows us to not need the input key and use stream generation to cycle through each element in S[i] and swap it with S[j]. Then we can use the XOR function with S[i] + S[j] % keylength to find the next byte of the plaintext (when encrypting) or we can do the same to find the next byte of the ciphertext (for decrypting). PRGA takes in the psuedo random state array (array S in KSA algorithm)

which outputs a stream key which is merged with the plaintext to create a stream of data using XOR bits that is encrypted and can be decrypted when trying to get the plaintext back.

```
Input your encryption key : hi
Input your plaintext : hello
This is your ciphertext : kdmkk
encryption time is:  4.768371582031250e-06 seconds
key generation time is:  4.291534423828125e-06 seconds

Welcome to my RC4 Algorithm
1. Encrypt
2. Decrypt
Please type in 1 or 2 if you want to encrypt or decrypt : 2

Input your encryption key : hi
Input your ciphertext : kdmkk
This is your plaintext : hello
decryption time is:  6.198883056640625e-06 seconds
key generation time is:  4.291534423828125e-06 seconds
```



```
Input your encryption key : yoloswagalicious
Input your plaintext : chupappimunyanyobby
This is your ciphertext : okipwaprvirsgqmgbhg
encryption time is:  2.6226043701171875e-06 seconds
key generation time is:  2.6226043701171875e-06 seconds

Welcome to my RC4 Algorithm
1. Encrypt
2. Decrypt
Please type in 1 or 2 if you want to encrypt or decrypt :

Input your encryption key : yoloswagalicious
Input your ciphertext : okipwaprvirsgqmgbhg
This is your plaintext : chupappimunyanyobby
decryption time is:  3.0994415283203125e-06 seconds
key generation time is:  2.6226043701171875e-06 seconds
```



We can see above the encryption time and decryption time is exactly the same (which is expected since the algorithm is symmetric), however on the second input our decryption time differs by a whole second. When running the program the time will vary but usually the encryption will have the same runtime as the decryption or it will at most have a difference of one second. Compared to the next cipher, the runtime is pretty consistent in the different

cases, however we see in at least one of the trials (thirty - three percent of the time given this model) there will be a difference of about one second with the decryption time will be one more second then the encryption time and both of the times for that one trial has a difference of about one second for both decryption and encryption as we can see in trial one for the first case and trial three in the second case. Other than those flaws, the program stays fairly accurate in giving similar runtimes.