Purple Stream Cipher

Karl Zander – pvial@kryptomagik.com, karl.zander00@gmail.com

Abstract.

Purple is an encryption algorithm that obfuscates data so that no one is able to read it without the key. It is small and fast, maintaining a 256-byte internal key stream state and utilizes an invertible output function in its powerful key stream generator. Purple can be used in pen and paper ciphering as well as in software.

1. Introduction

Purple is a fast encryption algorithm, originally designed to be a secure method of encrypting a message using only pen and paper, however, it has shown to be extremely fast, effective and secure in software. It is a stream cipher, operating on a single byte of data at a time.

Purple consists of a key setup function which produces a 256-byte key stream state, and a key stream generator which outputs pseudorandom bytes that are XORed with the input plaintext.

The cipher accepts key lengths ranging from 64 bits to 2048 bits. A nonce or initialization vector is required for encryption. Recommended nonce lengths are not to exceed 128 bits.

2. Design goals

In designing Purple, I wanted to get away from the RC4 construction of having to swap an element of 0-255 each round but still have a fast construction and have "no moving parts" became a requirement. I also wanted to improve on weaknesses in the Purple cipher and develop a cipher with no counter that is applied in mathematics to the cipher text.

Purple follows the following additional requirements:

The algorithm must be able to be done at a reduced state; for example, mod 26 for hand ciphering purposes as well as at a mod 256 full state to accommodate binary data.

The algorithm must employ a public (or potentially private) nonce or initialization vector that must not give the attacker more than 50% advantage in the key stream generation process.

The algorithm must produce a non-repeating stream (without period) of bytes with uniform characteristics.

The algorithm must pass known statistical testing for random number generators.

The algorithm must be extremely fast to implement by hand and in software.

3. Key Setup

Purle's key setup makes it difficult to invert the initial state of the key stream byte array and recover the key. It consists of 5 actions, loading the key, computing j, key obfuscation, loading the nonce, key obfuscation once more, key expansion (to 256 bytes) and key setup output function which ensure invertibility. All the while through this process, j is computed after each action.

To illustrate in more detail, the key stream array is initialized to the length of the key and set to zero. The key stream array is referred to as k[]. Each byte of the key stream is added mod 256 to the zero-byte array. During this process, j is computed which j is simply the sum of each progressive byte mod 256 plus a mod 256 counter. (Addition of the counter is to prevent cases when the key is all zero from producing weak cipher text.)

```
    for (c=0; c < keylen; c++) {
        k[c % keylen] = (k[c % keylen] + key[c % keylen]) % 256;
        j = (j + k[c % keylen] + c) % 256; }
```

j is an index the shows which key stream byte to operate on next.

The next step is to step the key stream array forward 256 times (this value is the same regardless of key length), which is simply to compute k[c] + k[j] each time afterward assigning the newly formed byte position as j. This process generates key stream bytes that are not the actual bytes of the key.

```
    for (c = 0; c < 256; c++) {
        k[j] = (k[c % keylen] + k[j]) & 0xff;
        j = k[j]; }
```

Next the nonce or iv is processed in the same way. The bytes are added mod 256 to the k[] array in succession and then the key stream is stepped forward 256 times one more time by computing j and adding it to the k[] elements.

```
     for (c = 0; c < noncelen; c++) {
        k[c] = (k[c % noncelen] + nonce[c]) & 0xff;
        j = (j + k[c % noncelen]) & 0xff; }
    for (c = 0; c < 256; c++) {
        k[j] = (k[c % keylen] + k[j]) & 0xff;
        j = k[j]; }
```

Then key stream array is expanded by the difference between 256 and the key length in bytes. Key stream array expansion is done by using the key stream generator fuction k[c] plus [k + 1] plus k[j] mod 256. j is incremented by k[c].

```
    for (c = 0; c < diff; c++) {
        k[c+keylen] = (k[c] + k[(c + 1) % diff] + j) & 0xff;
        j = (k[j] + k[c % diff]) & 0xff; }
```

Finally, the key stream bytes are added to each other and k[j] thus creating a difficult to invert key stream state. m is calculated by dividing 256 by 2 and is simply a marker of the middle of the array.

```
    for (c = 0; c < 256; c++) {
        k[c] = (k[c] + k[(c + m) & 0xff] + j) & 0xff;
        j = (j + k[c] + c) & 0xff; }
```

In the case that the user feeds a key of all zeros, the key becomes the counter values 0-255 and is transformed into a pseudorandom key stream state still capable of encrypting an input of all zeros and achieve maximum entropy.

4. Key Stream Generator

The key stream generator is the heart of the Purple algorithm. It produces a single pseudo randomly generated key byte each round with which to XOR with the plaintext byte. The key generator's output function is difficult to invert. Also, the key stream bytes cannot be recomputed easily by simply knowing the output byte.

The key stream array bytes are permuted one at a time or one per round but are operated on in pairs (k[c] + k[j]).

A counter is maintained in the key generator and operates modulo 256. The counter is denoted by the variable c. The counter simply changes the position within the array.

The key stream permutation formula first assigned the j index to the current value of the k[j] byte. Then it sums the present counter value of k[] and the present j index value of k[]. This can be best represented with the following equation:

```
j = k[j];
k[j] = (k[c] + k[j]) & 0xff;
```

The output function combines k[j] and a nested lookup of k[k[j]] to produce the output byte that will be XORed with the plaintext byte.

```
output = (k[k[j]] + k[j]) & 0xff;
```

Complete excerpt of the key stream generator from the source code:

```
for (int x = 0; x < datalen; x++) {
        j = k[j];
        k[j] = (k[c] + k[j]) & 0xff;
        output = (k[k[j]] + k[j]) & 0xff;
        data[x] = data[x] ^ output;
        c = (c + 1) & 0xff;
   }
```

5. Cryptanalysis

The output of Purple has not been able to be distinguished from a random output. Nothing can be learned from frequency analysis.

Under a known plaintext attack of n bytes, one may not invert the output function to regain the key state bytes. One may know n bytes of the plaintext and the remainder of the message will remain secure.

6. Statistical Properties

Purple was subjected to the Diehard battery of tests (dieharder). Overall, Purple passes the tests.

Tests were conducted on streams of 1 and 2 gigabytes of data with 100 different key, nonce pairs.

Purple was also subjected to NIST Statistical Test Suite battery of tests and passed.

7. Performance

Performance tuned implementations of Purple have shown speeds around 4500 megabit per second on some test systems.