ZanderFish2 Block Cipher

Karl Zander – pvial00@gmail.com

Abstract.

ZanderFish2 is an encryption algorithm that obfuscates blocks of data so that no one is able to read it without the key. It is a Feistel network and maintains a 128 bit block size, 128 bits of round key per round and has a fierce 8 bit substitution array.

1. Introduction

ZanderFish2 is a fast block encryption algorithm aimed at providing no attack vector other than brute force.

ZanderFish2 consists of two key schedulers, one for the round keys and one for the substitution boxes and a round function. It uses substitution and ARX operations to encrypt data and operates on blocks of 128 bits of data.

The cipher is designed to operate at key lengths from 256 bits to 1024 bits. An initialization vector is required for encryption. The length of the IV must be 128-bits to match the block size (could be be different depending on the mode in implementation).

2. Design goals

The algorithm must adhere to the strict avalanche effect and produce proper confusion and diffusion.

The algorithm must employ an initialization vector that does not effect the key 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 fast to implement in software.

3. Round Key Setup and S-Boxes

- Round Keys

The round key setup uses a modified version of the Amagus stream cipher's round function which is composed of ARX operations on a 1024 bit state. The function maintains a 64 bit output word that when run is updated with the XOR of all words in the state. The key is loaded into the state and update function is run 16 times, once for each round outputting a 64 bit key each time.

Round key generation function

The Amagus round function that generates keys utilizes ARX operations and is broken up in two phases.

1. Mixing phase

   - The mixing phase does one of a few operations on alternating words in the state. The first word is added to a word, the second word is XOR'ed with a word, the third word is XOR'ed is the XOR of the word and another rotated left so many bits. This pattern continues throughout the rest of the state.

2. Transposition/diffusion phase

   - In this phase the state columns are alternated through the same function.

Round key generation function in C code:

```
void *zander_F(struct zksa_state *state) {
   int r;
   for (r = 0; r < 10; r++) {
      state->r[0] += state->r[6];
      state->r[1] ^= state->r[15];
      state->r[2] = zander_rotl((state->r[2] ^ state->r[12]), 9);
      state->r[3] += state->r[9];
      state->r[4] ^= state->r[11];
      state->r[5] = zander_rotr((state->r[5] ^ state->r[10]), 6);
      state->r[6] += state->r[13];
      state->r[7] ^= state->r[8];
      state->r[8] = zander_rotl((state->r[8] ^ state->r[3]), 11);
      state->r[9] += state->r[1];
      state->r[10] ^= state->r[4];
      state->r[11] = zander_rotr((state->r[8] ^ state->r[7]), 7);
      state->r[12] += state->r[0];
      state->r[13] ^= state->r[2];
      state->r[14] = zander_rotl((state->r[14] ^ state->r[0]), 3);
      state->r[15] += state→r[5];

      state->r[15] += state->r[6];
      state->r[2] ^= state->r[15];
      state->r[14] = zander_rotl((state->r[14] ^ state->r[12]), 9);
      state->r[4] += state->r[9];
      state->r[13] ^= state->r[11];
      state->r[6] = zander_rotr((state->r[6] ^ state->r[10]), 6);
      state->r[12] += state->r[13];
      state->r[8] ^= state->r[8];
      state->r[11] = zander_rotl((state->r[11] ^ state->r[3]), 11);
      state->r[10] += state->r[1];
      state->r[1] ^= state->r[4];
      state->r[3] = zander_rotr((state->r[3] ^ state->r[7]), 7);
      state->r[5] += state->r[0];
      state->r[7] ^= state->r[2];
      state->r[9] = zander_rotl((state->r[9] ^ state->r[0]), 3);
```

```
        state->r[0] += state->r[5];
    }
    for (r = 0; r < 16; r++) {
        state->o ^= state->r[r];
    }
}
```

- S-Boxes

The S-Box generation uses an algorithm derived from the Purple stream cipher. The key is wrapped around an array with a range 0-255 values via the XOR operator. Then the Purple encryption algorithm is used to output pseudorandom bytes with which to swap with a counter.

The purple algorithm starts by establishing an index j = 0. The first action that is performed in the algorithm is by choosing a different index which is looked up in the 256 byte array k[] which is the array the key was wrapped around. Then the array k's jth value is modified by adding k[counter value] with k[j]. Next, the output byte is generated by adding a nested lookup of k[k[j]] and k[j] mod 256 which is then used as the swap value. Lastly, the S-box values at the counter value and the output value are swapped. This is done 256 times per 256 byte S-box.

Purple S-Box generation function in C code:

```
  for (s = 0; s < 8; s++) {
      for (i = 0; i < 256; i++) {
          j = k[j];
          k[j] = (k[c] + k[j]) & 0xff;
          o = (k[k[j]] + k[j]) & 0xff;
          temp = state->S[s][i];
          state->S[s][i] = state->S[s][o];
          state->S[s][o] = temp;
      }
  }
```

4. Round F function

ZanderFish2's Feistel construction requires some kind of F function for the round. A modified version of the BlowFish F function was chosen to perform the 8 bit substitutions.

The F function takes the right side as input and is XOR'ed with the left. The function begins by unpacking the 64 bit word and looks up each one in the S-Boxes according to a certain order, in the middle of the function the left half only is XOR'ed with the right half. Substitutions are made by the lookup of a byte and then being XOR'ed with with another byte. In a couple of places, two substitutions are added together to make a new substitution.

This can best be demonstrated by the following C code:

```
uint64_t zF(struct zander_state * state, uint64_t xr) {
    int v, x, y, z, a, b, c, d;
    v = (xr & 0xFF00000000000000) >> 56;
```

```
    x = (xr & 0x00FF000000000000) >> 48;
    y = (xr & 0x0000FF0000000000) >> 40;
    z = (xr & 0x000000FF00000000) >> 32;
    a = (xr & 0x00000000FF000000) >> 24;
    b = (xr & 0x0000000000FF0000) >> 16;
    c = (xr & 0x000000000000FF00) >> 8;
    d = (xr & 0x00000000000000FF);

    a = a ^ state->S[4][a];
    b = b ^ state->S[5][b];
    c = c ^ state->S[6][c];
    d = d ^ state->S[7][d];

    a = a ^ state->S[7][d] + state->S[6][a];
    b = b ^ state->S[6][c];
    c = c ^ state->S[5][b];
    d = d ^ state->S[4][a];

    a = a ^ v;
    b = b ^ x;
    c = c ^ y;
    d = d ^ z;

    v = v ^ state->S[0][v];
    x = x ^ state->S[1][x];
    y = y ^ state->S[2][y];
    z = z ^ state->S[3][z];

    v = v ^ state->S[1][z] + state->S[2][v];
    x = x ^ state->S[2][y];
    y = y ^ state->S[3][x];
    z = z ^ state->S[0][v];
    xr = ((uint64_t)v << 56) + ((uint64_t)x << 48) + ((uint64_t)y << 40) + ((uint64_t)z << 32) + ((uint64_t)a << 24) +
((uint64_t)b << 16) + ((uint64_t)c << 8) + d;
    return xr;
```

5. Round encryption/decryption

ZanderFish2 uses 16 rounds to obfuscate data. Each round a number of operations are performed on either the right or left side. There are 7 operations to an encryption round. Each round the right and the left are swapped.

1. The right half is XOR'ed with a 64 bit round key
2. The left half is XOR'ed with a 64 bit round key
3. The right half is run through F substitution function and the result is XOR'ed with the left half.
4. The left half is bitwise rotated to the left by 9 bits
5. The right half is added to the left half
6. The left half is added to the right half
7. The left and right are swapped.

6. Cryptanalysis

TBD

7. Statistical Properties

ZanderFish2 was subjected to the Diehard battery of tests (dieharder). Overall, ZanderFish2 passed the tests. Tests were conducted on streams of 1 gigabyte of data with 100 different key, nonce pairs. Amagus was also subjected to NIST Statistical Test Suite battery of tests and passed.