ZanderFish3 Block Cipher

Karl Zander – pvial00@gmail.com

Abstract.

ZanderFish3 is an encryption algorithm that obfuscates blocks of data so that no one is able to read it without the key. It is an ARX cipher utilizing only addition modulo 2^64th, bitwise rotation and the XOR operator. It maintains a 256 bit block size, 1280 bits of round key per round and supports key lengths from 128 bits to 1024 bits.

1. **Introduction**

ZanderFish3 is based on the concepts, techniques and primitives used in the ThreeFish/Spock and Amagus ciphers. It is a tweakable block cipher, consisting of a key scheduler, a two phase mixing function and 128 bits of tweak, not to be confused with meth. It uses only ARX operations to encrypt data and operates on blocks of 256 bits of data.

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

2. **Design goals**

ZanderFish3 must be faster than the ZanderFish2 algorithm.

The algorithm must not use lookup tables such as S-Boxes.

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**

Like ZanderFish2, 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. This differs from ZanderFish2's key scheduler which accumulates the overall XOR of all words processed. ZanderFish3's key scheduler produces a unique output of 64 bits each time it's run and is not related to the previous output.

The key scheduler works by loading the key into the state and update function is run 10 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 three 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.

3. Output phase

   - In the output phase, all words in the state are XOR'ed together creating a unique 64 bit word

Round key generation function in C code:

```
void *zander3_F(struct z3ksa_state *state) {
  int r;
  for (r = 0; r < 16; r++) {
    state->r[0] += state->r[6];
    state->r[1] ^= state->r[15];
    state->r[2] = zander3_rotl((state->r[2] ^ state->r[12]), 9);
    state->r[3] += state->r[9];
    state->r[4] ^= state->r[11];
    state->r[5] = zander3_rotr((state->r[5] ^ state->r[10]), 6);
    state->r[6] += state->r[13];
    state->r[7] ^= state->r[8];
    state->r[8] = zander3_rotl((state->r[8] ^ state->r[3]), 11);
    state->r[9] += state->r[1];
    state->r[10] ^= state->r[4];
    state->r[11] = zander3_rotr((state->r[8] ^ state->r[7]), 7);
    state->r[12] += state->r[0];
    state->r[13] ^= state->r[2];
    state->r[14] = zander3_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] = zander3_rotl((state->r[14] ^ state->r[12]), 9);
        state->r[4] += state->r[9];
        state->r[13] ^= state->r[11];
        state->r[6] = zander3_rotr((state->r[6] ^ state->r[10]), 6);
        state->r[12] += state->r[13];
        state->r[8] ^= state->r[8];
        state->r[11] = zander3_rotl((state->r[11] ^ state->r[3]), 11);
        state->r[10] += state->r[1];
        state->r[1] ^= state->r[4];
        state->r[3] = zander3_rotr((state->r[3] ^ state->r[7]), 7);
        state->r[5] += state->r[0];
        state->r[7] ^= state->r[2];
        state->r[9] = zander3_rotl((state->r[9] ^ state->r[0]), 3);
        state->r[0] += state->r[5];
    }
    state->o = 0;
    for (r = 0; r < 16; r++) {
        state->o ^= state->r[r];
    }
}
```

4. **ZanderFish3 Mixing Function**

The round mixing algorithm is based off the idea of the ThreeFish round function and Spock Cipher round function. The ThreeFish algorithm is used for confusion and the modified Spock algorithm for diffusion.

The tweak is applied only once per round. Please see the next section for tweak placement. The 128 bits of tweak provided with the cipher were generated pseudorandomly.

Confusion:

- A word is added to another word.
- A word is added to a key
- A word is bitwise rotated left so many positions and XOR'ed with another word

Diffusion:

1. Phase One (applied to the left half)
    - The word is rotated right so many positions
    - The word is added to another word
    - The word is XOR'ed with a key
2. Phase Two (applied to the right half)
    - The word is rotate left so many positions

- The word is XOR'ed with with another word

Finally, the diffusion key is added to the word

5. **Round encryption/decryption**

ZanderFish3 uses 52/56/72 and 80 rounds to obfuscate data for key lengths 128/256/512 and 1024 respectively. Each round a number of ARX operations are performed. 1280 bits of round key is applied each round. The round keys are broken up into 5 groups, the primary key groups, K[], K2[] K3[] and K4[] which are 256 bits each and the diffusion group D which is 256 bits. The round operations will now be listed for the forward encryption round. For this explanation the four 64 bit words operated on will be called A, B, C and D.

Encryption (Confusion Phase):

1. A round key from group K[] is added to D
2. A round key from group K[] and and D are added to B
3. A is rotated left 18 positions and XOR'ed with C
4. A round key from group K[] is added to C
5. A round key from group K[] and C are added to A
6. D is rotated left 26 positions and XOR'ed with B
7. D and tweak0 is added to B
8. C and a round key from group K2[] is added to A
9. C is rotated left 14 positions and XOR'ed with A
10. A round key from group K2[] is added to D
11. D is added to C
12. B is rotated left 16 positions and XOR'ed with D
13. A round key from group K2[] is added to A
14. A is added to B
15. D is rotated left 34 positions and XOR'ed with C
16. A round key from group K2[] is added to B
17. D is added to C
18. A is rotated left 28 positions and XOR'ed with D

Encryption (Diffusion Phase):

1. A is rotated right 46 positions

2. D is added to A
3. A is XOR'ed with a round key from group K4[]
4. B is rotated right 34 positions
5. C and tweak1 are added to B
6. B is XOR'ed with a round key from group K4[]
7. C is rotated left 4 positions
8. C is XOR'ed with D
9. D is rotated left 6 positions
10. D is XOR'ed with A
11. Round keys from group K3 are added to A, B, C and D

At the end of all rounds the diffusion keys are added to A, B, C and D. This does not constitute a round.

To decrypt, the operations are operations are simply done in reverse, subtracting where adding, left bitwise rotating where right and right where left.

The following C code demonstrates both the forward encryption algorithm and the backward decryption algorithm:

In code A, B, C and D are replaced with Xl, Xr, Xp and Xq.

Encryption:

```
uint64_t z3block_encrypt(struct zander3_state * state, uint64_t *xl, uint64_t *xr, uint64_t *xp, uint64_t *xq) {
    int i;
    uint64_t Xr, Xl, Xp, Xq, temp;

    Xl = *xl;
    Xr = *xr;
    Xp = *xp;
    Xq = *xq;

    for (i = 0; i < state->rounds; i++) {
/* Confusion */
        Xq += state->K[i][0];
        Xr += Xq + state->K[i][1];
        Xl = zander3_rotl(Xl, 18) ^ Xp;

        Xp += state->K[i][2];
        Xl += Xp + state->K[i][3];
        Xq = zander3_rotl(Xq, 26) ^ Xr;
        Xr += Xq + t0;
        Xl += Xp + state->K2[i][0];
        Xp = zander3_rotl(Xp, 14) ^ Xl;

        Xq += state->K2[i][1];
        Xp += Xq;
        Xr = zander3_rotl(Xr, 16) ^ Xq;

        Xl += state->K2[i][2];
        Xr += Xl;
        Xq = zander3_rotl(Xq, 34) ^ Xp;

        Xr += state->K2[i][3];
        Xp += Xq;
        Xl = zander3_rotl(Xl, 28) ^ Xq;

/* Diffusion */

        Xl = zander3_rotr(Xl, 46);
        Xl += Xq;
        Xl ^= state->K4[i][0];

        Xr = zander3_rotr(Xr, 34);
        Xr += Xp + t1;
        Xr ^= state->K4[i][1];

        Xp = zander3_rotl(Xp, 4);
        Xp ^= Xr;

        Xq = zander3_rotl(Xq, 6);
        Xq ^= Xl;

        Xl += state->K3[i][2];
        Xr += state->K3[i][3];
        Xp += state->K3[i][1];
        Xq += state->K3[i][0];

    }
    *xl = Xl + state->D[3];
    *xr = Xr + state->D[2];
    *xp = Xp + state->D[1];
    *xq = Xq + state->D[0];

}
```

Decryption:

```
uint64_t z3block_decrypt(struct zander3_state * state, uint64_t *xl, uint64_t *xr, uint64_t *xp, uint64_t *xq) {
    int i;
    uint64_t Xr, Xl, Xp, Xq, temp;

    Xl = *xl;
    Xr = *xr;
    Xp = *xp;
    Xq = *xq;
    Xl -= state->D[3];
    Xr -= state->D[2];
    Xp -= state->D[1];
    Xq -= state->D[0];

    for (i = (state->rounds - 1); i != -1; i--) {
/* Diffusion */

        Xq -= state->K3[i][0];
        Xp -= state->K3[i][1];
        Xr -= state->K3[i][3];
        Xl -= state->K3[i][2];

        Xq ^= Xl;
        Xq = zander3_rotr(Xq, 6);

        Xp ^= Xr;
        Xp = zander3_rotr(Xp, 4);

        Xr ^= state->K4[i][1];
        Xr -= Xp + t1;
        Xr = zander3_rotl(Xr, 34);

        Xl ^= state->K4[i][0];
        Xl -= Xq;
        Xl = zander3_rotl(Xl, 46);

/* Confusion */

        temp = Xl ^ Xq;
        Xl = zander3_rotr(temp, 28);
        Xp -= Xq;
        Xr -= state->K2[i][3];

        temp = Xq ^ Xp;
        Xq = zander3_rotr(temp, 34);
        Xr -= Xl;
        Xl -= state->K2[i][2];

        temp = Xr ^ Xq;
        Xr = zander3_rotr(temp, 16);
        Xp -= Xq;
        Xq -= state->K2[i][1];

        temp = Xp ^ Xl;
        Xp = zander3_rotr(temp, 14);
        Xl -= Xp + state->K2[i][0];
        Xr -= Xq + t0;


        temp = Xq ^ Xr;
        Xq = zander3_rotr(temp, 26);
        Xl -= Xp + state->K[i][3];
        Xp -= state->K[i][2];
```

6. **Cryptanalysis**

TBD

7. **Statistical Properties**

ZanderFish3 was subjected to the Diehard battery of tests (dieharder). Overall, ZanderFish3 passed the tests. Tests were conducted on streams of 1 gigabyte of data with 100 different key, nonce pairs. ZanderFish3 was also subjected to NIST Statistical Test Suite battery of tests and passed.