Kuzynki Block Cipher (AKMS)

Karl Zander – uvajda@protonmail.com

Abstract.

Kuzynki is a block cipher designed to conceal the input data so that no one is able to read it without knowledge of the key. Kuzynki is a substitution-permutation network using substitution lookup, modular addition/substraction/multiplication modulo 256 and binary XOR operations. It makes use of a 128 bit block size, applying 128 bits of key material each round after each permutation.

1. **Introduction**

The Kuzynki cipher is based on the 4 core Rijndael functions with the intent on making a secure but different cipher than Rijdael. Most ciphers based on Rijndael inherit its S-Box and GF2(8) algebraic formula. Kuzynki, however, does not. The idea behind the cipher was to demonstrate a different construction that utilizes more confusion than standard Rijndael to create a totally new secure cipher based on some of the principles of Rijndael cipher.

2. **Design goals**

Kuzynki must implement a more confusing S-Box than Rijndael.

The algorithm must make use of multiplicative properties modulo 256.

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

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**

The Kuzynki key scheduler was designed with opposite thinking to the design of the Rijndael key scheduler. A good key scheduler in my opinion is a complex construction that leaves no correlation between round keys. In other words, knowing the key to round 14 does not allow one to compute another round key nor does it give any clues to the construction of any other round keys.

The key scheduler is a non-invertible function. It has a 512 bit state to be able to support key lengths of up to 512 bits. The key scheduler operates by first loading the key scheduler constants and the key into the state. Before the key scheduler state is modified, it is saved to a temporary array. Then, each 32 bit element of the state is either added modulo 2^32 to another element of the state or is rotated left so many bits after being XOR'ing itself with another state element. After each element has been subject to an ARX operation. The state is added modulo 2^32 with the saved temporary array of itself, this makes it non-invertible. The final operation is to XOR all state elements into an output 32 bit word. Each 32 bit word for each round is calculated one after the other starting from the first round and ending with the last.

4. **Kuzynki Functions**

Kuzynki inherits mostly by name the 4 Rijndael functions listed in order (SubBytes, ShiftRows, MixColumns, AddRoundKey). These functions are applied in the same order as Rijndael, however, each function with the exception of AddRoundKey is different in construction. The following 4 sections of this document describe each function in detail.

5. **SubBytes (The Kuzynki S-Box)**

The Kuzynki S-Box is a hybrid construction of a singular S-Box and a singular A-Box (Affine). It is designed to be complex, chaotic and confusing. The S-Box itself is an array composed of 256 randomly assorted elements. There was no specific specification for the order or design of the S-Box the only requirement is that it must be randomly organized by some truly random algorithm. The S-Box and it's inverse is listed below.

Forward S-Box:

uint8_t akmsS0[256] = {117, 12, 48, 75, 111, 232, 123, 193, 125, 231, 121, 91, 122, 14, 11, 92, 173, 80, 54, 208, 31, 160, 0, 150, 159, 58, 76, 50, 105, 236, 114, 156, 33, 104, 5, 222, 53, 42, 99, 81, 87, 201, 132, 71, 139, 25, 13, 176, 59, 49, 148, 190, 182, 130, 29, 65, 84, 16, 223, 89, 118, 106, 157, 209, 110, 129, 175, 41, 171, 181, 131, 113, 141, 170, 4, 144, 38, 206, 179, 248, 128, 167, 6, 228, 188, 90, 96, 102, 73, 51, 15, 57, 154, 213, 230, 74, 169, 93, 36, 78, 2, 149, 189, 7, 155, 239, 227, 77, 82, 136, 9, 178, 112, 79, 17, 243, 37, 23, 172, 22, 28, 244, 229, 225, 238, 192, 191, 34, 134, 207, 161, 198, 164, 196, 220, 86, 221, 18, 3, 142, 35, 233, 215, 94, 137, 250, 19, 62, 98, 46, 68, 100, 85, 241, 27, 202, 138, 10, 107, 56, 205, 61, 21, 211, 101, 168, 64, 200, 30, 67, 165, 174, 195, 251, 163, 103, 45, 226, 135, 8, 83, 133, 185, 180, 219, 246, 115, 24, 210, 72, 187, 253, 242, 124, 204, 88, 255, 166, 109, 197, 151, 70, 63, 44, 60, 235, 247, 186, 143, 126, 32, 162, 194, 66, 95, 146, 214, 147, 97, 40, 152, 177, 216, 140, 153, 224, 26, 237, 127, 43, 1, 183, 234, 249, 158, 108, 217, 116, 245, 52, 212, 199, 119, 55, 20, 39, 240, 47, 252, 69, 203, 218, 184, 254, 145, 120};

Inverse S-Box:

uint8_t akmsSI0[256] = {22, 230, 100, 138, 74, 34, 82, 103, 179, 110, 157, 14, 1, 46, 13, 90, 57, 114, 137, 146, 244, 162, 119, 117, 187, 45, 226, 154, 120, 54, 168, 20, 210, 32, 127, 140, 98, 116, 76, 245, 219, 67, 37, 229, 203, 176, 149, 247, 2, 49, 27, 89, 239, 36, 18, 243, 159, 91, 25, 48, 204, 161, 147, 202, 166, 55, 213, 169, 150, 249, 201, 43, 189, 88, 95, 3, 26, 107, 99, 113, 17, 39, 108, 180, 56, 152, 135, 40, 195, 59, 85, 11, 15, 97, 143, 214, 86, 218, 148, 38, 151, 164, 87, 175, 33, 28, 61, 158, 235, 198, 64, 4, 112, 71, 30, 186, 237, 0, 60, 242, 255, 10, 12, 6, 193, 8, 209, 228, 80, 65, 53, 70, 42, 181, 128, 178, 109, 144, 156, 44, 223, 72, 139, 208, 75, 254, 215, 217, 50, 101, 23, 200, 220, 224, 92, 104, 31, 62, 234, 24, 21, 130, 211, 174, 132, 170, 197, 81, 165, 96, 73, 68, 118, 16, 171, 66, 47, 221, 111, 78, 183, 69, 52, 231, 252, 182, 207, 190, 84, 102, 51, 126, 125, 7, 212, 172, 133, 199, 131, 241, 167, 41, 155, 250, 194, 160, 77, 129, 19, 63, 188, 163, 240, 93, 216, 142, 222, 236, 251, 184, 134, 136, 35, 58, 225, 123, 177, 106, 83, 122, 94, 9, 5, 141, 232, 205, 29, 227, 124, 105, 246, 153, 192, 115, 121, 238, 185, 206, 79, 233, 145, 173, 248, 191, 253, 196};

The mathematical S-Box function combines both the S-Boxes and A-Boxes or Affine Boxes to provide a more confusing approach to simple byte substitution. The formula substitutes the first byte of the array and multiplies it by the lookup of the byte in the same column 3 rows in the A-Box. The result is XOR'ed with the byte to the right (wrapping around when it's the last byte in the row) of it and that result is finally looked up in the S-Box table. An equation that illustrates this is S[((S[x] * A[x – 1]) ^ x + 1)] where x signifies the input vector.

To decrypt, one simply applies the equation in reverse SI[((SI[x] ^ x + 1) * AI[x – 1])]

Each S-Box function is data dependent on exactly 2 other elements in the array.

Forward Affine Box:

uint8_t akmsA0[256] = {75, 173, 9, 29, 71, 117, 69, 83, 31, 97, 55, 95, 161, 81, 129, 233, 105, 239, 83, 65, 81, 63, 103, 211, 1, 29, 251, 171, 49, 191, 27, 57, 15, 203, 181, 213, 243, 61, 17, 73, 147, 237, 203, 201, 253, 159, 169, 35, 61, 39, 133, 107, 165, 77, 187, 13, 155, 63, 3, 223, 119, 185, 147, 89, 149, 19, 153, 49, 51, 53, 21, 91, 217, 27, 225, 169, 43, 45, 175, 171, 5, 121, 155, 117, 101, 5, 45, 75, 251, 41, 207, 233, 219, 127, 221, 59, 229, 137, 17, 235, 211, 231, 127, 99, 73, 21, 11, 195, 77, 199, 59, 167, 109, 189, 183, 97, 51, 153, 139, 55, 247, 245, 67, 151, 53, 107, 41, 213, 237, 65, 85, 15, 11, 13, 47, 115, 215, 205, 131, 245, 227, 199, 183, 123, 135, 225, 113, 207, 185, 173, 43, 7, 101, 69, 181, 123, 125, 209, 129, 9, 119, 241, 157, 79, 141, 145, 255, 33, 99, 231, 179, 161, 193, 103, 141, 93, 223, 229, 19, 151, 221, 79, 25, 235, 87, 121, 111, 105, 189, 137, 159, 91, 115, 67, 111, 217, 143, 37, 163, 177, 125, 253, 87, 143, 37, 243, 139, 247, 23, 1, 113, 249, 175, 93, 255, 85, 209, 191, 201, 31, 177, 179, 23, 71, 219, 145, 249, 149, 133, 157, 57, 7, 25, 3, 131, 187, 205, 35, 109, 241, 39, 89, 239, 215, 47, 165, 167, 95, 193, 197, 197, 163, 195, 135, 33, 227};

Inverse Affine Box:

```
uint8_t akmsAI0[256] = {99, 37, 57, 53, 119, 221, 141, 219, 223, 161, 135, 159,
97, 177, 129, 89, 217, 15, 219, 193, 177, 191, 87, 91, 1, 53, 51, 3, 209, 63,
19, 9, 239, 227, 157, 125, 59, 21, 241, 249, 155, 229, 227, 121, 85, 95, 153,
139, 21, 151, 77, 67, 45, 133, 115, 197, 147, 191, 171, 31, 71, 137, 155, 233,
189, 27, 169, 209, 251, 29, 61, 211, 105, 19, 33, 153, 131, 165, 79, 3, 205,
201, 147, 221, 109, 205, 165, 99, 51, 25, 47, 89, 83, 127, 117, 243, 237, 185,
241, 195, 91, 215, 127, 75, 249, 61, 163, 235, 133, 247, 243, 23, 101, 149, 7,
161, 251, 169, 35, 135, 199, 93, 107, 39, 29, 67, 25, 125, 229, 193, 253, 239,
163, 197, 207, 187, 231, 5, 43, 93, 203, 247, 7, 179, 55, 33, 145, 47, 137, 37,
131, 183, 109, 141, 157, 179, 213, 49, 129, 57, 71, 17, 181, 175, 69, 113, 255,
225, 75, 215, 123, 97, 65, 87, 69, 245, 31, 237, 27, 39, 117, 175, 41, 195,
103, 201, 143, 217, 149, 185, 95, 211, 187, 107, 143, 105, 111, 173, 11, 81,
213, 85, 103, 111, 173, 59, 35, 199, 167, 1, 145, 73, 79, 245, 255, 253, 49,
63, 121, 223, 81, 123, 167, 119, 83, 113, 73, 189, 77, 181, 9, 183, 41, 171,
43, 115, 5, 139, 101, 17, 151, 233, 15, 231, 207, 45, 23, 159, 65, 13, 13, 11,
235, 55, 225, 203};
```

6. **ShiftRows**

The name ShiftRows may not entirely describe this function as it shifts rows to adjacent columns. It could be in fact called ShiftColumns. Following the same principle in Rijndael that one of the column shifts must be zero. The ShiftRows shifts each row to an adjacent column designated by the shift value for that row. {0, 1, 2, 3} are the standard shift values.

7. **MixColumns**

MixColumns is similar to the S-Box transformation using the A-Boxes. Each operation on 1 byte of the state array combines 4 bytes together in an algebraic structure. A byte is first multiplied by the A-Box table lookup of the array

element to the left of it (wrapping around if at the beginning of a row) then added to the array element in the next column right 1 place, the modulus is taken modulo 256. The result is XOR'ed with the element to the right of the input. An equation representing this is: ((x * A[x – 1]) + (col+1) mod 256) ^ x + 1

The inverse for decryption utilizes the precomputed multiplicative inverse table AI. To decrypt, one simply applies each step in reverse order applying inverse operations when necessary. (((x ^ x + 1) – (col+1)) * AI[x – 1]) mod 256

8. **AddRoundKey**

The AddRoundKey function simply XOR's the state array with a 128 bit round key each round.

9. **Operating Parameters**

Key Length	Rounds
128 bit	10
256 bit	14
512 bit	20

The logic behind selecting rounds for each key size is largely based on the best attacks against Rijndael at those key lengths. In general, I agree with many cryptographers that Rijndael has too few rounds. However, instead of iterating more rounds, I compensate with a more complex S-Box construction. I also give a large bound between 256 and 512 bit.

10. **Advanced KryptoMagick Standard**

DarkCastle is a file encryption program developed by KryptoMagick and since the introduction of the ZanderFish3 cipher, it's been the recommended cipher to use. Having learned much since then regarding cipher design, I have selected the Kuzynki cipher to be the recommended cipher in DarkCastle and also the "Advanced KryptoMagick Standard".

11. **Cryptanalysis**

To be published soon

12. **Statistical Properties**

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