# Kasumi Block Cipher: Design, Implementation, and Analysis

K ARYA SEKHAR DAS

March 8, 2025

## 1 Introduction

Kasumi is a block cipher designed for the Universal Mobile Telecommunications System (UMTS) and standardized by the 3GPP in 1999. It is based on MISTY1 but optimized for hardware efficiency and performance in mobile environments. It serves as the core of the confidentiality algorithm (f8) and integrity algorithm (f9) in mobile communication.

## 2 Cipher Overview

### 2.1 Block and Key Size

Kasumi operates on:

- **Block Size**: 64 bits (8 bytes)
- **Key Size**: 128 bits (16 bytes)
- **Structure**: 8-round Feistel network

### 2.2 Feistel Structure

Kasumi follows an 8-round Feistel structure, splitting the 64-bit plaintext into two 32-bit halves. Each round applies different transformations:

- **Odd Rounds**: FL (Function Linear) followed by FO (Function Outer).
- **Even Rounds**: FO followed by FL.

This alternating approach enhances diffusion and security.

## 3 Key Components

### 3.1 FL (Function Linear)

The FL function applies linear mixing:

$$R' = R \oplus \text{ROL}(L \wedge KL_{i1}, 1)$$
$$L' = L \oplus \text{ROL}(R' \wedge KL_{i2}, 1)$$

where $\oplus$ denotes XOR, $\wedge$ denotes AND, and ROL is a left rotation.

### 3.2 FO (Function Outer)

FO introduces non-linearity using:

$$R_j = FI(L_{j-1} \oplus KO_{ij}, KI_{ij}) \oplus R_{j-1}$$
$$L_j = R_{j-1}$$

It acts as a mini-Feistel network with three rounds.

### 3.3 FI (Function Inner)

FI is a substitution-permutation function with S-boxes:

- $S9$: 9-bit to 9-bit substitution.
- $S7$: 7-bit to 7-bit substitution.

It ensures strong confusion properties.

### 3.4 Key Schedule

Kasumi derives subkeys from a 128-bit master key:

$$KL_{i1} = \text{ROL}(K_i, 0) \oplus C_i$$
$$KL_{i2} = \text{ROL}(K_{i+2 \mod 8}, 5)$$
$$KO_{i1} = \text{ROL}(K_{i+1 \mod 8}, 8)$$
$$KO_{i2} = \text{ROL}(K_{i+5 \mod 8}, 15)$$
$$KO_{i3} = \text{ROL}(K_{i+3 \mod 8}, 13)$$
$$KI_{i1} = K_{i+7 \mod 8}$$
$$KI_{i2} = K_{i+4 \mod 8}$$
$$KI_{i3} = K_{i+6 \mod 8}$$

These subkeys enhance security against related-key attacks.

## 4 Encryption Algorithm

Kasumi encrypts a 64-bit plaintext using a 128-bit key:

1. Split plaintext into $L_0$ and $R_0$.
2. Generate subkeys from the master key.
3. Apply 8 rounds:
   - If odd: $L_i = R_{i-1} \oplus FO(FL(L_{i-1}, KL_i), KO_i, KI_i)$
   - If even: $L_i = R_{i-1} \oplus FL(FO(L_{i-1}, KO_i, KI_i), KL_i)$
4. Ciphertext is $L_8 || R_8$.

Decryption follows the same process with subkeys in reverse order.

# 5 Python Implementation

```
S7 = [
    54, 50, 62, 56, 22, 34, 94, 96, 38, 6, 63, 93, 2, 18, 123, 33,
    55, 113, 39, 114, 21, 67, 65, 12, 47, 73, 46, 27, 25, 111, 124, 81,
    53, 9, 121, 79, 52, 60, 58, 48, 101, 127, 40, 120, 104, 70, 71, 43,
    20, 122, 72, 61, 23, 109, 13, 100, 77, 1, 16, 7, 82, 10, 105, 98,
    117, 116, 76, 11, 89, 106, 0, 125, 118, 99, 86, 69, 30, 57, 126, 87,
    112, 51, 17, 5, 95, 14, 90, 84, 91, 8, 35, 103, 32, 97, 28, 66,
    102, 31, 26, 45, 75, 4, 85, 92, 37, 74, 80, 49, 68, 29, 115, 44,
    64, 107, 108, 24, 110, 83, 36, 78, 42, 19, 15, 41, 88, 119, 59, 3
]

S9 = [
    167, 239, 161, 379, 391, 334, 9, 338, 38, 226, 48, 358, 452, 385, 90, 397,
    183, 253, 147, 331, 415, 340, 51, 362, 306, 500, 262, 82, 216, 159, 356, 177,
    175, 241, 489, 37, 206, 17, 0, 333, 44, 254, 378, 58, 143, 220, 81, 400,
    95, 3, 315, 245, 54, 235, 218, 405, 472, 264, 172, 494, 371, 290, 399, 76,
    165, 197, 395, 121, 257, 480, 423, 212, 240, 28, 462, 176, 406, 507, 288, 223,
    501, 407, 249, 265, 89, 186, 221, 428, 164, 74, 440, 196, 458, 421, 350, 163,
    232, 158, 134, 354, 13, 250, 491, 142, 191, 69, 193, 425, 152, 227, 366, 135,
    344, 300, 276, 242, 437, 320, 113, 278, 11, 243, 87, 317, 36, 93, 496, 27,
    487, 446, 482, 41, 68, 156, 457, 131, 326, 403, 339, 20, 39, 115, 442, 124,
    475, 384, 508, 53, 112, 170, 479, 151, 126, 169, 73, 268, 279, 321, 168, 364,
    363, 292, 46, 499, 393, 327, 324, 24, 456, 267, 157, 460, 488, 426, 309, 229,
    439, 506, 208, 271, 349, 401, 434, 236, 16, 209, 359, 52, 56, 120, 199, 277,
    465, 416, 252, 287, 246, 6, 83, 305, 420, 345, 153, 502, 65, 61, 244, 282,
    173, 222, 418, 67, 386, 368, 261, 101, 476, 291, 195, 430, 49, 79, 166, 330,
    280, 383, 373, 128, 382, 408, 155, 495, 367, 388, 274, 107, 459, 417, 62, 454,
    132, 225, 203, 316, 234, 14, 301, 91, 503, 286, 424, 211, 347, 307, 140, 374,
    35, 103, 125, 427, 19, 214, 453, 146, 498, 314, 444, 230, 256, 329, 198, 285,
    50, 116, 78, 410, 10, 205, 510, 171, 231, 45, 139, 467, 29, 86, 505, 32,
    72, 26, 342, 150, 313, 490, 431, 238, 411, 325, 149, 473, 40, 119, 174, 355,
    185, 233, 389, 71, 448, 273, 372, 55, 110, 178, 322, 12, 469, 392, 369, 190,
    1, 109, 375, 137, 181, 88, 75, 308, 260, 484, 98, 272, 370, 275, 412, 111,
    336, 318, 4, 504, 492, 259, 304, 77, 337, 435, 21, 357, 303, 332, 483, 18,
    47, 85, 25, 497, 474, 289, 100, 269, 296, 478, 270, 106, 31, 104, 433, 84,
    414, 486, 394, 96, 99, 154, 511, 148, 413, 361, 409, 255, 162, 215, 302, 201,
    266, 351, 343, 144, 441, 365, 108, 298, 251, 34, 182, 509, 138, 210, 335, 133,
    311, 352, 328, 141, 396, 346, 123, 319, 450, 281, 429, 228, 443, 481, 92, 404,
    485, 422, 248, 297, 23, 213, 130, 466, 22, 217, 283, 70, 294, 360, 419, 127,
    312, 377, 7, 468, 194, 2, 117, 295, 463, 258, 224, 447, 247, 187, 80, 398,
    284, 353, 105, 390, 299, 471, 470, 184, 57, 200, 348, 63, 204, 188, 33, 451,
    97, 30, 310, 219, 94, 160, 129, 493, 64, 179, 263, 102, 189, 207, 114, 402,
    438, 477, 387, 122, 192, 42, 381, 5, 145, 118, 180, 449, 293, 323, 136, 380,
    43, 66, 60, 455, 341, 445, 202, 432, 8, 237, 15, 376, 436, 464, 59, 461
]
def _bitlen(x):
    assert x >= 0, "x must be non-negative"
    return len(bin(x)) - 2

def _mod(x):
    return ((x - 1) % 8) + 1

def _shift(x, n):
```

```
    return ((x << n) | (x >> (16 - n))) & 0xFFFF

class Kasumi:
    def __init__(self):
        self.key_KL1 = [0] * 9
        self.key_KL2 = [0] * 9
        self.key_KO1 = [0] * 9
        self.key_KO2 = [0] * 9
        self.key_KO3 = [0] * 9
        self.key_KI1 = [0] * 9
        self.key_KI2 = [0] * 9
        self.key_KI3 = [0] * 9

    def set_key(self, master_key):
        assert _bitlen(master_key) <= 128

        key = [0] * 9
        key_prime = [0] * 9

        master_key_prime = master_key ^ 0x0123456789ABCDEFFEDCBA9876543210
        for i in range(1, 9):
            key[i] = (master_key >> (16 * (8 - i))) & 0xFFFF
            key_prime[i] = (master_key_prime >> (16 * (8 - i))) & 0xFFFF

        for i in range(1, 9):
            self.key_KL1[i] = _shift(key[_mod(i + 0)], 1)
            self.key_KL2[i] = key_prime[_mod(i + 2)]
            self.key_KO1[i] = _shift(key[_mod(i + 1)], 5)
            self.key_KO2[i] = _shift(key[_mod(i + 5)], 8)
            self.key_KO3[i] = _shift(key[_mod(i + 6)], 13)
            self.key_KI1[i] = key_prime[_mod(i + 4)]
            self.key_KI2[i] = key_prime[_mod(i + 3)]
            self.key_KI3[i] = key_prime[_mod(i + 7)]

    def fun_FI(self, input, round_key):
        left = input >> 7
        right = input & 0b1111111

        round_key_1 = round_key >> 9
        round_key_2 = round_key & 0b111111111

        tmp_l = right
        tmp_r = S9[left] ^ right

        left = tmp_r ^ round_key_2
        right = S7[tmp_l] ^ (tmp_r & 0b1111111) ^ round_key_1

        tmp_l = right
        tmp_r = S9[left] ^ right

        left = S7[tmp_l] ^ (tmp_r & 0b1111111)
        right = tmp_r

        return (left << 9) | right

    def fun_FO(self, input, round_i):
        in_left = input >> 16
        in_right = input & 0xFFFF
```

```
        out_left = in_right
        out_right = self.fun_FI(in_left ^ self.key_KO1[round_i], self.key_KI1[round_i]) ^ in_right

        in_left = out_right
        in_right = self.fun_FI(out_left ^ self.key_KO2[round_i], self.key_KI2[round_i]) ^ out_right

        out_left = in_right
        out_right = self.fun_FI(in_left ^ self.key_KO3[round_i], self.key_KI3[round_i]) ^ in_right

        return (out_left << 16) | out_right

    def fun_FL(self, input, round_i):
        in_left = input >> 16
        in_right = input & 0xFFFF

        out_right = in_right ^ _shift(in_left & self.key_KL1[round_i], 1)
        out_left = in_left ^ _shift(out_right | self.key_KL2[round_i], 1)

        return (out_left << 16) | out_right

    def fun_f(self, input, round_i):
        if round_i % 2 == 1:
            state = self.fun_FL(input, round_i)
            output = self.fun_FO(state, round_i)
        else:
            state = self.fun_FO(input, round_i)
            output = self.fun_FL(state, round_i)

        return output

    def enc_1r(self, in_left, in_right, round_i):
        out_right = in_left
        out_left = in_right ^ self.fun_f(in_left, round_i)

        return out_left, out_right

    def dec_1r(self, in_left, in_right, round_i):
        out_left = in_right
        out_right = self.fun_f(in_right, round_i) ^ in_left

        return out_left, out_right

    def enc(self, plaintext):
        assert _bitlen(plaintext) <= 64
        left = plaintext >> 32
        right = plaintext & 0xFFFFFFFF
        for i in range(1, 9):
            left, right = self.enc_1r(left, right, i)
        return (left << 32) | right

    def dec(self, ciphertext):
        assert _bitlen(ciphertext) <= 64
        left = ciphertext >> 32
        right = ciphertext & 0xFFFFFFFF
        for i in range(8, 0, -1):
            left, right = self.dec_1r(left, right, i)
        return (left << 32) | right
```

```
def main():
    print("KASUMI Encryption")
    print("=================")

    plaintext_input = input("Enter plaintext (16 hex characters): ").strip()
    key_input = input("Enter key (32 hex characters): ").strip()

    try:
        plaintext = int(plaintext_input, 16)
        key = int(key_input, 16)
    except ValueError:
        print("Invalid input. Please enter valid hexadecimal values.")
        return

    kasumi = Kasumi()
    kasumi.set_key(key)
    ciphertext = kasumi.enc(plaintext)

    print("\nCiphertext:", hex(ciphertext))

if __name__ == "__main__":
    main()
```

# 6 Cryptographic Properties

- **Security**: Resists linear and differential cryptanalysis.
- **Efficiency**: Optimized for hardware and mobile devices.
- **Diffusion**: Ensured through FL and FO functions.
- **Confusion**: Strengthened via FI S-boxes.

# 7 Applications and Limitations

## 7.1 Applications

- UMTS security (f8, f9 algorithms)
- GSM (A5/3 encryption)

## 7.2 Limitations

- 64-bit block size makes it vulnerable to birthday attacks.
- Theoretical cryptanalysis has weakened its security.
- Largely replaced by AES in 4G and 5G.

# 8 Conclusion

Kasumi provides a well-balanced approach to security and efficiency, making it a cornerstone of mobile cryptography. Though newer ciphers have replaced it, Kasumi remains a benchmark in lightweight cryptographic design.

# 9 References


- kasumi Implementation of Kasumi, *KASUMI Block Cipher on the StarCore SC140 Core*, by Mao Zeng. Available at: KASUMI BLOCK CIPHER
- kasumi Implementation of Kasumi, *FPGA implementation of an enhanced chaotic-KASUMI block cipher*, Mahdi Madani, Camel Tanougast Available at: KASUMI BLOCK CIPHER
- kasumi KSM1, *Ultra-Compact Kasumi Cipher Core*, IP Cores, Inc. Available at: KSM1
- Kasumi - A5/3 - Cipher Test Vectors , *Kasumi - A5/3 - Cipher*, Test vectors of Kasum Block cipher by Asecuritysite.com. Available at: Test Vectors of Kasumi