Enhancing KASUMI Security by Affixing A Metamorphic Function and The Ensuing Hardware Implementation

Rabie A. Mahmoud1, A. Baith Mohamed2, Magdy Saeb3

1. Arab Academy for Science, Technology & Maritime Transport (AASTMT), Latakia Branch, Syria.
2. Arab Academy of Science, Technology & Maritime Transport (AASTMT), Alexandria, Egypt.
3. Arab Academy of Science, Technology & Maritime Transport (AASTMT), Alexandria, Egypt., Great Wall Information Security (GWIS), Kuala Lumpur, Malaysia.

www.great-wall-security.com
mail@magdysaeb.net

Abstract: To enhance the security of the well-known KASUMI Cipher, we apply a Metamorphic Feistel Structure. The proposed structure uses four bit-balanced operations in the function FO of the KASUMI Cipher. These operations are: XOR, INV, ROR, and NOP for bitwise XOR, invert, rotate right, and no operation respectively. The operations constitute the building blocks of the Crypto Logic Unit (CLU). In the second KASUMI Metamorphic modification, another Generalized Crypto Logic Unit (GCLU) is utilized. In addition to the four-bit-balanced CLU operations, the GCLU utilizes XNOR, SWAP, ROL, and RevOr for bitwise XNOR, swap, left rotation, and reverse order operations respectively. In this work, we present the Metamorphic-KASUMI Cipher, and the Generalized-Metamorphic-KASUMI Cipher. In addition, we provide a Field Programmable Gate Array (FPGA) implementation of the two proposed algorithm modifications.

Keywords: Metamorphic, KASUMI, Cipher, FPGA.

1. Introduction

KASUMI cipher [1], [2], [3] is a Feistel 64-bit block cipher using a 128-bit key with eight rounds and nonlinear S-boxes. KASUMI cipher forms the heart of the confidentiality and integrity algorithms. It provides signalling and user data security within the Global Systems for Mobile Communications (GSM), General Packet Radio Service (GPRS), Enhanced Data Rates for GSM Evolution (EDGE), and the Third Generation Mobile System (3GPP) specifications for the Universal Mobile Telecommunications System (UMTS) networks. KASUMI implementation use different techniques in terms of clock frequency, power consumption, and throughput [4], [5], [6], [7].

“In 2001, an impossible differential attack on six rounds of KASUMI was presented by Kühn (2001). In 2003 Elad Barkan, Eli Biham and Nathan Keller demonstrated man-in-the-middle attacks against the GSM protocol which avoided the A5/3 cipher and thus breaking the protocol. This approach does not attack the A5/3 cipher, however. The full version of their paper was published later in 2006. In 2005, Israeli researchers Eli Biham, Orr Dunkelman and Nathan Keller published a related-key bracelet (boomerang) attack on KASUMI that can break all 8 rounds faster than exhaustive search. The attack requires 254.6 chosen plaintexts, each of which has been encrypted under one of four related keys, and has a time complexity equivalent to 276.1 KASUMI encryptions. While this is not a practical attack, it invalidates some proofs about the security of the 3GPP protocols that had relied on the presumed strength of KASUMI. In 2010, Dunkelman, Keller and Shamir published a new attack that allows an adversary to recover a full A5/3 key by related-key attack. The time and space complexities of the attack are low enough that the authors carried out the attack in two hours on an Intel Core 2 Duo desktop computer even using the unoptimized reference KASUMI implementation. The authors note that this attack may not be applicable to the way A5/3 is used in 3G systems; their main purpose was to discredit 3GPP’s assurances that their changes to MISTY wouldn’t significantly impact the security of the algorithm” [8].

In this work, we present the Metamorphic-KASUMI Cipher. It is a metamorphic cipher that combines the crypto logic unit of the Stone Metamorphic Cipher [9], [10], [11] with the function FO of KASUMI Cipher to encrypt 64-bit plaintext packets using 128-bit key. The crypto logic unit (CLU) is used in many famous ciphers to increase the cipher’s entropy and improve its security such as the Metamorphic Twofish Cipher [12], the Metamorphic MARS Cipher [13], and the Metamorphic-Key-Hopping GOST Cipher [14]. This CLU is built using four low-level bit-balanced operations: XORing a key bit with a plaintext bit (XOR), inverting a plaintext bit (INV), exchanging one plaintext bit with another one in a given plaintext word using a right rotation operation (ROR), and producing a plaintext bit without any change (NOP). The generalized crypto logic unit [15] uses the four operations of the crypto logic unit plus XNORing a key bit with a plaintext bit (XNOR), swapping a plaintext bit with another one in a given plaintext word (SWAP), a left rotation operation (ROL), and the reverse order operation that reverses a plaintext word (RevOr) is used in the function FO of KASUMI cipher to provide what we call the Generalized-Metamorphic-KASUMI Cipher. In the following few sections, we provide the structure of the Metamorphic-KASUMI Cipher by describing the CLU and the enhanced function Meta-FO. Subsequently, we discuss the modifications required for the KASUMI Generalized our FPGA implementation for the KASUMI Metamorphic Cipher, a discussion of the results of the FPGA implementation including comparisons among modified KASUMI ciphers, a summary and our conclusions.
2. The Metamorphic-KASUMI Structure

KASUMI Metamorphic cipher is a Feistel Cipher with eight rounds encrypting 64-bit plaintext packets using 128-bit key. The KASUMI function \( FO \) is merged with the crypto logic unit to convert it into a Meta-\( FO \). Figure 1 shows the block diagram of the proposed KASUMI Metamorphic Cipher.

![Metamorphic-KASUMI Diagram](image)

**Figure 1: The structure of the KASUMI Metamorphic Cipher**

2.1 The Crypto Logic Unit (CLU)

The CLU is a round key-dependent function that is used to modify the KASUMI cipher. The four low-level bit-balanced operations are the building blocks of the CLU:
- (XOR) by XORing a key bit with a plaintext bit,
- (INV) by inverting a plaintext bit,
- (ROR) by exchanging one plaintext bit with another one in a given plaintext word using a right rotation operation,
- (NOP) by producing the plaintext without any change.

Table 1 demonstrates each one of CLU operations and more details can be found in [9] where 2-bit operation selection bits (OSB) and 3-bit rotation selection bits (RSB) for crypto logic unit in function Meta-\( FO \) are chosen from the \( KO_{i,j} \) round keys in the KASUMI Metamorphic cipher. The operation selection bits determine the applied operation in CLU and the rotation selection bits determine the number of rotations when ROR operation is used.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Operation</th>
<th>Select Operation code</th>
</tr>
</thead>
<tbody>
<tr>
<td>XOR</td>
<td>( C_i = K_i \oplus P_i )</td>
<td>“00”</td>
</tr>
<tr>
<td>INV</td>
<td>( C_i = \neg (P_i) )</td>
<td>“01”</td>
</tr>
<tr>
<td>ROR</td>
<td>( P_i \leftarrow (P_i, m) )</td>
<td>“10”</td>
</tr>
<tr>
<td>NOP</td>
<td>( C_i = P_i )</td>
<td>“11”</td>
</tr>
</tbody>
</table>

2.2 The Function Meta-\( FO \)

The input to the function Meta-\( FO \) comprises
- 32-bit data input \( I \),
- 48-bit subkey \( KO_i \),
- 48-bit subkey \( KL_j \).

The 32-bit data input is split into two halves, \( L_0 \) and \( R_0 \) where

\[
I = L_0 \parallel R_0
\]

The 48-bit subkeys are subdivided into three 16-bit subkeys where

\[
KO_i = KO_{0,1} \parallel KO_{0,2} \parallel KO_{0,3}
\]

and

\[
KL_j = KL_{1,1} \parallel KL_{1,2} \parallel KL_{1,3}
\]

where for each integer \( j \) with \( 1 \leq j \leq 3 \) choose form \( KO_{i,j} \) 2-bit operation selection_bits, and 3-bit rotation_selection_bits.

We define

- If operation_selection_bits = “00” then
  \[
  R_j = F(I_{j-1} \oplus KO_{i,j}, KL_{i,j}) \oplus R_{j-1}
  \]
  \[
  L_j = R_{j-1}
  \]

- If operation_selection_bits = “01” then
  \[
  R_j = F(I_{j-1}, KL_{i,j}) \oplus R_{j-1}
  \]
  \[
  L_j = R_{j-1}
  \]

- If operation_selection_bits = “10” then
  \[
  R_j = F(ROR(I_{j-1}, m), KL_{i,j}) \oplus R_{j-1}
  \]
  \[
  L_j = R_{j-1}
  \]

- If operation_selection_bits = “11” then
  \[
  R_j = F(L_{j-1}, KL_{i,j}) \oplus R_{j-1}
  \]
  \[
  L_j = R_{j-1}
  \]

Finally, we return the 32-bit value \( (L_3 \parallel R_3) \).

3. The KASUMI Generalized Metamorphic Cipher

The KASUMI Generalized Metamorphic cipher is a modified form of the KASUMI cipher through adding the generalized crypto logic unit to the function \( FO \) of the KASUMI cipher. This is accomplished in the same way of KASUMI Metamorphic cipher and converting it into a Generalized-Meta-\( FO \). The Generalized Crypto Logic Unit (GCLU) has eight low-level bit-balanced operations:

- (XOR) by XORing a key bit with a plaintext bit,
- (INV) by inverting a plaintext bit,
- (ROR) by exchanging one plaintext bit with another one in a given plaintext word using a right rotation operation,
- (NOP) by producing the plaintext without any change,
- (XNOR) by inverting a plaintext bit,
- (SWAP) by exchanging one plaintext bit with another one in a given plaintext word using a swap operation,
- (ROL) by exchanging one plaintext bit with another one in a given plaintext word using a left rotation operation,
- (RevOr) by exchanging one plaintext bit with another one in a given plaintext word using a reverse order operation.

Figure 2 shows the generalized crypto logic unit GCLU. More details are found in [15] where the 3-bit operation selection bits (OSB) and the 3-bit rotation selection bits (RSB) are chosen from the \( KO_{i,j} \) round keys the Generalized-Meta-\( FO \) function.
4. The FPGA Implementation

The KASUMI Metamorphic Cipher FPGA-based implementation is applied to encrypt 64-bit plaintext packet using 128-bit user key producing 64-bit ciphertext packet at each cycle. We have implemented the Metamorphic-KASUMI cipher applying the VHDL hardware description language [16], [17], [18] and utilizing Altera design environment Quartus II 15.0 (64-bit) Web Edition [19] with ModelSim Altera Starter Edition 10.3d [20]. The FPGA design was implemented using EP4CGX50DF27C6, Cyclone IV GX family device. The schematic diagram of proposed cipher with the implementation results is shown in Figure 3. RTL screens of the FPGA implementation details are shown in Figures 4, and 5. Figures 6, 7, and 8 show the Technology Map Viewer of the hardware implementation of the cipher. Figure 9 demonstrates the floor plan for proposed modified cipher and Figure 10 displays the simulation showing the input, key, and the output cipher text bits. The details of the analysis and synthesis summary and timing analyzer are shown in appendix A. Major synthesis and timing differences among KASUMI, Metamorphic-KASUMI and Generalized-Metamorphic-KASUMI ciphers in the balanced optimization technique are shown in Appendix B. Appendix C demonstrates the hierarchical design and sample VHDL code for the KASUMI Metamorphic Cipher.

Figure 2: The generalized crypto logic unit (GCLC)

Figure 4: RTL screen of Metamorphic-KASUMI cipher

Figure 5: RTL screen for part of Metamorphic-KASUMI cipher

Figure 6: Technology Map Viewer of Metamorphic-KASUMI cipher

Figure 7: Technology Map Viewer for three rounds of Metamorphic-KASUMI cipher

Figure 3: Compiler tool screen showing correct implementation and schematic diagram of Metamorphic-KASUMI cipher
5. Summary and Conclusions

We have proposed two modified ciphers that are based on the KASUMI cipher. Those modified ciphers are called the KASUMI Metamorphic cipher and the KASUMI Generalized-Metamorphic Cipher. A crypto logic unit that utilizing the bit-balanced operations s XOR, INV, ROR, and NOP is immixed in each round of the function FO of KASUMI cipher. Likewise, a generalized crypto logic unit, based on XOR, INV, ROR, NOP, XNOR, SWAP, ROL, and RevOr bit-balanced operations, is merged in each round of the function FO of the KASUMI cipher. In addition, we have presented a proof-of-concept FPGA hardware implementation of the proposed cipher. Various FPGA optimization techniques namely Balanced, High Performance Effort, Aggressive Performance, High Power Effort, Aggressive Power, and Aggressive Area optimization techniques were compared for proposed ciphers. Moreover, resources and timing delays comparisons between the KASUMI, the KASUMI Metamorphic and the KASUMI Generalized Metamorphic were shown. Certainly, the increased security comes at a cost of increased resources.

References

[1] 3GPP’s site: http://www.3gpp.org
Appendix A: The fitter and timing report details of implementing Metamorphic-KASUMI cipher

FPGA synthesis of Metamorphic-KASUMI cipher is implemented in Balanced, High Performance Effort, Aggressive Performance, High Power Effort, Aggressive Power, and Aggressive Area optimization techniques determining the usage number of logic elements, connections, and time delays where Balanced optimizes synthesis for balanced implementation with respects to timing constraints, High Performance Effort makes high effort to optimize synthesis for speed performance by increases synthesis run time, Aggressive Performance makes aggressive effort to optimize synthesis for speed performance by increases synthesis run time and device resource use, High Power Effort makes high effort to optimize synthesis for low power by increases synthesis run time, Aggressive Power makes aggressive effort to optimize synthesis for low power by increases synthesis time and reduces speed performance, and Aggressive Area makes aggressive effort to reduce the device area required to implement the design. Also, no time restrictions are applied in Metamorphic-KASUMI cipher implementation but Slow 1200mV 85°C Timing Model with slowest silicon, low voltage, and 85°C junction temperature conditions; Slow 1200mV 0°C Timing Model with slowest silicon, low voltage, and 0°C junction temperature conditions; Fast 1200mV 0°C Timing Model with fast silicon, high voltage, and 0°C junction temperature conditions provide timing delays for the FPGA. Table A1 shows the number of usage logic elements and their interconnections among optimization techniques of implementing Metamorphic-KASUMI cipher, and Table A2 shows the timing delays among optimization techniques related with timing model. Figure A.1 shows a comparison chart of timing delays for Metamorphic-KASUMI designs. We noticed that optimization technique implementations of Metamorphic-KASUMI consumed a similar number of usage resources and different routing and interconnects between logic elements except Aggressive Performance optimization technique where these similar usage resources and different routing related with nested functions VHDL programming of Metamorphic-KASUMI and compiler’s high level optimization strategy. In Aggressive Performance optimization technique, the increasing of 4-input functions and direct links decreased the Fan-Out which caused rising the speed through reducing the propagation delays. Also, the increasing of block interconnects and direct links in High power Effort optimization technique saved dynamic power but affecting the performance through rising the propagation delays. Although, the Aggressive Power optimization technique consumed less routing than High power Effort optimization technique to save the power but provided less performance than High power Effort optimization technique.

Analysis & Synthesis and Fitter Summary
- Family: Cyclone IV GX
- Device: EP4CGX50DF27C6
- Nominal Core Voltage: 1.20 V
- Minimum Core Junction Temperature: 0 °C
- Maximum Core Junction Temperature: 85 °C.
- Optimization Technique: Balanced
  - Total logic elements: 7,622 out of 49,888 (15%)  
    -- Combinational with no register: 7,622  
    -- Register only: 0  
    -- Combinational with a register: 0
- Logic element usage by number of LUT inputs
  - 4 input functions: 5,732
  - 3 input functions: 1,333
  - <=2 input functions: 557
  -- Register only: 0
- Logic elements by mode
  -- Normal mode: 7,622
  -- Arithmetic mode: 0
  - Total LABs: 755 out of 3,118 (18 %)
  - I/O pins: 256 out of 343 (75 %)
    -- Clock pins: 2 out of 10 (20 %)
    -- Dedicated input: 0 out of 25 (0 %)
  - Total block memory bits: 0 out of 2,562,048 (0 %)
  - Embedded Multiplier 9-bit elements: 0 out of 280 (0 %)
  - Maximum fan-out: 93
  - Highest non-global fan-out: 93
  - Total fan-out: 28,366
  - Average fan-out: 3.48
  - Average interconnect usage (total/H/V): 5.1% / 4.5% / 5.8%
  - Peak interconnect usage (total/H/V): 24.1% / 22.6% / 27.6%
  - Block interconnects: 10,033 out of 232,464 (4 %)
  - C16 interconnects: 783 out of 6,642 (12 %)
  - C4 interconnects: 5,906 out of 136,080 (4 %)
  - Direct links: 1,193 out of 232,464 (< 1 %)
  - GXB block output buffers: 0 out of 2,640 (0 %)
  - Global clocks: 0 out of 30 (0 %)
  - Interquad Reference Clock Outputs: 0 out of 2 (0 %)
  - Interquad TXRX Clocks: 0 out of 16 (0 %)
  - Interquad TXRX PCSRX outputs: 0 out of 8 (0 %)
  - Interquad TXRX PCSTX outputs: 0 out of 8 (0 %)
  - Local interconnects: 4,412 out of 73,920 (6 %)
  - R24 interconnects: 703 out of 6,930 (10 %)
  - R4 interconnects: 5,818 out of 190,740 (3 %)

TimeQuest Timing Analyzer Summary
- Slow 1200mV 85°C Model
  - Longest propagation delay RR which is measured from rising edge to rising edge was 194.146 ns from input port “INPUT[35]” to output port “Output[60]”. Also, longest delay RF which is measured from rising edge to falling edge
was 194.096 ns, longest delay FR which is measured from falling edge to rising edge was 194.815 ns, and longest delay FF which is measured from falling edge to falling edge was 194.765 ns.

- Longest minimum propagation delay was from input port “INPUT[20]” to output port “Output[27]” where RR was 26.335 ns, RF was 26.322 ns, FR was 26.891 ns, and FF was 26.878 ns.

- Slow 1200mV 0°C Model
  - Longest propagation delay was from input port “INPUT[35]” to output port “Output[60]” where RR was 174.609 ns, RF was 174.587 ns, FR was 175.064 ns, and FF was 175.042 ns.
  - Longest minimum propagation delay was from input port “INPUT[20]” to output port “Output[27]” where RR was 23.697 ns, RF was 23.723 ns, FR was 24.152 ns, and FF was 24.178 ns.

- Fast 1200mV 0°C Model
  - Longest propagation delay was from input port “INPUT[35]” to output port “Output[60]” where RR was 113.251 ns, RF was 112.990 ns, FR was 114.087 ns, and FF was 113.826 ns.
  - Longest minimum propagation delay was from input port “INPUT[20]” to output port “Output[27]” where RR was 15.098 ns, RF was 14.835 ns, FR was 15.816 ns, and FF was 15.553 ns.

<table>
<thead>
<tr>
<th>Table A1: A resource and routing usage comparison among optimization technique implementations of Metamorphic-KASUMI cipher</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td>---------------------------------------------------------------</td>
</tr>
<tr>
<td>Total LEs</td>
</tr>
<tr>
<td>4 input Function</td>
</tr>
<tr>
<td>3 input</td>
</tr>
<tr>
<td>≤2 input</td>
</tr>
<tr>
<td>Fan-Out Total</td>
</tr>
<tr>
<td>Max</td>
</tr>
<tr>
<td>Average</td>
</tr>
<tr>
<td>Interconnects</td>
</tr>
<tr>
<td>Block</td>
</tr>
<tr>
<td>C16</td>
</tr>
<tr>
<td>C4</td>
</tr>
<tr>
<td>R24</td>
</tr>
<tr>
<td>Locals</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table A2: A timing delays comparison among optimization technique implementations of Metamorphic-KASUMI cipher</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td>---------------------------------------------------------------</td>
</tr>
<tr>
<td>Slow 1200mV 0°C Model</td>
</tr>
<tr>
<td>RR</td>
</tr>
<tr>
<td>RF</td>
</tr>
<tr>
<td>FF</td>
</tr>
</tbody>
</table>

| Slow 1200mV 0°C Model | 113.251  | 101.413          | 96.914                  | 102.321    | 105.743         | 113.251       |
| RR              | 112.990  | 100.650          | 96.639                  | 101.901    | 105.374         | 112.990       |
| RF              | 114.087  | 101.306          | 97.880                  | 103.161    | 106.475         | 114.087       |
| FF              | 113.826  | 101.498          | 97.605                  | 102.741    | 106.106         | 113.826       |

| Fast 1200mV 0°C Model | 15.098   | 13.221           | 13.628                  | 14.786     | 14.748          | 15.098        |
| RF              | 15.816   | 13.911           | 14.269                  | 15.514     | 15.487          | 15.816        |
| FF              | 15.553   | 14.093           | 14.061                  | 15.133     | 15.117          | 15.553        |

![Figure A.1: Delays of optimization technique implementations of Metamorphic-KASUMI cipher](image)
Appendix B: Major implementing differences among KASUMI and modified KASUMI ciphers in Balanced optimization technique

Metamorphic-KASUMI cipher consumes more resources and routing than KASUMI cipher where the four operations in each crypto logic unit in function Meta-FO especially right rotation operation consume more resources and routing than XOR operations in regular function FO of KASUMI cipher. Moreover, each generalized crypto logic unit in function Generalized-Meta-FO of Generalized-Metamorphic-KASUMI cipher consumes more resources and routing comparing with KASUMI cipher because including eight operations with both right and left rotation operations. Table B1 shows the number of usage logic elements and their interconnects, and Table B2 shows the timing delays among KASUMI, Metamorphic-KASUMI, and Generalized-Metamorphic-KASUMI in Balanced optimization technique. Figure B.1 shows a comparison chart of those timing delays for our designs.

Table B1: A resource and routing usage comparison among KASUMI and modified KASUMI ciphers

<table>
<thead>
<tr>
<th>Function</th>
<th>KASUMI</th>
<th>Metamorphic-KASUMI</th>
<th>Generalized-Metamorphic-KASUMI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total LEs</td>
<td>5582</td>
<td>7622</td>
<td>10559</td>
</tr>
<tr>
<td>4 input</td>
<td>4365</td>
<td>5732</td>
<td>7759</td>
</tr>
<tr>
<td>3 input</td>
<td>665</td>
<td>1333</td>
<td>2026</td>
</tr>
<tr>
<td>&lt;=2 input</td>
<td>552</td>
<td>557</td>
<td>774</td>
</tr>
<tr>
<td>Total</td>
<td>20884</td>
<td>28366</td>
<td>38987</td>
</tr>
<tr>
<td>Max</td>
<td>31</td>
<td>93</td>
<td>97</td>
</tr>
<tr>
<td>Average</td>
<td>3.42</td>
<td>3.48</td>
<td>3.52</td>
</tr>
<tr>
<td>Block</td>
<td>7733</td>
<td>10033</td>
<td>13633</td>
</tr>
<tr>
<td>C16</td>
<td>702</td>
<td>783</td>
<td>894</td>
</tr>
<tr>
<td>C4</td>
<td>4773</td>
<td>5906</td>
<td>8324</td>
</tr>
<tr>
<td>Local</td>
<td>3124</td>
<td>4412</td>
<td>6142</td>
</tr>
<tr>
<td>R24</td>
<td>599</td>
<td>703</td>
<td>846</td>
</tr>
<tr>
<td>R4</td>
<td>5103</td>
<td>5818</td>
<td>7819</td>
</tr>
<tr>
<td>Direct Links</td>
<td>939</td>
<td>1193</td>
<td>1564</td>
</tr>
</tbody>
</table>

Table B2: A timing delays comparison among KASUMI and modified KASUMI ciphers

<table>
<thead>
<tr>
<th>Slow 1200mV 0°C Model</th>
<th>KASUMI</th>
<th>Metamorphic-KASUMI</th>
<th>Generalized-Metamorphic-KASUMI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Slow Longest</td>
<td>RR</td>
<td>148.443</td>
<td>194.146</td>
</tr>
<tr>
<td></td>
<td>RF</td>
<td>148.506</td>
<td>194.096</td>
</tr>
<tr>
<td></td>
<td>FR</td>
<td>149.028</td>
<td>194.815</td>
</tr>
<tr>
<td></td>
<td>FF</td>
<td>149.091</td>
<td>194.765</td>
</tr>
<tr>
<td>Slow Longest</td>
<td>RR</td>
<td>25.630</td>
<td>26.335</td>
</tr>
<tr>
<td></td>
<td>RF</td>
<td>25.627</td>
<td>26.322</td>
</tr>
<tr>
<td></td>
<td>FR</td>
<td>26.143</td>
<td>26.891</td>
</tr>
</tbody>
</table>

Figure B.1: Delays of implementations of modified KASUMI ciphers

Appendix C: Sample VHDL code of implementing Generalized-Metamorphic-KASUMI cipher

Hierarchy design is used to implement Generalized-Metamorphic-KASUMI cipher through programming multi-nested VHDL functions defined in packages:

- ROL_pkg package includes definitions for fifteen left circular rotation bit functions (ROL1, ROL2, ..., ROL15) where ROL1 function is a 1-bit left circular rotation function which has one 16-bit input and return 16-bit rotated 1-bit output; ROL2 function is a 2-bit left circular rotation function which has one 16-bit input and return 16-bit rotated 2-bit output; and so on to ROL15 which is a 15-bit left circular rotation function.

- ROR_pkg package includes definitions for fifteen right circular rotation bit functions (ROR1, ROR2, ..., ROR15).
applied on 16-bit input where ROR function is a 1-bit right circular rotation function; and so on to ROR15.
• ZE_pkg package includes ZE (Zero Extend) function which converts the 7-bit input to 9-bit output by adding two zero bits to the most-significant end.
• TR_pkg package includes TR (TRuncate) function which converts the 9-bit input to 7-bit output by discarding the two most-significant bits.
• S9_pkg package includes S9 function which is a 9-bit S-box implemented in combinational logic.
• S7_pkg package includes S7 function which is a 7-bit S-box implemented also in combinational logic.
• FLi_pkg package includes FLi function with three input parameters which are 32-bit input, 16-bit KLi1 and 16-bit KLi2 the keys for FLi where 1 ≤ i ≤ 8 is related to the number of KASUMI round. FLi function returns a 32-bit output after applying the FL function specific series of operations besides calling ROL1 function from ROL_pkg package.
• Fli_pkg package includes Fli function with two input parameters which are 16-bit input, and 16-bit Klij the key for Fli where 1 ≤ j ≤ 3 is related to the number of FOij round. Fli function returns a 16-bit output after applying the FI function specific series of operations besides calling S9, S7, ZE, and TR functions from S9_pkg, S7_pkg, ZE_pkg, and TR_pkg packages respectively.
• GCLU_pkg package includes GCLU function representing the Generalized Crypto Logic Unit function and has two input parameters which are 16-bit input, and 16-bit KOi the key for FOij where the operation_selection_bits (2-bit OSB), and the rotation_selection_bits (3-bit RSB) are selected from KOij. GCLU function returns a 16-bit output after applying the GCLU operations which are related to OSB besides calling all left and right rotation functions from ROL_pkg, and ROR_pkg packages.
• FOij_pkg package includes FOij function representing one round of the 3 rounds in FOi function with three input parameters which are 32-bit input, 16-bit KOi, and 16-bit Klij. FOij function returns a 32-bit output after applying the one round specific operations of FO besides calling Flij, and GCLU functions from Flij_pkg, and GCLU_pkg packages respectively which need Klij and KOij keys.
• FOi_pkg package includes FOi function which calls FOij function from FOij_pkg package three times to implement the three rounds of FO function and returns a 32-bit output. FOi function has seven input parameters which are used as 32-bit input, 16-bit KO1, and 16-bit Kl1 for round1; the output of round1, 16-bit Kli2, and 16-bit KOi2 for round2; and the output of round2, 16-bit KO1, and 16-bit Kl1 for round3.
• fi_OddRounds_pkg package includes fi_OddRounds function with nine input parameters which are 32-bit input and all round keys 16-bit KLi1, 16-bit KLi2, 16-bit KO1, 16-bit KOi2, 16-bit KO3, 16-bit Kl1, 16-bit Kl2, and 16-bit Kl3. fi_OddRounds function returns a 32-bit output after passing the input and the keys through FLi and then FOi functions from FLi_pkg and FOi_pkg packages respectively.
• fi_EvenRounds_pkg package includes fi_EvenRounds function which is similar to fi_OddRounds function but it returns a 32-bit output after passing the input and the keys through FOi and then FLi functions. Then using component configuration of generated statements to program Key_Schedual, OddRound, and EvenRound components where Key_Schedual component is responsible to generate all round keys which are derived from 128-bit input key K for each round. OddRound component is used for rounds 1, 3, 5, and 7 where it takes a 64-bit input, and eight sub-keys from Key_Schedual component related with each specific round then returns 64-bit output under control of fi_OddRounds function. EvenRound component is similar to OddRound component but it is used in rounds 2, 4, 6, and 8 where returns 64-bit output under control of fi_EvenRounds function. Finally, Key_Schedual and eight round blocks are connected together in block diagram/schematic file. Sample VHDL codes are:

```vhdl
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;
USE WORK.ROL_pkg.ALL; -- To call ROL1 Function --

PACKAGE FLi_pkg IS
FUNCTION FLi(Input : in std_logic_vector(31 downto 0);
KL1 : in std_logic_vector(15 downto 0);
KL2 : in std_logic_vector(15 downto 0)) RETURN std_logic_vector;
END FLi_pkg;

PACKAGE BODY FLi_pkg IS
FUNCTION FLi(Input : in std_logic_vector(31 downto 0);
KL1 : in std_logic_vector(15 downto 0);
KL2 : in std_logic_vector(15 downto 0)) RETURN std_logic_vector IS
-- The input halves L and R --
VARIABLE Input_L : std_logic_vector(15 downto 0);
VARIABLE Input_R : std_logic_vector(15 downto 0);
-- The output halves L' and R' --
VARIABLE Output_L:std_logic_vector(15 downto 0);
VARIABLE Output_R:std_logic_vector(15 downto 0);
VARIABLE Output : std_logic_vector(31 downto 0);
BEGIN
-- Splitting the Input
Input_L := Input(31 downto 16);
Input_R := Input(15 downto 0);
-- Operation Series
Output_R:= Input_RXOR(ROL1(Input_L AND KL11));
Output_L:= Input_LXOR(ROL1(Output_R OR KL22));
-- Output of Function
Output := Output_L & Output_R;
RETURN Output;
END FUNCTION FLi;
END PACKAGE BODY FLi_pkg;
```

```vhdl
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;
USE WORK.S7_pkg.ALL; -- To call S7 Function --
USE WORK.S9_pkg.ALL; -- To call S9 Function --
USE WORK.ZE_pkg.ALL; -- To call ZE Function --
USE WORK.TR_pkg.ALL; -- To call TR Function --
```

-- VHDL Code for Flij_pkg

```vhdl
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;
```

---

Sample VHDL code for FLi_pkg and Flij_pkg packages, implementing the described functionalities for the KASUMI cipher, including input and output formats, variable declarations, and function implementations for each of the components outlined in the text.
FUNCTION Flij (Input : in std_logic_vector(15 downto 0); Klij : in std_logic_vector(15 downto 0))
    RETURN std_logic_vector;
END Flij_pkg;

PACKAGE BODY Flij_pkg IS
FUNCTION Flij (Input : in std_logic_vector(15 downto 0); Klij : in std_logic_vector(15 downto 0))
    RETURN std_logic_vector IS
    VARIABLE Klij1 : std_logic_vector(6 downto 0);
    VARIABLE Klij2 : std_logic_vector(8 downto 0);
    VARIABLE R0, R2 : std_logic_vector(6 downto 0);
    VARIABLE L1, L3, L4 : std_logic_vector(8 downto 0);
    VARIABLE R1, R3, R4 : std_logic_vector(8 downto 0);
    VARIABLE L0, L2 : std_logic_vector(8 downto 0);
    VARIABLE Output : std_logic_vector(15 downto 0);
BEGIN
    -- Splitting the key Klij
    Klij1 := Klij(15 downto 9);
    Klij2 := Klij(8 downto 0);
    -- Splitting the Input
    L0 := Input(15 downto 7);
    R0 := Input(6 downto 0);
    -- Operation Series
    L1 := R0;
    R1 := S9(L0) XOR ZE(R0);
    L2 := R1 XOR Klij2;
    R2 := S7(L1) XOR TR(R1) XOR Klij1;
    L3 := R2;
    R3 := S9(L2) XOR ZE(R2);
    L4 := S7(L3) XOR TR(R3);
    R4 := R3;
    -- Output of Function
    Output := L4 & R4;
RETURN Output;
END FUNCTION Flij;
END PACKAGE BODY Flij_pkg;

VHDL Code for GCLU_pkg
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;
USE WORK.ROR_pkg.ALL; -- To call ROR Functions
USE WORK.ROL_pkg.ALL; -- To call ROL Functions

PACKAGE GCLU_pkg IS
FUNCTION GCLU(Input: in std_logic_vector(15 downto 0); KOij: in std_logic_vector(15 downto 0))
    RETURN std_logic_vector;
END GCLU_pkg;

PACKAGE BODY GCLU_pkg IS
FUNCTION GCLU(Input: in std_logic_vector(15 downto 0); KOij: in std_logic_vector(15 downto 0))
    RETURN std_logic_vector IS
    VARIABLE Output: std_logic_vector(15 downto 0);
BEGIN
    OSB := KOij(7) & KOij(5) & KOij(3);
    RSB := KOij(0) & KOij(5) & KOij(10) & KOij(15);
    IF OSB ="000" THEN Output := Input XOR KOij;
    ELSIF OSB ="001" THEN Output := NOT Input;
    ELSIF OSB ="011" THEN Output := Input;
    ELSIF OSB ="100" THEN Output := Input XNOR KOij;
    ELSIF OSB ="101" THEN Output := Input(7 downto 0) & Input(15 downto 8);
    ELSIF OSB ="111" THEN Output := Input(0) & Input(1) & Input(2) & Input(3) & Input(4) & Input(5) & Input(6) & Input(7) & Input(8) & Input(9) & Input(10) & Input(11) & Input(12) & Input(13) & Input(14) & Input(15);
    ELSIF OSB ="010" THEN
        IF RSB ="0000" THEN Output:= Input ;
        ELSIF RSB ="0001" THEN Output:=ROL1(Input) ;
        ELSIF RSB ="0010" THEN Output:=ROL2(Input) ;
        ELSIF RSB ="0011" THEN Output:=ROL3(Input) ;
        ELSIF RSB ="1110" THEN Output:=ROR14(Input) ;
        ELSIF RSB ="1111" THEN Output:=ROR15(Input) ;
    END IF;
    ELSIF OSB ="110" THEN
        IF RSB ="0000" THEN Output:= Input ;
        ELSIF RSB ="0001" THEN Output:=ROL1(Input) ;
        ELSIF RSB ="0010" THEN Output:=ROL2(Input) ;
        ELSIF RSB ="0011" THEN Output:=ROL3(Input) ;
        ELSIF RSB ="1110" THEN Output:=ROR14(Input) ;
        ELSIF RSB ="1111" THEN Output:=ROR15(Input) ;
    END IF;
    END IF;
RETURN Output;
END FUNCTION GCLU;
END PACKAGE BODY GCLU_pkg;
VARIABLE Output : std_logic_vector(31 downto 0);
BEGIN
  -- Splitting the Input
  Input_L := Input(31 downto 16);
  Input_R := Input(15 downto 0);
  -- Operation Series
  Output_L := Input_R;
  Output_GCLU := GCLU(Input_L , KOij);
  Output_R := FLi(O(input_GCLU , KIij)) XOR Input_R;
  -- Output of Function
  Output := Output_L & Output_R;
  RETURN Output;
END FUNCTION FOij;
END PACKAGE BODY FOij_pkg;

VHDL Code for fi_OddRounds_pkg
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;
USE WORK.FLi_pkg.ALL; -- To call FLi Function --
USE WORK.FOi_pkg.ALL; -- To call FOi Function --
PACKAGE fi_OddRounds_pkg IS
FUNCTION fi_OddRounds (Input : in std_logic_vector(31 downto 0);
                        KLi1 : in std_logic_vector(15 downto 0);
                        KLi2 : in std_logic_vector(15 downto 0);
                        KOi1 : in std_logic_vector(15 downto 0);
                        KOi2 : in std_logic_vector(15 downto 0);
                        KOi3 : in std_logic_vector(15 downto 0);
                        KIi1 : in std_logic_vector(15 downto 0);
                        KIi2 : in std_logic_vector(15 downto 0);
                        KIi3 : in std_logic_vector(15 downto 0)))
RETURN std_logic_vector;
END fi_OddRounds_pkg;
PACKAGE BODY fi_OddRounds_pkg IS
FUNCTION fi_OddRounds (Input : in std_logic_vector(31 downto 0);
                        KLi1 : in std_logic_vector(15 downto 0);
                        KLi2 : in std_logic_vector(15 downto 0);
                        KOi1 : in std_logic_vector(15 downto 0);
                        KOi2 : in std_logic_vector(15 downto 0);
                        KOi3 : in std_logic_vector(15 downto 0);
                        KIi1 : in std_logic_vector(15 downto 0);
                        KIi2 : in std_logic_vector(15 downto 0);
                        KIi3 : in std_logic_vector(15 downto 0)))
RETURN std_logic_vector IS
  VARIABLE Output : std_logic_vector(31 downto 0);
BEGIN
  Output := FOi (FLi (Input, KLi1, KLi2) , KOi1, KOi2, KOi3, KIi1, KIi2, KIi3);
  RETURN Output;
END FUNCTION fi_OddRounds;
END PACKAGE BODY fi_OddRounds_pkg;