Cryptanalysis of the ARADI Block Cipher

Chandan Dey

Visiting Scientist,
R. C. Bose Centre for Cryptology and Security,
Indian Statistical Institute Kolkata

1 Introduction

The ARADI [6] block cipher, developed by the U.S. National Security Agency (NSA) , is specifically engineered for low-latency applications in memory encryption. It was introduced alongside a complementary mode of operation, LLAMA, designed to provide authenticated encryption, thereby ensuring both confidentiality and integrity for computer memory. As modern processor architectures increasingly protect CPUs within secure enclaves to safeguard sensitive computations, providing equivalent protections for main memory (RAM) remains a significant challenge due to inherent physical and architectural limitations. These challenges have made RAM a frequent target for physical attacks, such as cold boot attacks, which exploit data remanence in RAM to recover cryptographic keys and other sensitive information.

In response, advanced cryptographic memory protection schemes have been developed that encrypt the contents of RAM and verify the integrity using cryptographic checks. However, the design of such systems requires a careful balance between robust security and stringent performance demands, as encryption must be rapid enough to avoid adding latency to memory accesses.

ARADI addresses these challenges by optimizing its cipher structure to deliver extremely low latency without compromising strong security guarantees. This balance makes ARADI particularly well-suited for modern computing environments that demand fast and secure data protection. Its focus on speed coupled with authenticated encryption is especially crucial given the rise of increasingly sophisticated physical memory attacks that threaten both data confidentiality and integrity.

ARADI is a lightweight block cipher designed to provide fast and secure memory encryption. However, its security needs to be carefully tested to see how well it can resist real-world cryptanalytic attacks. So far, no one has studied how ARADI behaves under fault attacks, although a recent side-channel attack [7] and some traditional cryptanalysis studies [2, 3, 5] have examined some of its weaknesses in reduced-round versions.

In our work [4], we present the first Differential Fault Analysis (DFA) attack on ARADI. Our approach uses a nibble-based fault model and shows that it is possible to recover the secret key under realistic fault injection conditions.

This report presents the design of the ARADI block cipher, its implementation in C, and a survey of recent cryptanalysis and our recent differential fault attack on ARADI[4].

2 Design of ARADI

In this section, we discuss the design of the ARADI block cipher. The ARADI block cipher is architected based on the Substitution–Permutation Network (SPN) paradigm, which is foundational in modern symmetric encryption design. In ARADI, encryption and decryption processes operate over 128-bit plaintext blocks, providing a robust structure for both efficiency and security. The cipher's cryptographic transformations are parameterized by a 256-bit master key, ensuring a large keyspace that resists brute-force attacks and aligns with contemporary security standards.

Internal State Representation

The 128-bit internal state can be represented as four 32-bit words

$$(w, x, y, z),$$

and arranged in a 4×32 bit matrix. Here, w is the most significant word, while z is the least significant.

Round Function

The round function of ARADI consists of three main components: the substitution layer (SBox), the linear layer, and the round key addition. We adopt the same notation as the original specification [6].

- The substitution layer is denoted by π,
- The i-th linear layer is denoted by Λ_i,
- The XOR with the 128-bit round key is denoted by τ_{k_i}.

Hence, the i-th round function is

$$R_i = \tau_{k_i} \circ \Lambda_i \circ \pi,$$

where k_i is the round key.

An initial key addition, referred to as *pre-whitening*, is applied before the first round, denoted by τ_{k_0}. The cipher consists of 16 rounds, and the full encryption is expressed as

$$(\tau_{k_{16}} \circ \Lambda_{16} \circ \pi) \circ \cdots \circ (\tau_{k_1} \circ \Lambda_1 \circ \pi) \circ \tau_{k_0}.$$

S-Box Layer (π): The nonlinear layer π of ARADI operates on the four 32-bit words (w, x, y, z) of the internal state. Instead of being defined via a lookup table, the transformation can be equivalently expressed as a set of word-level Boolean operations:

$$x \leftarrow x \oplus (w \,\&\, y),$$
$$z \leftarrow z \oplus (x \,\&\, y),$$
$$y \leftarrow y \oplus (w \,\&\, z),$$
$$w \leftarrow w \oplus (x \,\&\, z),$$

where $\oplus$ denotes bitwise XOR and & denotes bitwise AND.

Linear Layer (Λ): The linear layer updates each row of the internal state via a round-dependent linear transformation. At round $i \in \{1, \ldots, 16\}$, the map Λ_i acts on the state S as

$$\Lambda_i(S) \;=\; \big(L_j(w),\, L_j(x),\, L_j(y),\, L_j(z)\big), \qquad j \;=\; (i-1) \bmod 4.$$

Each L_0, L_1, L_2, L_3 is an involutive linear mapping over $\mathbb{F}_2^{32}$. Writing a 32-bit row as the concatenation of two 16-bit halves, u (upper) and l (lower), we define

$$L_j(u \parallel l) \;=\; \Big(u \oplus (u \lll a_j) \oplus (l \lll c_j) \;\Big\|\; l \oplus (u \lll a_j) \oplus (l \lll b_j)\Big),$$

where $\lll$ denotes a left rotation on the respective 16-bit half (i.e., rotations are taken modulo 16). The rotation offsets (a_j, b_j, c_j) are listed in Table 1.

Table 1: Rotation offsets of the linear layer in ARADI.

j	a_j	b_j	c_j
0	11	8	14
1	10	9	11
2	9	4	14
3	8	9	7

Key Addition (τ_{k_i}): The 128 bit state is xored ($\oplus$) with the 128 bit subkey in each round as specified above. The subkeys are generated using the following key scheduling algorithm.

Key Schedule

The master key K of ARADI is 256 bits, represented as an array of eight 32-bit words. We denote the key schedule state at round i by

$$K^i = (K_0^i, K_1^i, \ldots, K_7^i).$$

Initially,

$$K^0 = K = (K_0^0, K_1^0, \ldots, K_7^0).$$

The 128-bit pre-whitening key k_0 and the round keys k_i for $i = 1, \ldots, 16$ are derived by concatenating four words:

$$k_i = \begin{cases} K_0^i \parallel K_1^i \parallel K_2^i \parallel K_3^i, & i \bmod 2 = 0, \\ K_4^i \parallel K_5^i \parallel K_6^i \parallel K_7^i, & i \bmod 2 = 1. \end{cases}$$

At each step i, the key state is updated by the following:

Linear Transformations: Two invertible maps M_0 and M_1, each on pairs of words, are defined as

$$M_0(x, y) = \big((x \lll 1) \oplus y,\ (y \lll 3) \oplus (x \lll 1) \oplus y\big),$$
$$M_1(x, y) = \big((x \lll 9) \oplus y,\ (y \lll 28) \oplus (x \lll 9) \oplus y\big).$$

Here, M_0 is applied to (K_0^i, K_1^i) and (K_4^i, K_5^i), while M_1 is applied to (K_2^i, K_3^i) and (K_6^i, K_7^i).

Word Permutation: A permutation P is applied to the words of K^i:

$$P = \begin{cases} (1\,2)(5\,6), & i \bmod 2 = 0, \\ (0\,3)(4\,7), & i \bmod 2 = 1. \end{cases}$$

Round Constant Addition: Finally, the round index i is XORed to K_7^i.

Thus, ARADI's key schedule is an affine transformation, making all round keys affine combinations of the master key bits.

3 Recent Cryptanalysis

This section reviews the state-of-the-art cryptanalysis results on ARADI achieved to date.

1. In the paper [2], the authors present the first independent cryptanalysis of the ARADI block cipher by implementing its bitsliced and S-box variants within the CLAASP framework. Using CLAASP's comprehensive cryptanalysis tools, a detailed preliminary security assessment of ARADI is provided. Avalanche testing demonstrates that a single-bit input difference fully diffuses throughout the state after five rounds, achieving complete bit-avalanche effects in both plaintext and key variables. Statistical analysis utilizing NIST tests confirms that ARADI exhibits strong randomness after five rounds. Continuous diffusion assessment reveals a Continuous Avalanche Factor (CAF) of 0.725 by round seven, increasing to 0.955 by round eight, indicating robust diffusion in half of the cipher's 16 rounds. Differential and linear cryptanalysis identify optimal differential characteristics for up to ten rounds and linear trails up to eight rounds in a single-key context, with probabilities and correlation values reflecting strong resistance. Additionally, a substantial set of impossible differentials has been uncovered for eight rounds, derived from lower-round differentials with minimal Hamming weight

input and output differences. Algebraic analysis through Gröbner bases and division property techniques uncovers integral distinguishers up to seven rounds and reveals a gradual increase in algebraic degree, which helps to understand the structural security of ARADI. Neural network-based distinguishers have also been demonstrated for five rounds in a single-key setting and six rounds in related-key scenarios, providing valuable insight for further analysis. Taken together, these findings offer a rigorous basis for evaluating the security of the ARADI cipher.

2. Previous studies by Bellini et al. [2] and Avanzi et al. [1] did not analyze the structure or unique properties of ARADI [6]. However, in the work [3], the authors deeply examine the ARADI design, uncovering new weaknesses that allow advanced algebraic distinguishers. Here we listed the key findings in the paper [3].

   - **Division Property Modeling:** The research uses mixed integer linear programming to model the three-subset division property, resulting in better distinguishers and lower data requirements than previous methods.
   - **Structural Weakness:** A new property called weakly-composed Toffoli gates is identified, enabling the extension of algebraic distinguishers by one round without increasing data complexity; these rely on key bits and produce consistent cube-sum outputs, a feature not seen before in block ciphers.
   - **Improved Distinguishers:** The new approach achieves effective distinguishers up to 8 rounds, outperforming earlier data complexity requirements.
   - **Design Limitation:** The identified weakness remains regardless of linear layer changes, indicating that only a redesign of the nonlinear layer could prevent it.
   - **Key Recovery:** Using the 8-round distinguisher, a key recovery attack on 10-round ARADI is shown with data complexity of 2^{124} and time complexity of 2^{177}.

3. In the paper [5], the authors employ the Fast Walsh–Hadamard Transform (FHT) technique to improve existing key-recovery approaches on ARADI. Building on the ZeroSum distinguishers introduced by Bellini et al. [3], they extend these distinguishers to successfully attack up to 12 out of 16 rounds in the single-key setting, advancing the state of the art by two rounds compared to prior work.

   The authors' key-recovery attacks using ZeroSum distinguishers rely on three main steps:

   (a) Guessing selected parts of the round key.

   (b) Partially decrypting the corresponding ciphertexts based on these guesses.

   (c) Checking whether the XOR sum of the partially decrypted values is zero on selected bits.

   A significant contribution of the authors' work is showing that partial decryption can be performed across up to four rounds without requiring the full round keys. This is made possible by a critical structural weakness in ARADI: after 3.5 rounds (three full

rounds plus one S-box layer), the cipher does not achieve full diffusion in both forward and backward directions.

The authors highlight a fundamental challenge in cipher design: balancing low-latency linear layers with strong diffusion properties to both enhance performance and maintain security. The low-latency design choices in ARADI, while beneficial for efficiency, introduce exploitable vulnerabilities, underscoring the importance of carefully analyzing linear layer diffusion in lightweight cipher constructions.

Comparative Results

Rounds	Time	Data	Memory (bits)	Reference
10	2^{192}	2^{124}	2^{131}	[3]
10	2^{126}	2^{126}	$2^{18.87}$	[5]
10	$2^{118.6}$	$2^{118.6}$	2^{56}	[5]
11	2^{127}	2^{127}	$2^{47.4}$	[5]
11	$2^{154.4}$	2^{120}	$2^{119.45}$	[5]
12	$2^{157.5}$	$2^{127.9}$	$2^{127.49}$	[5]

Table 2: Comparison of key-recovery attack results on ARADI

The authors of [3] report a time complexity of 2^{177} for their key-recovery attack on 10 rounds of ARADI, which includes a brute-force component estimated at 2^{64}. However, it is important to point out that the brute-force step should actually require 2^{192} encryptions. This discrepancy arises because the ZeroSum distinguisher employed in [3] recovers 2 bits of key information per check, and after performing 32 such checks, 64 bits of the master key are obtained. Consequently, recovering the remaining 192 bits of the key would require an exhaustive search over 2^{192} possible candidates which is reported in [5].

4 Our Work

In our work [4], we study the ARADI cipher and show a practical differential fault attack using a new nibble-based fault model called Permissible Nibble Differences (PNDs). This model offers a good balance between accuracy and practicality. Traditional bit-level fault models need too many faults to be useful, while word-level models make it hard to pinpoint exactly where a fault occurred. Our nibble-based method avoids both problems.

In our attack model, we inject *random nibble faults* rather than single-bit flips. Although faults are applied at an arbitrary nibble position within a chosen word, the analysis still relies on bit-level information because each nibble fault corresponds to a specific 4-bit difference. A nibble fault may be injected into any nibble position inside a chosen word. Because the ARADI is not built around an explicit nibble-aligned internal structure, we must formally define which nibble differences are considered admissible.

Permissible Nibble Difference (PND): A *permissible nibble difference* is a non-zero, truncated difference whose Hamming weight does not exceed 4. The set of permissible differences for a word of size b is

$$\mathcal{D}_b^{\Delta} = \Big\{ (*^4)\, 0^{\,b-4} \gg k \;\Big|\; 0 \le k < b-4,\ *^4 \in \{0,1\}^4 \setminus \{0^4\} \Big\},$$

where "$*^4$" denotes any non-zero 4-bit pattern, $0^{\,b-4}$ is the $(b-4)$-bit zero string, and "$\gg k$" denotes a right rotation/shift by k bits.

For example, when $b = 32$ (as in ARADI), $\mathcal{D}_{32}^{\Delta}$ contains all 32-bit values that consist of a single non-zero nibble placed at an arbitrary nibble-aligned position inside the 32-bit word.

Now suppose a random PND is injected at an arbitrary nibble position within a chosen word at the start of round $r-1$. The propagation of the resulting differences for the affected word z is illustrated in Table 6. A concrete nibble-fault propagation example is shown in Figure 1. Each injected nibble fault produces a characteristic difference pattern after it propagates through two rounds of ARADI's computation. These distinct patterns allow the attacker to detect that a nibble fault occurred and to determine, from the ciphertext difference, which nibble was corrupted much like the analysis carried out for single-bit faults.

4.1 Attack Procedure

The key-recovery attack proceeds as follows (see algorithm 1 for a summary). For each injected *Permissible Nibble Difference (PND)*, the corresponding original and faulty ciphertext pair is collected. Using the known fault-propagation patterns, the affected bits in z^{r-2} are identified. These corrupted bits are then mapped to positions in z^{r-1} that produce *known active differences*, allowing the construction of XOR-difference equations relating observed ciphertext differences to the unknown key bits.

Equations from multiple PNDs are grouped by subkey, with each group corresponding to a 16-bit portion of the last-round key. Each group is solved independently (e.g., using a SAT solver) to produce candidate subkey values. Finally, any remaining uncertainty is resolved via exhaustive search over the small remaining key space, resulting in the full 128-bit key.

Key steps of the attack:

1. Inject f random PNDs into random nibble positions of z^{r-2} and record the resulting original and faulty ciphertext pairs.

2. Examine the difference pattern at (w^r, x^r, y^r, z^r) for each ciphertext pair to determine which bits of z^{r-2} were affected.

3. Map affected bits of z^{r-2} to positions in z^{r-1} with *known active differences*, and construct the corresponding XOR-difference equations linking ciphertext and key bits.

4. Partition all equations into 8 groups, each representing a distinct 16-bit subkey. Solve each group independently to obtain candidate subkey values.

5. Perform an exhaustive search over the remaining possibilities to recover the unique 128-bit key.

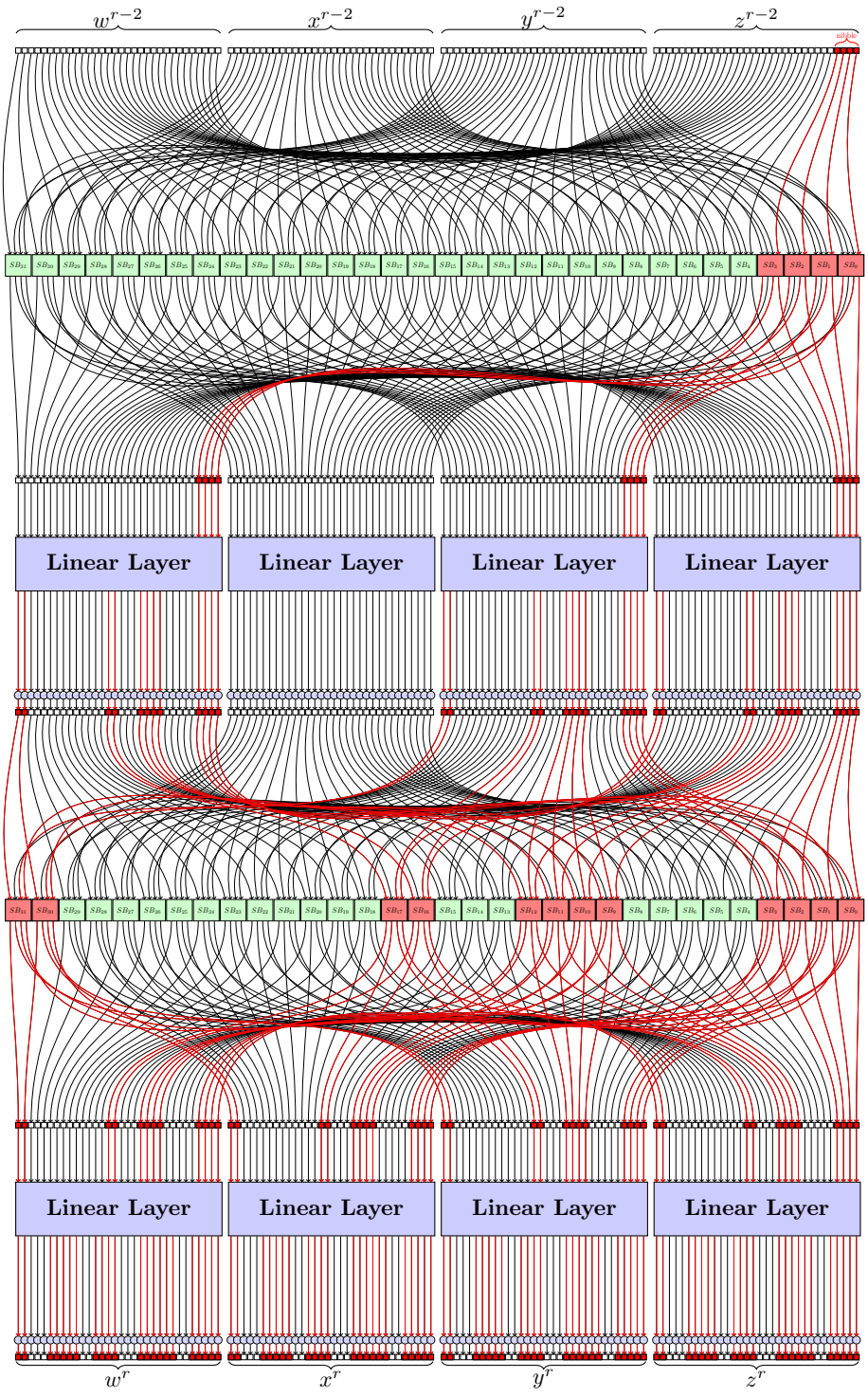

Figure 1: Diagram of Nibble Fault Propagation through the last two rounds of ARADI

Attack Algorithm:

Illustrative Example of the Attack Procedure: To demonstrate the attack

Table 3: Bit-based fault propagation for the fault injected in the word z

Fault Injected bit	Active Bits	
Beginning of 15th round	End of 15th round	End of 16th round ($a \in \{w, x, y, z\}$)
$z^{14}[0]$	$z^{15}[0, 9, 30]$, $y^{15}[0, 9, 30]$, $w^{15}[0, 9, 30]$	$a^{16}[0, 1, 7, 8, 9, 16, 22, 23, 30]$
$z^{14}[1]$	$z^{15}[1, 10, 31]$, $y^{15}[1, 10, 31]$, $w^{15}[1, 10, 31]$	$a^{16}[1, 2, 8, 9, 10, 17, 23, 24, 31]$
$z^{14}[2]$	$z^{15}[2, 11, 16]$, $y^{15}[2, 11, 16]$, $w^{15}[2, 11, 16]$	$a^{16}[2, 3, 9, 10, 11, 16, 18, 24, 25]$
$z^{14}[3]$	$z^{15}[3, 12, 17]$, $y^{15}[3, 12, 17]$, $w^{15}[3, 12, 17]$	$a^{16}[3, 4, 10, 11, 12, 17, 19, 25, 26]$
$z^{14}[4]$	$z^{15}[4, 13, 18]$, $y^{15}[4, 13, 18]$, $w^{15}[4, 13, 18]$	$a^{16}[4, 5, 11, 12, 13, 18, 19, 26, 27]$
$z^{14}[5]$	$z^{15}[5, 14, 19]$, $y^{15}[5, 14, 19]$, $w^{15}[5, 14, 19]$	$a^{16}[5, 6, 12, 13, 14, 19, 21, 27, 28]$
$z^{14}[6]$	$z^{15}[6, 15, 20]$, $y^{15}[6, 15, 20]$, $w^{15}[6, 15, 20]$	$a^{16}[6, 7, 13, 14, 15, 20, 22, 28, 29]$
$z^{14}[7]$	$z^{15}[0, 7, 21]$, $y^{15}[0, 7, 21]$, $w^{15}[0, 7, 21]$	$a^{16}[0, 7, 8, 14, 15, 21, 23, 29, 30]$
$z^{14}[8]$	$z^{15}[1, 8, 22]$, $y^{15}[1, 8, 22]$, $w^{15}[1, 8, 22]$	$a^{16}[0, 1, 8, 9, 15, 22, 24, 30, 31]$
$z^{14}[9]$	$z^{15}[2, 9, 23]$, $y^{15}[2, 9, 23]$, $w^{15}[2, 9, 23]$	$a^{16}[0, 1, 2, 9, 10, 16, 23, 25, 31]$
$z^{14}[10]$	$z^{15}[3, 10, 24]$, $y^{15}[3, 10, 24]$, $w^{15}[3, 10, 24]$	$a^{16}[1, 2, 3, 10, 11, 16, 17, 24, 26]$
$z^{14}[11]$	$z^{15}[4, 11, 25]$, $y^{15}[4, 11, 25]$, $w^{15}[4, 11, 25]$	$a^{16}[2, 3, 4, 11, 12, 17, 18, 25, 27]$
$z^{14}[12]$	$z^{15}[5, 12, 26]$, $y^{15}[5, 12, 26]$, $w^{15}[5, 12, 26]$	$a^{16}[3, 4, 5, 12, 13, 18, 19, 26, 28]$
$z^{14}[13]$	$z^{15}[6, 13, 27]$, $y^{15}[6, 13, 27]$, $w^{15}[6, 13, 27]$	$a^{16}[4, 5, 6, 13, 14, 19, 20, 27, 29]$
$z^{14}[14]$	$z^{15}[7, 14, 28]$, $y^{15}[6, 13, 27]$, $w^{15}[6, 13, 27]$	$a^{16}[5, 6, 7, 14, 15, 20, 21, 28, 30]$
$z^{14}[15]$	$z^{15}[8, 15, 29]$, $y^{15}[6, 13, 27]$, $w^{15}[6, 13, 27]$	$a^{16}[0, 6, 7, 8, 15, 21, 22, 29, 31]$
$z^{14}[16]$	$z^{15}[4, 16, 25]$, $y^{15}[4, 16, 25]$, $w^{15}[4, 16, 25]$	$a^{16}[2, 4, 9, 12, 16, 17, 24, 25, 27]$
$z^{14}[17]$	$z^{15}[5, 17, 26]$, $y^{15}[5, 17, 26]$, $w^{15}[5, 17, 26]$	$a^{16}[3, 5, 10, 13, 17, 18, 25, 26, 28]$
$z^{14}[18]$	$z^{15}[6, 18, 27]$, $y^{15}[6, 18, 27]$, $w^{15}[6, 18, 27]$	$a^{16}[4, 6, 11, 14, 18, 19, 26, 27, 29]$
$z^{14}[19]$	$z^{15}[7, 19, 28]$, $y^{15}[7, 19, 28]$, $w^{15}[7, 19, 28]$	$a^{16}[5, 7, 12, 15, 19, 20, 27, 28, 30]$
$z^{14}[20]$	$z^{15}[8, 20, 29]$, $y^{15}[8, 20, 29]$, $w^{15}[8, 20, 29]$	$a^{16}[0, 6, 8, 13, 20, 21, 28, 29, 31]$
$z^{14}[21]$	$z^{15}[9, 21, 30]$, $y^{15}[9, 21, 30]$, $w^{15}[9, 21, 30]$	$a^{16}[1, 7, 9, 14, 16, 21, 22, 29, 30]$
$z^{14}[22]$	$z^{15}[10, 22, 31]$, $y^{15}[10, 22, 31]$, $w^{15}[10, 22, 31]$	$a^{16}[2, 8, 10, 15, 17, 22, 23, 30, 31]$
$z^{14}[23]$	$z^{15}[11, 16, 23]$, $y^{15}[11, 16, 23]$, $w^{15}[11, 16, 23]$	$a^{16}[0, 3, 9, 11, 16, 18, 23, 24, 31]$
$z^{14}[24]$	$z^{15}[12, 17, 24]$, $y^{15}[12, 17, 24]$, $w^{15}[12, 17, 24]$	$a^{16}[1, 4, 10, 12, 16, 17, 19, 24, 25]$
$z^{14}[25]$	$z^{15}[13, 18, 25]$, $y^{15}[13, 18, 25]$, $w^{15}[13, 18, 25]$	$a^{16}[2, 5, 11, 13, 17, 18, 20, 25, 26]$
$z^{14}[26]$	$z^{15}[14, 19, 26]$, $y^{15}[14, 19, 26]$, $w^{15}[14, 19, 26]$	$a^{16}[3, 6, 12, 14, 18, 19, 21, 26, 27]$
$z^{14}[27]$	$z^{15}[15, 20, 27]$, $y^{15}[15, 20, 27]$, $w^{15}[15, 20, 27]$	$a^{16}[4, 7, 13, 15, 19, 20, 22, 27, 28]$
$z^{14}[28]$	$z^{15}[0, 21, 28]$, $y^{15}[0, 21, 28]$, $w^{15}[0, 21, 28]$	$a^{16}[0, 5, 8, 14, 20, 21, 23, 28, 29]$
$z^{14}[29]$	$z^{15}[1, 22, 29]$, $y^{15}[1, 22, 29]$, $w^{15}[1, 22, 29]$	$a^{16}[1, 6, 9, 15, 21, 22, 24, 29, 30]$
$z^{14}[30]$	$z^{15}[2, 23, 30]$, $y^{15}[2, 23, 30]$, $w^{15}[2, 23, 30]$	$a^{16}[0, 2, 7, 10, 22, 23, 25, 30, 31]$
$z^{14}[31]$	$z^{15}[3, 24, 31]$, $y^{15}[3, 24, 31]$, $w^{15}[3, 24, 31]$	$a^{16}[1, 3, 8, 11, 16, 23, 24, 26, 31]$

Input : Required number of nibble faults f
Output: Recovered 128-bit key
Initialize an empty list (or multimap) $\mathcal{E}$ of equations indexed by bit positions $0, \ldots, 31$;
for $i \leftarrow 1$ **to** f **do**
    Inject a random PND f_i at a uniformly random nibble position in z^{r-2};
    Collect the original and faulty ciphertext pair (C, C');
    Determine the affected bit positions of z^{r-2} using the pattern at (w^r, x^r, y^r, z^r) (see bit-pattern Table 3);
    From the affected z^{r-2} bits, infer which positions j in z^{r-1} yield known active differences (using the bit-pattern Table 3);
    for *each such position* j **do**
        Construct the XOR-difference equation $E_{i,j}$ that relates known ciphertext bits and the unknown key variables;
        Append $E_{i,j}$ to $\mathcal{E}[j]$;
    end
end
Partition the 32 positions into 8 groups (each group covering the 4 positions relevant to a 16-bit subkey);
for *each group* $g = 1, \ldots, 8$ **do**
    Collect all equations from $\mathcal{E}$ that belong to group g;
    Encode the collected equations into the chosen solver format (e.g., SAT);
    Run the solver to obtain candidate assignments for the 16 subkey bits of group g;
    Store the candidate solutions for group g;
end
Combine candidate group-wise solutions and perform exhaustive search over any remaining undecided bits to find the unique 128-bit key;
return *Recovered 128-bit key*

Algorithm 1: Recovering the 128-bit Key

procedure, consider the case where the adversary injects ten random Permissible Nibble Differences (PNDs) into z^{14}. For each injection we collect the original and faulty ciphertext pair, inspect the observed difference pattern at $(w^{16}, x^{16}, y^{16}, z^{16})$, and consult Table 3 to identify which bit positions of z^{15} become *known* (active) as a result. The ten injected PNDs (order not important here) are

{`0x0000000f`, `0x00000700`, `0x00300000`, `0x00e00000`, `0x00001000`,
`0x00900000`, `0x00a00000`, `0x00078000`, `0x0c000000`, `0x3c000000`}

Each PND affects up to four bit positions of z^{14}, which in turn map to specific red (active)

Table 4: Frequency of $z^{15}[i]$ in the list

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
count	2	3	2	2	1	2	1	0	4	5	3	5	2	0	2	3
i	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
count	5	2	1	2	4	4	3	4	1	1	4	3	1	4	4	2

Table 5: Total number of equations corresponding to each group

Group	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Total Equations	12	14	8	12	8	7	11	10

entries in the z^{15} row of Table 3. The PND-to-z^{14} bit mapping is

$$\begin{cases} \{\texttt{0x0000000f} \to \{3,2,1,0\}\}, \\ \{\texttt{0x00000700} \to \{10,9,8\}\}, \\ \{\texttt{0x00300000} \to \{21,20\}\}, \\ \{\texttt{0x00e00000} \to \{23,22,21\}\}, \\ \{\texttt{0x00001000} \to \{12\}\}, \\ \{\texttt{0x00900000} \to \{23,20\}\}, \\ \{\texttt{0x00a00000} \to \{23,21\}\}, \\ \{\texttt{0x00078000} \to \{18,17,16,15\}\}, \\ \{\texttt{0x0c000000} \to \{27,26\}\}, \\ \{\texttt{0x3c000000} \to \{29,28,27,26\}\} \end{cases}$$

Using the propagation rules in Table 3, each of the above PNDs yields a set of active indices in z^{15}. The full active-bit mappings for these ten PNDs are respectively

$$\begin{cases} \{z^{15}[0,9,30],\ z^{15}[1,10,31],\ z^{15}[2,11,16],\ z^{15}[3,12,17]\}, \\ \{z^{15}[8,22,1],\ z^{15}[9,23,2],\ z^{15}[10,24,3]\}, \\ \{z^{15}[20,29,8],\ z^{15}[21,30,9]\}, \\ \{z^{15}[21,30,9],\ z^{15}[22,31,10],\ z^{15}[23,16,11]\}, \\ \{z^{15}[12,26,5]\}, \\ \{z^{15}[20,29,8],\ z^{15}[23,16,11]\}, \\ \{z^{15}[21,30,9],\ z^{15}[23,16,11]\}, \\ \{z^{15}[8,15,29],\ z^{15}[16,25,4],\ z^{15}[17,26,5],\ z^{15}[18,27,6]\}, \\ \{z^{15}[14,19,26],\ z^{15}[15,20,27]\}, \\ \{z^{15}[14,19,26],\ z^{15}[15,20,27],\ z^{15}[28,21,0],\ z^{15}[29,22,1]\} \end{cases}$$

Each listed z^{15} index above denotes a *known* bit position produced by the corresponding PND; these known positions are then translated into XOR-difference equations that relate ciphertext bits to unknown key variables.

Table 4 summarizes how many times each $z^{15}[i]$ became known across the ten injections. These frequencies determine how many independent difference equations are available at each bit position and are used to assemble group-wise equation sets. The total number of equations obtained for each group (from the ten PNDs) is shown in Table 5.

Finally, for each group, we submit the corresponding set of difference equations to a SAT solver to obtain candidate assignments for the 16-bit subkey associated with that group; the candidate solutions from all groups are combined, and any remaining key bits are resolved by exhaustive search to recover the unique 128-bit key.

Complexity: In our attack, the attacker introduces random nibble faults at the start of round 15 into a selected 32-bit word, without knowing which nibble was affected or what fault value was used. By analyzing how these faults change the ciphertexts, and using ARADI's known fault propagation behavior, we can form a set of equations that link the observed differences to the unknown key bits.

We group these equations according to the subkeys and show that:

- 27 equations can reduce the possible values of a 16-bit subkey from 2^{16} to 2^4
- With 36 equations, the keyspace reduces further to 2^2.
- 45 equations are enough to find a unique subkey.

This procedure is repeated across all eight groups, and we observed consistent results on average. To balance the tradeoff between time complexity and number of faults, we tolerate a slightly larger time complexity in order to minimize fault injections. Thus, we choose 27 equations per group, leading to a time complexity of $(2^4)^8 = 2^{32}$, which is practical.

By repeating this process for all eight group 16-bit subkeys, the full 128-bit last-round subkey can be recovered with at most 36 random PND injections. Extending the method allows full recovery of the 256-bit master key with at most 108 random faults.

The total time complexity of the attack is about 3×2^{32}, showing that the attack is practical for the lightweight ARADI cipher. Overall, this work introduces the first differential fault analysis of ARADI, using a realistic, low-cost attack model with clear resource estimates, and reveals previously unexplored weaknesses in the cipher.

This work was carried out during my visiting scientist period. The full paper has been submitted to a journal, and a preprint is also available[4].

References

[1] Roberto Avanzi, Orr Dunkelman, and Shibam Ghosh. A note on aradi and llama. *Cryptology ePrint Archive*, 2024.

[2] Emanuele Bellini, Mattia Formenti, David Gérault, Juan Grados, Anna Hambitzer, Yun Ju Huang, Paul Huynh, Mohamed Rachidi, Raghvendra Rohit, and Sharwan K. Tiwari. Claasping ARADI: automated analysis of the ARADI block cipher. In Sourav Mukhopadhyay and Pantelimon Stanica, editors, *Progress in Cryptology - INDOCRYPT 2024 - 25th International Conference on Cryptology in India, Chennai, India, December 18-21, 2024, Proceedings, Part II*, volume 15496 of *Lecture Notes in Computer Science*, pages 90–113. Springer, 2024.

[3] Emanuele Bellini, Mohamed Rachidi, Raghvendra Rohit, and Sharwan K. Tiwari. On the structural properties of toffoli gate composition in ARADI: implications for algebraic distinguishers. In Marc Fischlin and Veelasha Moonsamy, editors, *Applied Cryptography and Network Security - 23rd International Conference, ACNS 2025, Munich, Germany, June 23-26, 2025, Proceedings, Part II*, volume 15826 of *Lecture Notes in Computer Science*, pages 400–425. Springer, 2025.

[4] Chandan Dey, Soumya Sahoo, and Santanu Sarkar. Evaluating the resistance of ARADI against differential fault attack. Cryptology ePrint Archive, Paper 2025/1977, 2025.

[5] Orr Dunkelman and Shibam Ghosh. Improved key-recovery attacks on aradi. *Cryptology ePrint Archive*, 2025.

[6] Patricia Greene, Mark Motley, and Bryan Weeks. ARADI and LLAMA: Low-latency cryptography for memory encryption. Cryptology ePrint Archive, Paper 2024/1240, 2024.

[7] Donggeun Kwon and Seokhie Hong. Side-channel attack on aradi in non-profiling scenarios. *IEEE Access*, 13:122628–122635, 2025.

Implementation of ARADI block cipher in C language

Here we provide the C implementation of ARADI block cipher and validate with the test vectors given in the paper [6].

```c
#include <stdint.h>
#include <stdio.h>
#include <string.h>

#define ROTL32(x, n) (((x) << (n)) | ((x) >> (32 - (n))))
#define ROTL16(x, n) (((x) << (n)) | ((x) >> (16 - (n))))


typedef struct {
    uint32_t k[17][4];
} KeySchedule;

//Linear transformation used in the key scheduling algorithm
void M(uint8_t i, uint8_t j, uint32_t* X, uint32_t* Y) {
    uint32_t x = *X;
    uint32_t y = *Y;
    uint32_t t1 = ROTL32(y, i) ^ ROTL32(x, j) ^ x;
    uint32_t t2 = ROTL32(y, i) ^ x;
    *X = t1;
    *Y = t2;
}

// Linear layer used in the round function
uint32_t L(uint8_t a, uint8_t b, uint8_t c, uint32_t input) {
    uint16_t x1, x = (input >> 16) & 0xFFFF;
    uint16_t y = input & 0xFFFF;
    x1 = x ^ ROTL16(x, a) ^ ROTL16(y, c);
    y = y ^ ROTL16(y, a) ^ ROTL16(x, b);
    return ((uint32_t)x1 << 16) | y;
}

//Key scheduling algorithm
void key_expansion(uint32_t* K, KeySchedule* ks) {
    for (int i = 0; i < 16; i++) {
        int j = i % 2;
        for (int m = 0; m < 4; m++) ks->k[i][m] = K[4 * j + m];

        M(1, 3, &K[1], &K[0]);
        M(9, 28, &K[3], &K[2]);
        M(1, 3, &K[5], &K[4]);
        M(9, 28, &K[7], &K[6]);
        K[7] ^= i;

        if (j == 0) {
            uint32_t T = K[1]; K[1] = K[2]; K[2] = T;
            T = K[5]; K[5] = K[6]; K[6] = T;
        } else {
            uint32_t T = K[1]; K[1] = K[4]; K[4] = T;
            T = K[3]; K[3] = K[6]; K[6] = T;
```

```
        }
    }
    for (int m = 0; m < 4; m++) ks->k[16][m] = K[m];
}

void encrypt(uint32_t* wxyz, KeySchedule* ks) {
    uint32_t w = wxyz[0];
    uint32_t x = wxyz[1];
    uint32_t y = wxyz[2];
    uint32_t z = wxyz[3];

    uint8_t a[4] = {11,10,9,8};
    uint8_t b[4] = {8,9,4,9};
    uint8_t c[4] = {14,11,14,7};

    for (int i = 0; i < 16; i++) {
        // Key addition
        z ^= ks->k[i][3];
        y ^= ks->k[i][2];
        x ^= ks->k[i][1];
        w ^= ks->k[i][0];

        // Sbox or non linear layer
        x ^= (w & y);
        z ^= (x & y);
        y ^= (w & z);
        w ^= (x & z);

        // Linear layer
        int j = i % 4;
        z = L(a[j], b[j], c[j], z);
        y = L(a[j], b[j], c[j], y);
        x = L(a[j], b[j], c[j], x);
        w = L(a[j], b[j], c[j], w);

        printf("Round %d subcipher: %08x %08x %08x %08x\n", i, w, x, y, z)
            ;
    }

    // Last key addition
    z ^= ks->k[16][3];
    y ^= ks->k[16][2];
    x ^= ks->k[16][1];
    w ^= ks->k[16][0];

    wxyz[0] = w;
    wxyz[1] = x;
    wxyz[2] = y;
    wxyz[3] = z;
}

int main() {
    uint32_t key[8] = {
    0x03020100,
```

```
    0x07060504,
    0x0b0a0908,
    0x0f0e0d0c,
    0x13121110,
    0x17161514,
    0x1b1a1918,
    0x1f1e1d1c
    };
    uint32_t data[4] = {0x00000000, 0x00000000, 0x00000000, 0x00000000};

    KeySchedule ks;
    key_expansion(key, &ks);
    encrypt(data, &ks);

    printf("\n\n");
    for (int i = 0; i < 17; i++) {
        printf("Subkey %d: %08x %08x %08x %08x\n",
               i, ks.k[i][0], ks.k[i][1], ks.k[i][2], ks.k[i][3]);
    }

    printf("\n");
    printf("Ciphertext: %08x %08x %08x %08x\n", data[0], data[1], data[2],
        data[3]);
    return 0;
}
```

Listing 1: ARADI cipher Implementation in C

Table 6: Nibble-based fault propagation pattern for the fault injected in the word z

Nibble Fault	Active Bits	
Beginning of 15th round	End of 15th round	End of 16th round ($a \in \{w, x, y, z\}$)
0x0000000f	z^{15}[0, 1, 2, 3, 9, 10, 11, 12, 16, 17, 30, 31], y^{15}[0, 1, 2, 3, 9, 10, 11, 12, 16, 17, 30, 31], w^{15}[0, 1, 2, 3, 9, 10, 11, 12, 16, 17, 30, 31]	[0, 1, 2, 3, 4, 7, 8, 9, 10, 11, 12, 16, 17, 18, 19, 22, 23, 24, 25, 26, 30, 31]
0x0000001e	z^{15}[1, 2, 3, 4, 10, 11, 12, 13, 16, 17, 18, 31], y^{15}[1, 2, 3, 4, 10, 11, 12, 13, 16, 17, 18, 31], w^{15}[1, 2, 3, 4, 10, 11, 12, 13, 16, 17, 18, 31]	[1, 2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 16, 17, 18, 19, 20, 23, 24, 25, 26, 27, 31]
0x0000003c	z^{15}[2, 3, 4, 5, 11, 12, 13, 14, 16, 17, 18, 19], y^{15}[2, 3, 4, 5, 11, 12, 13, 14, 16, 17, 18, 19], w^{15}[2, 3, 4, 5, 11, 12, 13, 14, 16, 17, 18, 19]	[2, 3, 4, 5, 6, 9, 10, 11, 12, 13, 14, 16, 17, 18, 19, 20, 21, 24, 25, 26, 27, 28]
0x00000078	z^{15}[3, 4, 5, 6, 12, 13, 14, 15, 17, 18, 19, 20], y^{15}[3, 4, 5, 6, 12, 13, 14, 15, 17, 18, 19, 20], w^{15}[3, 4, 5, 6, 12, 13, 14, 15, 17, 18, 19, 20]	[3, 4, 5, 6, 7, 10, 11, 12, 13, 14, 15, 17, 18, 19, 20, 21, 22, 25, 26, 27, 28, 29]
0x000000f0	z^{15}[0, 4, 5, 6, 7, 13, 14, 15, 18, 19, 20, 21], y^{15}[0, 4, 5, 6, 7, 13, 14, 15, 18, 19, 20, 21], w^{15}[0, 4, 5, 6, 7, 13, 14, 15, 18, 19, 20, 21]	[0, 4, 5, 6, 7, 8, 11, 12, 13, 14, 15, 18, 19, 20, 21, 22, 23, 26, 27, 28, 29, 30]
0x000001e0	z^{15}[0, 1, 5, 6, 7, 8, 14, 15, 19, 20, 21, 22], y^{15}[0, 1, 5, 6, 7, 8, 14, 15, 19, 20, 21, 22], w^{15}[0, 1, 5, 6, 7, 8, 14, 15, 19, 20, 21, 22]	[0, 1, 5, 6, 7, 8, 9, 12, 13, 14, 15, 19, 20, 21, 22, 23, 24, 27, 28, 29, 30, 31]
0x000003c0	z^{15}[0, 1, 2, 6, 7, 8, 9, 15, 20, 21, 22, 23], y^{15}[0, 1, 2, 6, 7, 8, 9, 15, 20, 21, 22, 23], w^{15}[0, 1, 2, 6, 7, 8, 9, 15, 20, 21, 22, 23]	[0, 1, 2, 6, 7, 8, 9, 10, 13, 14, 15, 16, 20, 21, 22, 23, 24, 25, 28, 29, 30, 31]
0x00000780	z^{15}[0, 1, 2, 3, 7, 8, 9, 10, 21, 22, 23, 24], y^{15}[0, 1, 2, 3, 7, 8, 9, 10, 21, 22, 23, 24], w^{15}[0, 1, 2, 3, 7, 8, 9, 10, 21, 22, 23, 24]	[0, 1, 2, 3, 7, 8, 9, 10, 11, 14, 15, 16, 17, 21, 22, 23, 24, 25, 26, 29, 30, 31]
0x00000f00	z^{15}[1, 2, 3, 4, 8, 9, 10, 11, 22, 23, 24, 25], y^{15}[1, 2, 3, 4, 8, 9, 10, 11, 22, 23, 24, 25], w^{15}[1, 2, 3, 4, 8, 9, 10, 11, 22, 23, 24, 25]	[0, 1, 2, 3, 4, 8, 9, 10, 11, 12, 15, 16, 17, 18, 22, 23, 24, 25, 26, 27, 30, 31]
0x00001e00	z^{15}[2, 3, 4, 5, 9, 10, 11, 12, 23, 24, 25, 26], y^{15}[2, 3, 4, 5, 9, 10, 11, 12, 23, 24, 25, 26], w^{15}[2, 3, 4, 5, 9, 10, 11, 12, 23, 24, 25, 26]	[0, 1, 2, 3, 4, 5, 9, 10, 11, 12, 13, 16, 17, 18, 19, 23, 24, 25, 26, 27, 28, 31]
0x00003c00	z^{15}[3, 4, 5, 6, 10, 11, 12, 13, 24, 25, 26, 27], y^{15}[3, 4, 5, 6, 10, 11, 12, 13, 24, 25, 26, 27], w^{15}[3, 4, 5, 6, 10, 11, 12, 13, 24, 25, 26, 27]	[1, 2, 3, 4, 5, 6, 10, 11, 12, 13, 14, 16, 17, 18, 19, 20, 24, 25, 26, 27, 28, 29]
0x00007800	z^{15}[4, 5, 6, 7, 11, 12, 13, 14, 25, 26, 27, 28], y^{15}[4, 5, 6, 7, 11, 12, 13, 14, 25, 26, 27, 28], w^{15}[4, 5, 6, 7, 11, 12, 13, 14, 25, 26, 27, 28]	[2, 3, 4, 5, 6, 7, 11, 12, 13, 14, 15, 17, 18, 19, 20, 21, 25, 26, 27, 28, 29, 30]
0x0000f000	z^{15}[5, 6, 7, 8, 12, 13, 14, 15, 26, 27, 28, 29], y^{15}[5, 6, 7, 8, 12, 13, 14, 15, 26, 27, 28, 29], w^{15}[5, 6, 7, 8, 12, 13, 14, 15, 26, 27, 28, 29]	[0, 3, 4, 5, 6, 7, 8, 12, 13, 14, 15, 18, 19, 20, 21, 22, 26, 27, 28, 29, 30, 31]
0x0001e000	z^{15}[4, 6, 7, 8, 13, 14, 15, 16, 25, 27, 28, 29], y^{15}[4, 6, 7, 8, 13, 14, 15, 16, 25, 27, 28, 29], w^{15}[4, 6, 7, 8, 13, 14, 15, 16, 25, 27, 28, 29]	[0, 2, 4, 5, 6, 7, 8, 9, 12, 13, 14, 15, 16, 17, 19, 20, 21, 22, 24, 25, 27, 28, 29, 30, 31]
0x0003c000	z^{15}[4, 5, 7, 8, 14, 15, 16, 17, 25, 26, 28, 29], y^{15}[4, 5, 7, 8, 14, 15, 16, 17, 25, 26, 28, 29], w^{15}[4, 5, 7, 8, 14, 15, 16, 17, 25, 26, 28, 29]	[0, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 13, 14, 15, 16, 17, 18, 20, 21, 22, 24, 25, 26, 27, 28, 29, 30, 3]
0x00078000	z^{15}[4, 5, 6, 8, 15, 16, 17, 18, 25, 26, 27, 29], y^{15}[4, 5, 6, 8, 15, 16, 17, 18, 25, 26, 27, 29], w^{15}[4, 5, 6, 8, 15, 16, 17, 18, 25, 26, 27, 29]	[0, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 21, 22, 24, 25, 26, 27, 28, 29, 31]
0x000f0000	z^{15}[4, 5, 6, 7, 16, 17, 18, 19, 25, 26, 27, 28], y^{15}[4, 5, 6, 7, 16, 17, 18, 19, 25, 26, 27, 28], w^{15}[4, 5, 6, 7, 16, 17, 18, 19, 25, 26, 27, 28]	[2, 3, 4, 5, 6, 7, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 24, 25, 26, 27, 28, 29, 30]
0x001e0000	z^{15}[5, 6, 7, 8, 17, 18, 19, 20, 26, 27, 28, 29], y^{15}[5, 6, 7, 8, 17, 18, 19, 20, 26, 27, 28, 29], w^{15}[5, 6, 7, 8, 17, 18, 19, 20, 26, 27, 28, 29]	[0, 3, 4, 5, 6, 7, 8, 10, 11, 12, 13, 14, 15, 17, 18, 19, 20, 21, 25, 26, 27, 28, 29, 30, 31]
0x003c0000	z^{15}[6, 7, 8, 9, 18, 19, 20, 21, 27, 28, 29, 30], y^{15}[6, 7, 8, 9, 18, 19, 20, 21, 27, 28, 29, 30], w^{15}[6, 7, 8, 9, 18, 19, 20, 21, 27, 28, 29, 30]	[0, 1, 4, 5, 6, 7, 8, 9, 11, 12, 13, 14, 15, 16, 18, 19, 20, 21, 22, 26, 27, 28, 29, 30, 31]
0x00780000	z^{15}[7, 8, 9, 10, 19, 20, 21, 22, 28, 29, 30, 31], y^{15}[7, 8, 9, 10, 19, 20, 21, 22, 28, 29, 30, 31], w^{15}[7, 8, 9, 10, 19, 20, 21, 22, 28, 29, 30, 31]	[0, 1, 2, 5, 6, 7, 8, 9, 10, 12, 13, 14, 15, 16, 17, 19, 20, 21, 22, 23, 27, 28, 29, 30, 31]
0x00f00000	z^{15}[8, 9, 10, 11, 16, 20, 21, 22, 23, 29, 30, 31], y^{15}[8, 9, 10, 11, 16, 20, 21, 22, 23, 29, 30, 31], w^{15}[8, 9, 10, 11, 16, 20, 21, 22, 23, 29, 30, 31]	[0, 1, 2, 3, 6, 7, 8, 9, 10, 11, 13, 14, 15, 16, 17, 18, 20, 21, 22, 23, 24, 28, 29, 30, 31]
0x01e00000	z^{15}[9, 10, 11, 12, 16, 17, 21, 22, 23, 24, 30, 31], y^{15}[9, 10, 11, 12, 16, 17, 21, 22, 23, 24, 30, 31], w^{15}[9, 10, 11, 12, 16, 17, 21, 22, 23, 24, 30, 31]	[0, 1, 2, 3, 4, 7, 8, 9, 10, 11, 12, 14, 15, 16, 17, 18, 19, 21, 22, 23, 24, 25, 29, 30, 31]
0x03c00000	z^{15}[10, 11, 12, 13, 16, 17, 18, 22, 23, 24, 25, 31], y^{15}[10, 11, 12, 13, 16, 17, 18, 22, 23, 24, 25, 31], w^{15}[10, 11, 12, 13, 16, 17, 18, 22, 23, 24, 25, 31]	[0, 1, 2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 15, 16, 17, 18, 19, 20, 22, 23, 24, 25, 26, 30, 31]
0x07800000	z^{15}[11, 12, 13, 14, 16, 17, 18, 19, 23, 24, 25, 26], y^{15}[11, 12, 13, 14, 16, 17, 18, 19, 23, 24, 25, 26], w^{15}[11, 12, 13, 14, 16, 17, 18, 19, 23, 24, 25, 26]	0, 1, 2, 3, 4, 5, 6, 9, 10, 11, 12, 13, 14, 16, 17, 18, 19, 20, 21, 23, 24, 25, 26, 27, 31
0x0f000000	z^{15}[12, 13, 14, 15, 17, 18, 19, 20, 24, 25, 26, 27], y^{15}[12, 13, 14, 15, 17, 18, 19, 20, 24, 25, 26, 27], w^{15}[12, 13, 14, 15, 17, 18, 19, 20, 24, 25, 26, 27],	[1, 2, 3, 4, 5, 6, 7, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 24, 25, 26, 27, 28]
0x1e000000	z^{15}[0, 13, 14, 15, 18, 19, 20, 21, 25, 26, 27, 28], y^{15}[0, 13, 14, 15, 18, 19, 20, 21, 25, 26, 27, 28], w^{15}[0, 13, 14, 15, 18, 19, 20, 21, 25, 26, 27, 28]	[0, 2, 3, 4, 5, 6, 7, 8, 11, 12, 13, 14, 15, 17, 18, 19, 20, 21, 22, 23, 25, 26, 27, 28, 29]
0x3c000000	z^{15}[0, 1, 14, 15, 19, 20, 21, 22, 26, 27, 28, 29], y^{15}[0, 1, 14, 15, 19, 20, 21, 22, 26, 27, 28, 29], w^{15}[0, 1, 14, 15, 19, 20, 21, 22, 26, 27, 28, 29]	[0, 1, 3, 4, 5, 6, 7, 8, 9, 12, 13, 14, 15, 18, 19, 20, 21, 22, 23, 24, 26, 27, 28, 29, 30]
0x78000000	z^{15}[0, 1, 2, 15, 20, 21, 22, 23, 27, 28, 29, 30], y^{15}[0, 1, 2, 15, 20, 21, 22, 23, 27, 28, 29, 30], w^{15}[0, 1, 2, 15, 20, 21, 22, 23, 27, 28, 29, 30]	[0, 1, 2, 4, 5, 6, 7, 8, 9, 10, 13, 14, 15, 19, 20, 21, 22, 23, 24, 25, 27, 28, 29, 30, 31]
0xf0000000	z^{15}[0, 1, 2, 3, 21, 22, 23, 24, 28, 29, 30, 31], y^{15}[0, 1, 2, 3, 21, 22, 23, 24, 28, 29, 30, 31], w^{15}[0, 1, 2, 3, 21, 22, 23, 24, 28, 29, 30, 31]	[0, 1, 2, 3, 5, 6, 7, 8, 9, 10, 11, 14, 15, 16, 20, 21, 22, 23, 24, 25, 26, 28, 29, 30, 31]