Network Working Group
Internet-Draft
Intended status: Informational
Expires: 20 August 2026

Y. Sakemi, Ed.
S. Kanno
GMO CONNECT Inc.
T. Isobe
The University of Osaka

16 February 2026

Areion: Highly-Efficient Permutations and Its Applications
draft-sakemi-areion-01

Abstract

This document specifies a series of cryptographic wide-block permutations referred to as Areion-256 and Areion-512. These permutations are constructed using AES round operations and are designed for ultra-low latency implementations on modern processors with AES instructions. The Areion permutations can be used as building blocks in various cryptographic constructions, including authenticated encryption and hashing of relatively short input data. Additionally, it describes AEAD schemes and hash functions constructed from Areion.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 20 August 2026.

Copyright Notice

Table of Contents

1. Introduction

The recent evolution of communication technologies demands unprecedented performance in data processing, especially in networking and secure data transmission. To address these demands, cryptographic primitives must be designed to minimize latency while maintaining strong security guarantees. In particular, data-centric architectures such as Content Delivery Networks (CDNs), IoT devices, and 6G networks require cryptographic primitives that can offer both robust security and high efficiency. Wide-block ciphers, which operate on larger block sizes than traditional block ciphers, have emerged as a promising solution to address the limitations of 128-bit block sizes, such as those in AES-GCM. Notably, concerns regarding the birthday bound security limitation of 64-bit data security in AES-GCM have driven research into cryptographic primitives that can provide beyond-birthday-bound security while maintaining competitive performance.

As shown in [SP800-38A], modes of operation for block ciphers often have inherent limitations in terms of security bounds due to the fixed block size. For instance, AES-GCM provides a data security bound limited to approximately 2^64 encrypted blocks when using a 128-bit block cipher like AES. This limitation can be problematic in high-throughput applications where large amounts of data are encrypted under a single key. Therefore, there is a growing need for cryptographic schemes that can offer higher data security bounds, such as 2^128, while also being efficient in practice. Moreover, recent discussions, such as those in the public comments on NIST SP800-38A [PublicCommentOnSP800-38A], have emphasized the need for re-evaluating block cipher modes and considering alternatives that can provide stronger security guarantees and better performance characteristics for modern applications.

This need for short-message optimization is driven by real-world data, as detailed in [Areion]. In communication environments, the overwhelming majority of cases require encryption or hashing of short inputs (up to 2K bytes). For example, 44% of "real-world" TCP/IP packets are between 40 and 100 bytes long. Furthermore, the maximum packet lengths for protocols like Zigbee (127 bytes) and Bluetooth Low Energy (47 bytes) are less than 128 bytes. Existing schemes are often not optimized for these common short inputs.

Areion is a novel cryptographic primitive designed to meet these demands by providing highly efficient wide-block permutations based on AES round operations. By using modern CPU instructions for AES and SIMD (Single Instruction Multiple Data) operations, Areion achieves ultra-low latency encryption and decryption while maintaining strong security guarantees. The Areion permutations, Areion-256 and Areion-512, operate on 256-bit and 512-bit blocks respectively, providing a flexible and scalable foundation for constructing secure cryptographic schemes.

This document focuses on the design and specification of the Areion permutation algorithm, detailing the construction of Areion-256 and Areion-512. The Areion permutations are designed to be efficient when implemented using AES instructions and SIMD capabilities on modern processors, making them suitable for high-performance cryptographic applications. The design leverages AES round operations, including SubBytes (SB), ShiftRows (SR), MixColumns (MC), and AddRoundConstant (AC), to construct a secure and efficient permutation suitable for various cryptographic constructions.

The design and analysis of Areion have been studied in detail in [Areion]; this document provides an implementation-oriented specification suitable for Internet deployment.

Areion’s design is deeply influenced by the AES instruction set and modern processor architectures, allowing it to achieve high throughput and low latency. By carefully selecting the number of rounds, round constants, and the structure of the permutation, Areion aims to provide a secure and efficient permutation that can serve as a building block for authenticated encryption, hashing, and other cryptographic primitives. This document specifies the full algorithmic details necessary to implement the Areion permutations and their use in an AEAD scheme based on the OPP (Offset Public Permutation) mode.

In this document, we specify a detailed specification of permutation on Areion.

Note: While Areion can be applied for hashing, this document focuses on the permutation, its use in authenticated encryption, and hash functions built from the permutation.

1.1. Conventions Used in This Document

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

2. Design Rationale

Areion’s design is based on several key principles derived from the extended analysis in [Areion], balancing high performance on modern CPUs with robust security margins.

2.1. Maximal Use of AES Instructions

Areion is designed to be implemented solely using AES instructions, such as aesenc in Intel AES-NI or vaeseq and vaesmcq in ARMv8 NEON. These instructions are among the most efficient and cryptographically strong operations available in modern SIMD instruction sets. This approach avoids slower shuffle operations and leverages the deep security analysis of the AES round function.

2.2. Pipeline-Friendly Feistel Structure

Modern processors can execute multiple AES instructions in parallel through pipelining (e.g., Intel Ice Lake can pipeline up to 6 aesenc instructions). Areion adopts a "pipeline-friendly" Feistel-type structure, which, unlike traditional Feistel schemes or Simpira v2, adds F-functions in a way that takes full advantage of this hardware parallelism. This structure allows for more AES instructions to be executed in parallel within a single round, significantly reducing latency.

2.3. Systematic Search for Optimal Functions

The specific F-functions used in Areion-256 (F_1, F_2) and Areion-512 (F_0, F_1, F_3) were not chosen arbitrarily. They are the result of a systematic search over all possible combinations of 1- and 2-round AES operations. The chosen structures, (2, 1)-perm for Areion-256 and (0, 1, 0, 3, pi_1)-perm for Areion-512, were identified as providing the best trade-off between the lowest number of AES instructions required and the highest security level achieved against differential, linear, impossible differential, and integral attacks.

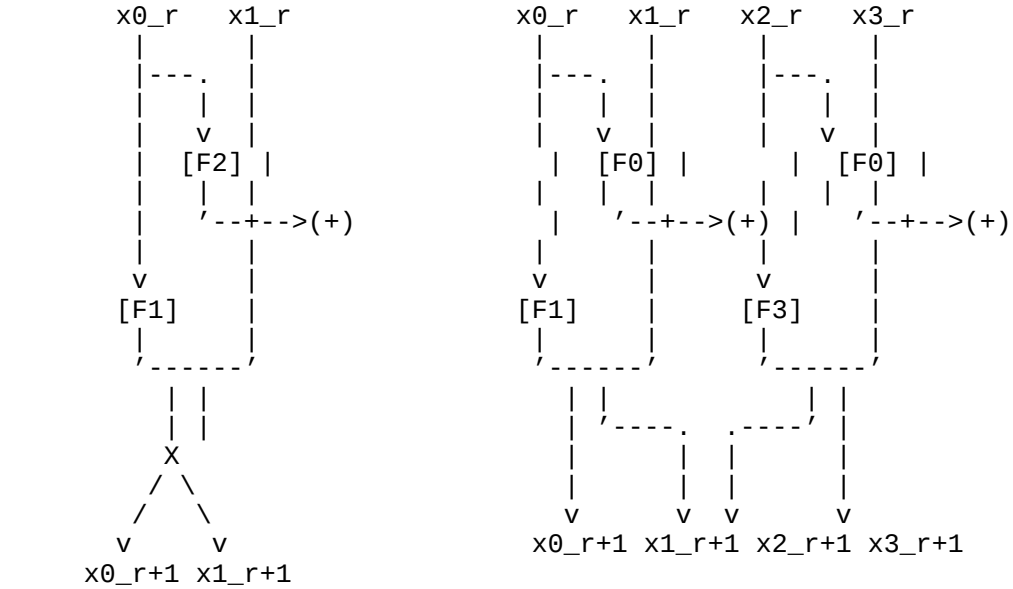
2.4. Round Counts and Security Margins

The number of rounds, 10 for Areion-256 and 15 for Areion-512, was determined by detailed security analysis. This analysis (detailed in Section 6) identified the longest possible attacks (e.g., 5-round zero-sum for Areion-256, 10-round zero-sum for Areion-512) and established the full round counts to provide a sufficient security margin against all known cryptanalytic techniques.

3. Design of Areion Permutation

The Areion permutation algorithm is designed to provide ultra-low latency cryptographic operations while maintaining strong security properties. The design leverages AES round operations and modern CPU instruction sets to construct efficient 256-bit and 512-bit permutations. These permutations serve as the core components for various cryptographic modes, including authenticated encryption and hashing. This section outlines the notations and the structure of the Areion permutation algorithm for both Areion-256 and Areion-512.

The round functions for Areion-256 and Areion-512 are illustrated in Figure 1.



(a) Areion-256 (b) Areion-512

Figure 1: The round functions of Areion [Areion]

3.1. Notations

SB: SubBytes

SR: ShiftRows

MC: MixColumns

AC: AddRoundConstant operations of the AES round function. This operation adds a round constant to the state, similar to the AddRoundKey operation in AES, but instead of a round key, a constant is added.

^: Bitwise XOR operation

: Function composition, where the function on the right is applied first

3.2. Functions

Based on the operations in the AES round function, we define four 128-bit functions F_i for i in {0, 1, 2, 3}. Each function maps a 128-bit state to a 128-bit state. The AES round operations SubBytes, ShiftRows, MixColumns, and AddRoundConstant are denoted by SB, SR, MC, and AC, respectively. AC(x, C) denotes the bitwise XOR of a 128-bit constant C to the 128-bit state x.

* F_0(x) = MC SR SB(x)

* F_1(x) = SR SB(x)

* For a given round index r, F_2^{(r)}(x) = MC SR SB AC(MC SR SB(x), RC_r)

* For a given round index r, F_3^{(r)}(x) = MC SR SB AC(SR SB(x), RC_r)

The constants RC_r used in AC(, RC_r) are the round constants defined in Section 3.3. F_0 and F_1 do not use round constants.

3.3. Round Constants

Areion uses 128-bit round constants RC_r for r = 0, 1, ..., 14. The constants are derived from the binary expansion of the fractional part of π and are given in hexadecimal notation in Table 1. Each constant is used in little-endian byte order when applied by AC(x, RC_r).

r	RC_r (hexadecimal)
0	0x243f6a8885a308d313198a2e03707344
1	0xa4093822299f31d0082efa98ec4e6c89
2	0x452821e638d01377be5466cf34e90c6c
3	0xc0ac29b7c97c50dd3f84d5b5b5470917
4	0x9216d5d98979fb1bd1310ba698dfb5ac
5	0x2ffd72dbd01adfb7b8e1afed6a267e96
6	0xba7c9045f12c7f9924a19947b3916cf7
7	0x801f2e2858efc16636920d871574e690
8	0xa458fea3f4933d7e0d95748f728eb658
9	0x718bcd5882154aee7b54a41dc25a59b5
10	0x9c30d5392af26013c5d1b023286085f0
11	0xca417918b8db38ef8e79dcb0603a180e
12	0x6c9e0e8bb01e8a3ed71577c1bd314b27
13	0x78af2fda55605c60e65525f3aa55ab94
14	0x5748986263e8144055ca396a2aab10b6

Table 1: Round constants RC_r

In round r of Areion-256 (0 r 9) and Areion-512 (0 r 14), the constant RC_r is added to exactly one 128-bit word via AC(x, RC_r) inside F_2^{(r)} or F_3^{(r)}, respectively.

3.4. Areion-256 Permutation

Input: A 256-bit block divided into two 128-bit halves: x0 and x1.

Procedures:

Let (x0, x1) be the two 128-bit words of the input.

The Areion-256 permutation processes 10 rounds organized as 5 pairs. Each pair of rounds swaps the parameter positions to enable efficient pipelining:

```
For i from 0 to 9 step 2:
  Round i:   (x0, x1) := RoundFunction256(x0, x1, i)
  Round i+1: (x1, x0) := RoundFunction256(x1, x0, i+1)

Where RoundFunction256(a, b, r) is defined as:
  rc0 = RC_r (round constant from Table 1)
  rc1 = 0^128 (all-zero 128-bit value)
  b := aesenc(aesenc(a, rc0), b)
  a := aesenclast(a, rc1)
  return (a, b)
```

<t>Here, aesenc(state, key) performs SubBytes, ShiftRows, MixColumns, then XOR with key (corresponds to _mm_aesenc_si128 in Intel AES-NI or equivalent), and aesenclast(state, key) performs SubBytes, ShiftRows, then XOR with key without MixColumns (corresponds to _mm_aesenclast_si128).

Output: Concatenation of x0 and x1 after all rounds.

Implementation Note: The parameter swapping between round pairs (x0,x1) → (x1,x0) is critical for correct operation and matches the official reference implementation.

3.5. Inverse Areion-256 Permutation

The inverse permutation, denoted Areion-256-Inverse, reverses the steps of the forward permutation. It takes a 256-bit block (x0, x1) as input and returns the original block.

The decryption process iterates from round r = 9 down to 0, processing two rounds at a time to mirror the forward pair structure.

```
          // Initial state: (x0, x1) from forward output

          For i from 8 down to 0 step 2:
            // Inverse Round i+1 (Odd round: Input was (x1, x0))
            // In forward round i+1: x1 = F_1(x1), x0 = x0 XOR F_2(x1, i+1)
            // Inverse operations:
            x1 = F_1_Inverse(x1)
            x0 = x0 XOR F_2(x1, i+1)

            // Inverse Round i (Even round: Input was (x0, x1))
            // In forward round i: x0 = F_1(x0), x1 = x1 XOR F_2(x0, i)
            // Inverse operations:
            x0 = F_1_Inverse(x0)
            x1 = x1 XOR F_2(x0, i)

          return (x0, x1)

          Where F_1_Inverse(x) is defined as:
            return aesdeclast(x, 0^128)

          And F_2(x, i) is the same forward function used in encryption.
```

Note: aesdeclast(state, key) performs InverseShiftRows, InverseSubBytes, then XOR with key. This corresponds exactly to the inverse of aesenclast with a zero key.

3.6. Areion-512 Permutation

Input: A 512-bit block divided into four 128-bit quarters: A, B, C, and D.

Procedures:

Let (A, B, C, D) be the four 128-bit words of the input.

For each round r from 0 to 14:
For each round r from 0 to 14:

1. x1 = x1 XOR F_0(x0)
2. x3 = x3 XOR F_0(x2)
3. x0 = F_1(x0)
4. x2 = F_3^{(r)}(x2)
5. Shuffle (x0, x1, x2, x3) -> (x1, x2, x3, x0) (Left Rotate).

Output is the concatenation of x0, x1, x2, and x3.

Output: Concatenation of A, B, C, and D.

4. Hash Functions Based on Areion

This section specifies hash functions built from the Areion permutations. Two short fixed-input hash functions, Areion256-DM and Areion512-DM, and a variable-input hash function, Areion512-MD, are defined. All constructions are based on the Davies-Meyer compression function and the Merkle-Damgrd paradigm, as described in [Areion].

4.1. Short Fixed-Input Hash Functions

Areion256-DM and Areion512-DM are short fixed-input hash functions built from Areion-256 and Areion-512, respectively. They are intended for hashing a single 256-bit or 512-bit input block, for example short keys, nonces, or identifiers. These constructions do not define padding; callers MUST supply inputs of the required bit length.

4.1.1. Areion256-DM

Input: A 256-bit string X.

Output: A 256-bit hash value H.

The input X is mapped to the two 128-bit words (L, R) used by Areion-256 as specified in Section 3.4.

```
Areion256-DM(X):
    // X is a 256-bit block mapped to (L, R)
    Y = Areion256(L || R)
    H = Y XOR X
    return H
```

Here, Areion256 denotes the Areion-256 permutation, and all XOR operations are taken bitwise on 256-bit strings.

4.1.2. Areion512-DM

Input: A 512-bit string X.

Output: A 256-bit hash value H.

Let Y = Areion512(X) XOR X, where Areion512 is the Areion-512 permutation defined in Section 3.6. Interpret Y as a sequence of sixteen 32-bit words Y = y_0 || y_1 || ... || y_15, where y_0 is the most significant 32 bits. Then the output H is obtained by the truncation function:

H = y_2 || y_3 || y_6 || y_7 || y_8 || y_9 || y_12 || y_13

This truncation matches the definition used in the Areion design paper [Areion] for 512-bit permutations instantiated in Davies-Meyer mode.

4.2. Variable-Input Hash Function Areion512-MD

Areion512-MD is a variable-input-length hash function built from the Areion-512 permutation using a Davies-Meyer compression function in the Merkle-Damgrd framework. It outputs a 256-bit message digest and targets a 256-bit preimage security level, as in [Areion].

4.2.1. Padding and Message Parsing

Areion512-MD uses a padding method analogous to that of SHA-256, adapted to a 256-bit message block size.

Given an input message M of arbitrary length len(M) bits, the padded message is computed as follows:

* Append a single bit 1 to the message.

* Append k zero bits, where k is the smallest non-negative integer such that len(M) + 1 + k + 64 is a multiple of 256.

* Append a 64-bit big-endian representation of the original length len(M).

The resulting padded message has a bit length that is a multiple of 256. It is then parsed into t 256-bit message blocks M_0, M_1, ..., M_{t-1}; each block is treated as a 256-bit string.

4.2.2. Initial Value

Areion512-MD uses a 256-bit initial hash value H^{(0)} consisting of two 128-bit words H_0 and H_1. These constants are identical to the initial value of SHA-256, grouped into 128-bit words:

H0 = 0x6a09e667bb67ae853c6ef372a54ff53a
H1 = 0x510e527f9b05688c1f83d9ab5be0cd19

The pair (H_0, H_1) is used as the initial chaining value H^{(0)}.

4.2.3. Compression Function

The compression function of Areion512-MD instantiates the Davies-Meyer construction with the Areion-512 permutation as follows. Each 256-bit message block M_i and the current 256-bit chaining value H^{(i)} are combined into a 512-bit input to Areion-512.

Input to compression for block i: The pair (M_i, H^{(i)}), where both are 256-bit strings.

Output of compression for block i: The next chaining value H^{(i+1)}, a 256-bit string.

The 512-bit input to Areion-512 is formed by concatenating M_i and H^{(i)} and mapping them to the four 128-bit words (A, B, C, D) of Areion-512 as follows:

```
A || B = M_i
C || D = H^(i)
```

The compression function updates the chaining value as follows:

```
    H^(i+1) = Areion512-DM(A || B || C || D)
            = Areion512-DM(M_i || H^(i))
```

where Areion512-DM is the function defined in Section 4.1.2.

All concatenations above are in big-endian bit order, and XOR is taken bitwise on 256-bit strings.

4.2.4. Finalization and Output

After processing the last message block M_{t-1}, the final chaining value H^{(t)} is returned as the 256-bit hash value:

```
Areion512-MD(M) = H^(t)
```

No additional finalization transformation is applied beyond what is specified above.

5. Permutation-based AEAD with Areion

This section describes authenticated encryption schemes based on the Areion permutations. In particular, we focus on a variant of OPP mode instantiated with Areion-256, referred to as Areion256-OPP.

This section specifies the parameters of an AEAD scheme instantiated with Areion-256 using the Offset Public Permutation (OPP) mode. We refer to this instantiation as Areion256-OPP. The OPP mode itself is specified in [OPP-eprint]; this document fixes the permutation, masking function, and key/nonce formatting for the Areion-based instantiation.

5.1. Parameters

The parameters for Areion256-OPP are defined as follows:

* Underlying permutation: Areion-256 as specified in Section 3.4.
* Block size b: 256 bits.
* Word size w: 64 bits.
* Number of words in LFSR state n: 4 (the masking state is (x0, x1, x2, x3)).
* Recommended key sizes: 128 or 256 bits.
* Nonce size: 128 bits.
* Tag length: 256 bits.

5.2. Masking Function

The masking function of the Masked Even-Mansour (MEM) construction used inside OPP is implemented by a word-oriented LFSR over 256-bit states. Each state consists of four 64-bit words (x0, x1, x2, x3). The LFSR update function ϕ is defined as follows:

ϕ : (x0, x1, x2, x3) (x1, x2, x3, (x0 <<< 3) XOR (x3 >> 5))

Here, rotation to the left (<<<) and logical right shift (>>) are taken on 64-bit words. Note: In the reference implementation and test vectors, the right operation is instantiated as a logical right shift (>> 5), not a rotation. For Areion256-OPP, >> 5 MUST be interpreted as logical right shift.

5.3. Key and Nonce Formatting

Let N denote the 128-bit nonce and K denote the secret key. The underlying permutation in the MEM construction takes a 256-bit input which is mapped to the two 128-bit words (L, R) for Areion-256.

* For a 128-bit key K, the initial 256-bit input to Areion-256 is N || K, where N is the most-significant 128 bits and K is the least-significant 128 bits. This concatenation is mapped to the two 128-bit words (L, R) = (N, K).
* For a 256-bit key K, this document RECOMMENDS the following initialization:

L || R = (N || 0^128) XOR K

Here, 0^128 denotes the 128-bit all-zero value. In other words, the 256-bit quantity (N || 0^128) is XORed with the 256-bit key K, and the result is assigned to (L, R).

The following subsections describe the full encryption and decryption algorithms, derived from the Generic OPP specification [OPP-eprint] but instantiated specifically for Areion-256.

5.3.1. Helper Functions

We define the following helper functions on the 256-bit state S = (x0, x1, x2, x3):

ϕ(S) The LFSR update function defined in Section 5.2: (x1, x2, x3, (x0 <<< 3) XOR (x3 >> 5)).

β(S) Defined as ϕ(S) XOR S.

γ(S) Defined as ϕ(ϕ(S)) XOR ϕ(S) XOR S.

Pad(X) Appends a single '1' bit to X, followed by the minimum number of '0' bits to make the length a multiple of 256 bits.

Note that β(S) effectively multiplies the state polynomial by (x+1), and γ(S) multiplies by (x^2+x+1).

MEM Helper: We define the helper function MEM(X, Y), which corresponds to the MEM (Mask-Encrypt-Mask) operation used in OPP:

MEM(X, Y) = Areion-256(X XOR Y) XOR Y
MEM-Inverse(X, Y) = Areion-256-Inverse(X XOR Y) XOR Y

where Areion-256-Inverse denotes the inverse permutation of Areion-256.

5.3.2. Areion256-OPP Encryption

Input: Key K (128/256 bits), Nonce N (128 bits), Associated Data A, Message M.
Output: Ciphertext C, Tag T (256 bits).

Byte Ordering: All conversions between byte strings and 64-bit words use Little Endian byte ordering.

Padding: The padding function Pad(X) appends a single byte 0x01 to the input X, followed by the minimum number of 0x00 bytes to make the length a multiple of 256 bits.

Initialization:

```
   // Map (K, N) to initial mask La
   If |K| == 128: S_init = Nonce || Key
   Else (|K| == 256): S_init = (Nonce || 0^128) XOR Key
   La = Areion-256(S_init)
   Le = γ(La)

   // Initialize accumulators
   Sa = 0^256 (Associated Data accumulator)
   Se = 0^256 (Encryption accumulator)
```

Processing Associated Data A:

```
Break A into 256-bit blocks A_0, ..., A_{h-1}
For i = 0 to h-2:
    // Absorb full block
    B = A_i XOR La
    P = Areion-256(B)
    Sa = Sa XOR P XOR La
    La = ϕ(La)

// Process last block A_{h-1} (potentially partial or empty)
// Note: Reference implementation treats empty A same as partial
If |A| > 0:
    La = β(La)
    If |A_{h-1}| == 256:
        // Full last block treated as partial in logic flow if separate function
        // But ref optimization absorbs normally.
        // Standard OPP Logic for Last Block:
        PadA = Pad(A_{h-1}) // If already full, Pad appends full block?
                            // No, ref 'opp_absorb_lastblock' always pads.
        // Ref logic: if input length is multiple of block size, process as full block
s.
        // If partial remaining, process last block.
        // However, for consistency with ref 'opp_absorb_data':

    If A_{h-1} is full block (processed in loop):
        (Already done)
    Else (Partial A_{h-1}):
        PadA = Pad(A_{h-1})
        B = PadA XOR La
        P = Areion-256(B)
        Sa = Sa XOR P XOR La
        La = ϕ(La)
```

Correction based on Reference: The reference ‘opp_absorb_data‘ processes all full blocks. If there is a partial remainder (or if explicitly finalized), it calls ‘opp_absorb_lastblock‘. ‘opp_absorb_lastblock‘ applies ‘β ‘ to the mask, pads the input, and absorbs.

```
// Corrected Logic matching ref ‘opp_absorb_data‘ and ‘opp_absorb_lastblock‘
Mask = La
While |A| >= 256:
    Block = A[0..255]
    Sa = Sa XOR Areion-256(Block XOR Mask) XOR Mask
    Mask = ϕ (Mask)
    A = A[256..end]

If |A| > 0 (Partial Remainder):
    Mask = β (Mask)
    PadA = Pad(A)
    Sa = Sa XOR Areion-256(PadA XOR Mask) XOR Mask
    Mask = ϕ (Mask)
```

Encryption of Message M:

```
   // Se accumulates message checksum, Le is mask
   Mask = Le
   While |M| >= 256:
       Block = M[0..255]
       // Encrypt Block using MEM
       // Out = Areion-256(Block XOR Mask) XOR Mask
       Out = MEM(Block, Mask)
       Append Out to Ciphertext
       // Accumulate Plaintext
       Se = Se XOR Block
       Mask = ϕ (Mask)
       M = M[256..end]

   If |M| > 0 (Partial Remainder):
       Mask = β (Mask)
       PadM = Pad(M)
       // Encrypt Partial
       // Keystream generation: Encrypt Zero-Block with Mask
       Keystream = MEM(0^256, Mask)
       Out = Keystream XOR PadM
       Append Out[0..|M|-1] to Ciphertext
       // Accumulate Padded Plaintext
       Se = Se XOR PadM
```

Finalization (Tag Generation):

```
// Calculate Tag Mask M_tag
// Ref: ‘opp_finalise‘: m = beta(beta(mask))

M_tag = β (β (Mask))

// Tag = Sa XOR MEM(Se, M_tag)
// Block = Areion-256(Se XOR M_tag) XOR M_tag
Block = MEM(Se, M_tag)
Tag = Sa XOR Block

Return (Ciphertext, Tag)
```

5.3.3. Areion256-OPP Decryption

Decryption proceeds similarly to encryption, but uses the Areion-256-Inverse permutation for full blocks to recover the message, consistent with the OPP mode specification [OPP-eprint].

For full blocks C_j, M_j = MEM-Inverse(C_j, Mask). For the partial last block, the keystream is recovered by re-encrypting the mask (using forward Areion-256), then XORed with C_{m-1} to recover M_{m-1}. Verification fails if the calculated tag does not match the received tag.

6. Security Considerations

The security of Areion and its applications is analyzed in detail in [Areion]. This section summarizes the security claims and findings.

6.1. Security Claims

* *Permutations (Areion-256, Areion-512):* The permutations are claimed to provide 128-bit security as public permutations.

* *SFIL Hash (Areion256-DM, Areion512-DM):* These hash functions claim 256-bit security against preimage attacks. Collision resistance is not claimed as it is generally not required for their intended applications (e.g., hash-based signatures).

* *VIL Hash (Areion512-MD):* This hash function claims 256-bit security against preimage attacks and 128-bit security against collision attacks, consistent with SHA2-256.

6.2. AEAD Security

Areion256-OPP provides confidentiality and integrity for messages under the assumption that Areion-256 behaves as a pseudorandom permutation. The security properties can be described using the terminology in [RFC9771].

* *Nonce Misuse:* OPP is a nonce-respecting mode. If a nonce is repeated with the same key, confidentiality is lost for the colliding blocks (similar to ECB mode for those blocks), and authenticity is completely compromised. Implementations MUST ensure nonces are unique.

* *Related-Key Attacks:* As noted in Section 5.3, the 256-bit key initialization (N || 0^128) XOR K allows trivial related-key attacks. If an attacker can control specific bit-flips in both the Key and the Nonce, they can force the same initial state. Usage of Areion256-OPP in protocols that allow related-key queries is NOT RECOMMENDED.

* *Usage Limits:* While OPP enables parallel processing, the birthday bound security (128 bits for 256-bit block) implies that one should not process more than 2^128 blocks with a single key. Given the large block size, this limit is practically unreachable.

6.3. Cryptanalysis Summary

The full-round versions of Areion-256 (10 rounds) and Areion-512 (15 rounds) provide sufficient security margins against known attacks.

* *Differential/Linear Attacks:* The number of active S-boxes in reduced-round variants grows quickly, satisfying the 128-bit security threshold (22 active S-boxes) at 4 rounds for Areion-256 and 6 rounds for Areion-512, providing a large margin for the full-round versions.

* *Impossible Differential Attacks:* The longest impossible differential distinguishers found are for 4 rounds of Areion-256 and 8 rounds of Areion-512, well below the full round counts.

* *Integral and Zero-Sum Attacks:* The most effective integral-style attack is the zero-sum distinguisher. Such distinguishers have been found for 5 rounds of Areion-256 (with 2^32 data/time) and 10 rounds of Areion-512 (with 2^32 data/time). These are the attacks with the deepest penetration, and the full round counts are set to provide a 2x or 1.5x margin over them, respectively.

* *MITM Preimage Attacks:* For the DM hash constructions, meet-in-the-middle preimage attacks were found on 5-round Areion256-DM and 10-round Areion512-DM. This confirms the security margins of the 10-round and 15-round permutations used in the final hash constructions.

7. IANA Considerations

This document has no IANA actions.

8. References

8.1. Normative References

[Areion] Isobe, T., Ito, R., Liu, F., Minematsu, K., Nakahashi, M., Sakamoto, K., and R. Shiba, "Areion: Highly-Efficient Permutations and Its Applications (Extended Version)", 2023, <https://eprint.iacr.org/2023/794.pdf>.

[OPP-eprint] Granger, R., Jovanovic, P., Mennink, B., and S. Neves, "Improved Masking for Tweakable Blockciphers with Applications to Authenticated Encryption", 2016, <https://infoscience.epfl.ch/server/api/core/bitstreams/9580a315-f12f-482c-b3ef-591411d8c65c/content>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

8.2. Informative References

[PublicCommentOnSP800-38A] National Institute of Standards and Technology, "PUBLIC COMMENTS ON SP 800-38A, Recommendation for Block Cipher Modes of Operation: Methods and Techniques, Annex C: Ciphertext Stealing Modes and Related Issues Addendum, Three Variants of Ciphertext Stealing for CBC Mode", 2021, <https://csrc.nist.gov/CSRC/media/Projects/crypto-publication-review-project/documents/initial-comments/sp800-38a-initial-public-comments-2021.pdf>.

[RFC5116] McGrew, D., "An Interface and Algorithms for Authenticated Encryption", RFC 5116, DOI 10.17487/RFC5116, January 2008, <https://www.rfc-editor.org/info/rfc5116>.

[RFC9771] IRTF, "Properties of Authenticated Encryption with Associated Data (AEAD) Algorithms", RFC 9771, May 2025, <https://www.rfc-editor.org/info/rfc9771>.

[SP800-38A] Dworkin, M., "Recommendation for Block Cipher Modes of Operation: Methods and Techniques", 2001, <https://doi.org/10.6028/NIST.SP.800-38A>.

Appendix A. Example Implementation

The following C code provides a reference implementation for the Areion-256 and Areion-512 permutations using Intel AES-NI intrinsics, as derived from Appendix A.1 of [Areion]. This code corresponds to the optimal (2, 1)-perm and (0, 1, 0, 3, pi_1)-perm choices identified in the paper, which match the F-function definitions in Section 3.2.

Note: The implementation in [Areion] uses macros that directly map to the chosen (2,1)-perm and (0,1,0,3,pi_1)-perm. The code below is a more direct translation of the algorithmic descriptions in Section 3.4 and Section 3.6 using the function definitions from Section 3.2 for clarity.

```
#include <stdint.h>
#include <immintrin.h>

/* Round Constants (from Table 1) */
/* Stored as 15 x 128-bit values */
const uint32_t RC[15][4] = {
    {0x243f6a88, 0x85a308d3, 0x13198a2e, 0x03707344},
    {0xa4093822, 0x299f31d0, 0x082efa98, 0xec4e6c89},
    {0x452821e6, 0x38d01377, 0xbe5466cf, 0x34e90c6c},
    {0xc0ac29b7, 0xc97c50dd, 0x3f84d5b5, 0xb5470917},
    {0x9216d5d9, 0x8979fb1b, 0xd1310ba6, 0x98dfb5ac},
    {0x2ffd72db, 0xd01adfb7, 0xb8e1afed, 0x6a267e96},
    {0xba7c9045, 0xf12c7f99, 0x24a19947, 0xb3916cf7},
    {0x801f2e28, 0x58efc166, 0x36920d87, 0x1574e690},
    {0xa458fea3, 0xf4933d7e, 0x0d95748f, 0x728eb658},
    {0x718bcd58, 0x82154aee, 0x7b54a41d, 0xc25a59b5},
    {0x9c30d539, 0x2af26013, 0xc5d1b023, 0x286085f0},
    {0xca417918, 0xb8db38ef, 0x8e79dcb0, 0x603a180e},
    {0x6c9e0e8b, 0xb01e8a3e, 0xd71577c1, 0xbd314b27},
    {0x78af2fda, 0x55605c60, 0xe65525f3, 0xaa55ab94},
```

```
    {0x57489862, 0x63e81440, 0x55ca396a, 0x2aab10b6}
};

/* Load constant RC_r for little-endian byte order */
#define RC_LOAD(i) _mm_setr_epi32(RC[i][3], RC[i][2], RC[i][1], RC[i][0])
/* Zero constant for F1 and F0 */
#define RC_ZERO    _mm_setzero_si128()

/* F_0(x) = MC(SR(SB(x))) */
static inline __m128i F_0(__m128i x) {
    return _mm_aesenc_si128(x, RC_ZERO);
}

/* F_1(x) = SR(SB(x)) */
static inline __m128i F_1(__m128i x) {
    return _mm_aesenclast_si128(x, RC_ZERO);
}

/* F_2^{(r)}(x) =
     MC(SR(SB( AC(MC(SR(SB(x))), RC_r) ))) */
static inline __m128i F_2(__m128i x, int r) {
    __m128i tmp = _mm_aesenc_si128(x, RC_LOAD(r));
    return _mm_aesenc_si128(tmp, RC_ZERO);
}

/* F_3^{(r)}(x) =
     MC(SR(SB( AC(SR(SB(x)), RC_r) ))) */
static inline __m128i F_3(__m128i x, int r) {
    __m128i tmp = _mm_aesenclast_si128(x, RC_LOAD(r));
    return _mm_aesenc_si128(tmp, RC_ZERO);
}


/* Areion-256 Permutation */
void permute_areion_256(__m128i state[2])
{
    __m128i L = state[0];
    __m128i R = state[1];
    __m128i T;

    for (int i = 0; i < 10; ++i) {
        /*
           Round function logic matching Figure 1(a):
           x0 (L) feeds F2, output XORed to x1 (R).
           x0 (L) feeds F1, output becomes new x0.
           Then swap.
        */
```

```
        /* 1. T = F_2^{(r)}(L) */
        T = F_2(L, i);
        /* 2. R = R ^ T */
        R = _mm_xor_si128(R, T);
        /* 3. L = F_1(L) */
        L = F_1(L);

        /* Swap L and R for next round, except last */
        if (i < 9) {
            T = L; L = R; R = T;
        }
    }

    state[0] = L;
    state[1] = R;
}

/* Areion-512 Permutation */
void permute_areion_512(__m128i state[4])
{
    __m128i A = state[0], B = state[1], C = state[2], D = state[3];
    __m128i tmp;

    for (int i = 0; i < 15; ++i) {
        /*
          Ref Logic: Simultaneous Update
          B = F_0(A) ^ B  => aesenc(A, B)
          D = F_0(C) ^ D  => aesenc(C, D)
          A = F_1(A)      => aesenclast(A, 0)
          C = F_3(C, RC)  => aesenc(aesenclast(C, RC), 0)
        */

        __m128i next_B = _mm_aesenc_si128(A, B);
        __m128i next_D = _mm_aesenc_si128(C, D);
        __m128i next_A = _mm_aesenclast_si128(A, RC_ZERO);

        __m128i tmp_C  = _mm_aesenclast_si128(C, RC_LOAD(i));
        __m128i next_C = _mm_aesenc_si128(tmp_C, RC_ZERO);

        A = next_A;
        B = next_B;
        C = next_C;
        D = next_D;

        /* Shuffle (A,B,C,D) -> (B,C,D,A) (Left Rotate) */
        /* Note: Matches test vectors when applied in all rounds */
        tmp = A; A = B; B = C; C = D; D = tmp;
    }
```

```
    state[0] = A;
    state[1] = B;
    state[2] = C;
    state[3] = D;
}
```

Appendix B. Test Cases & Test Vectors

This section provides test vectors for the Areion permutations, hash functions, and the Areion256-OPP AEAD scheme. These vectors are derived from Appendix B of the Areion design paper [Areion] and are provided here so that implementations can be validated using this document alone.

B.1. Areion-256 Permutation

The following test vectors apply the Areion-256 permutation to a 256-bit input block.

```
/* test vector #1 */
Input:
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
Output:
28 12 a7 24 65 b2 6e 9f ca 75 83 f6 e4 12 3a a1
49 0e 35 e7 d5 20 3e 4b a2 e9 27 b0 48 2f 4d b8

/* test vector #2 */
Input:
00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f
10 11 12 13 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f
Output:
68 84 5f 13 2e e4 61 60 66 c7 02 d9 42 a3 b2 c3
a3 77 f6 5b 13 bb 05 c7 cd 1f b2 9c 89 af a1 85
```

B.2. Areion-512 Permutation

The following test vectors apply the Areion-512 permutation to a 512-bit input block.

```
/* test vector #1 */
Input:
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
Output:
b2 ad b0 4f a9 1f 90 15 59 36 71 22 cb 3c 96 a9
78 cf 3e e4 b7 3c 6a 54 3f e6 dc 85 77 91 02 e7
e3 f5 50 10 16 ce ed 1d d2 c4 8d 0b c2 12 fb 07
ad 16 87 94 bd 96 cf f3 59 09 cd d8 e2 27 49 28

/* test vector #2 */
Input:
00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f
10 11 12 13 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f
20 21 22 23 24 25 26 27 28 29 2a 2b 2c 2d 2e 2f
30 31 32 33 34 35 36 37 38 39 3a 3b 3c 3d 3e 3f
Output:
b6 90 b8 82 97 ec 47 0b 07 dd a9 2b 91 95 9c ff
13 5e 9a c5 fc 3d c9 b6 47 a4 3f 4d aa 8d a7 a4
e0 af bd d8 e6 e2 55 c2 45 27 73 6b 29 8b d6 1d
e4 60 ba b9 ea 79 15 c6 d6 dd be 05 fe 8d de 40
```

B.3. Areion256-DM

The Areion256-DM hash function is the Davies-Meyer construction using the Areion-256 permutation. The following test vectors give the 256-bit hash value H = Areion256(X) XOR X for a single 256-bit input block X.

```
/* test vector #1 */
Input:
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
Output:
28 12 a7 24 65 b2 6e 9f ca 75 83 f6 e4 12 3a a1
49 0e 35 e7 d5 20 3e 4b a2 e9 27 b0 48 2f 4d b8

/* test vector #2 */
Input:
00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f
10 11 12 13 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f
Output:
68 85 5d 10 2a e1 67 67 6e ce 08 d2 4e ae bc cc
b3 66 e4 48 07 ae 13 d0 d5 06 a8 87 95 b2 bf 9a
```

B.4. Areion512-DM

The Areion512-DM hash function uses the Areion-512 permutation in Davies-Meyer mode and truncates the 512-bit output as specified in this document. The following test vectors provide the 256-bit hash value for a single 512-bit input block.

```
/* test vector #1 */
Input:
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
Output:
59 36 71 22 cb 3c 96 a9 3f e6 dc 85 77 91 02 e7
e3 f5 50 10 16 ce ed 1d ad 16 87 94 bd 96 cf f3

/* test vector #2 */
Input:
00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f
10 11 12 13 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f
20 21 22 23 24 25 26 27 28 29 2a 2b 2c 2d 2e 2f
30 31 32 33 34 35 36 37 38 39 3a 3b 3c 3d 3e 3f
Output:
0f d4 a3 20 9d 98 92 f0 5f bd 25 56 b6 90 b9 bb
c0 8e 9f fb c2 c7 73 e5 d4 51 88 8a de 4c 23 f1
```

B.5. Areion512-MD

The Areion512-MD hash function is the Merkle-Damg\u00e5rd construction based on the Areion-512 permutation and a Davies-Meyer compression function, as specified in this document. The following test vectors give the 256-bit digest for two messages; the padding and processing are fully defined in this document.

```
/* test vector #1 */
Input:
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
Output:
7f 22 34 44 5f 3a 72 00 65 93 79 42 01 53 6c 94
09 5d ab d3 fd b5 84 67 48 d3 59 55 5c 52 e6 51

/* test vector #2 */
Input:
00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f
10 11 12 13 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f
20 21 22 23 24 25 26 27 28 29 2a 2b 2c 2d 2e 2f
30 31 32 33 34 35 36 37 38 39 3a 3b 3c 3d 3e 3f
40 41 42 43 44 45 46 47 48 49 4a 4b 4c 4d 4e 4f
50 51 52 53 54 55 56 57 58 59 5a 5b 5c 5d 5e 5f
60 61 62 63 64 65 66 67 68 69 6a 6b 6c 6d 6e 6f
70 71 72 73 74 75 76 77 78 79 7a 7b 7c 7d 7e 7f
Output:
3e 4d 31 0f be 21 d0 7b b9 00 46 88 a1 50 36 b7
ab d9 ae 2f e9 e6 0c 9a ca 2a cc 36 98 5e 60 0b
```

B.6. Areion256-OPP AEAD

The following test vectors correspond to the Areion256-OPP authenticated encryption scheme specified in this document. Each vector gives the key, nonce, associated data, plaintext, ciphertext, and authentication tag.

```
/* test vector #1 */
key:
00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f
nonce:
0f 0e 0d 0c 0b 0a 09 08 07 06 05 04 03 02 01 00
associated data:
00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f
plaintext:
00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f
10 11 12 13 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f
20 21 22 23 24 25 26 27 28 29 2a 2b 2c 2d 2e 2f
30 31 32 33 34 35 36 37 38 39 3a 3b 3c 3d 3e 3f
ciphertext:
```

```
   a4 69 c0 ab 00 bf b6 8e 1f f3 74 54 b8 3d da 59
   ef 61 1b 32 30 c0 a7 f0 a7 36 7c ab 36 c8 8a 59
   d4 dc e1 ec 7e cb 9b ad b4 77 16 93 24 b9 22 b4
   ef 04 17 8a 46 58 85 10 c2 44 ae 7b 7c bc 05 a0
   tag:
   76 12 8b 16 b6 cd 68 21 e3 7b df 58 69 27 61 a5
   05 dd 89 f4 cc 81 b7 c9 28 96 53 d6 83 a7 a8 a7

   /* test vector #2 */
   key:
   00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f
   10 11 12 13 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f
   nonce:
   0f 0e 0d 0c 0b 0a 09 08 07 06 05 04 03 02 01 00
   associated data:
   00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f
   10 11 12 13 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f
   plaintext:
   00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f
   10 11 12 13 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f
   20 21 22 23 24 25 26 27 28 29 2a 2b 2c 2d 2e 2f
   30 31 32 33 34 35 36 37 38 39 3a 3b 3c 3d 3e 3f
   40 41 42 43 44 45 46 47 48 49 4a 4b 4c 4d 4e 4f
   50 51 52 53 54 55 56 57 58 59 5a 5b 5c 5d 5e 5f
   60 61 62 63 64 65 66 67 68 69 6a 6b 6c 6d 6e 6f
   70 71 72 73 74 75 76 77 78 79 7a 7b 7c 7d 7e 7f
   ciphertext:
   16 d7 b2 7a 50 0a a0 3e a1 d1 79 f3 26 63 b3 b9
   e3 f0 41 b9 ba dd 0e 4d 59 f1 bf 87 82 5b 2a 30
   f9 00 11 96 fd 45 30 6d 59 86 d7 a2 57 0c 6c 8a
   df 68 8e 7e a2 0a 27 1b 61 e0 67 39 4f a2 85 5d
   e8 71 76 5c ce 79 5b 4d 81 6c 7e b3 74 b1 66 6f
   dc a1 de c1 af 22 8b bb eb 76 74 86 b8 52 08 c1
   26 f2 b2 79 87 94 0b 03 00 f6 23 27 86 55 ba 5d
   c9 db 3e bc 56 55 69 a0 f2 16 22 9d a4 a6 63 d8
   tag:
   25 d9 b9 09 41 45 e6 1f f0 f5 49 be 6d fe 81 a2
   ec 7c e7 8c 8f c0 ba b0 d7 72 1b 9d 80 d4 76 f7
```

Authors' Addresses

Yumi Sakemi (editor)
GMO CONNECT Inc.
Email: sakemi-yumi@gmo-connect.jp

Satoru Kanno
GMO CONNECT Inc.

Email: kanno@gmo-connect.jp

Takanori Isobe
The University of Osaka
Email: takanori.isobe@ist.osaka-u.ac.jp