ARADI and LLAMA: Low-Latency Cryptography for Memory Encryption

Patricia Greene
Mark Motley
Bryan Weeks

National Security Agency
9800 Savage Road, Fort Meade, MD 20755, USA

{ppgreen, mjmotle}@nsa.gov, beweeks@uwe.nsa.gov

Abstract

In this paper, we describe a low-latency block cipher (ARADI) and authenticated encryption mode (LLAMA) intended to support memory encryption applications.

1. Introduction

For modern processor architectures, it is often the case that the CPU can be protected within a secure enclave to ensure its security. However, it is infeasible to provide the same kind of protection for the processor's RAM.

Because of this, it is desirable to encrypt data written to RAM using a key held within the tamper boundary. In addition to confidentiality, we want to ensure integrity of this data. In other words, we want to provide authenticated encryption.

This application presents some unique challenges. In particular, in order to avoid delaying memory accesses, we must achieve authenticated encryption with very low latency [9]. In this paper, we introduce a low-latency block cipher ARADI and an authenticated encryption mode LLAMA tailored to the memory encryption application.

2. An Overview of Memory Encryption

Traditionally, cryptography for computer security has focused on providing confidentiality and integrity for communication links, or data at rest on the hard drive.

However, more recent physical attacks have made it clear that protection must also be provided for the RAM. Cold boot attacks (see [7] for example) can potentially provide access to sensitive information such as unwrapped keys.

Ideally, a memory encryption scheme is sufficiently flexible that it can operate efficiently in a variety of scenarios. For example, an efficient FPGA implementation is needed when we need to make use of commercial CPUs and memory "as-is", whereas an efficient implementation in dedicated hardware is needed if memory encryption is to be embedded in a CPU design.

Satisfying these constraints is a challenging problem as bandwidth between memory and the CPU has become highly optimized. Required response times have reduced dramatically, and the latency options between memory and CPU are limited. For example, an AM1810 ARM processor only has options for 2, 3, 4 or 5 clock cycles.

These constraints strongly influence our design decisions. Achieving sufficient throughput to handle burst data will require pipelining, so we can vary the round function of our block cipher without any additional performance cost. The stringent latency requirements also suggest that we favor block cipher designs that increase the work per round in order to reduce the round count. If we want to provide authenticated encryption, a mode that is fully parallelizable (particularly in the decrypt direction) is preferable.

On the other hand, in this application we can reasonably enforce frequent re-keying (say once per day), and so we have a relatively small upper bound on the amount of data available to the attacker on a given key. In addition, related-key security is not a significant concern for memory encryption as keys are generated and stored locally.

In this paper, rather than propose a full memory encryption scheme, we describe a block cipher and accompanying mode with performance characteristics suitable for use in memory encryption.

3. The ARADI Low-Latency Block Cipher

In this section we describe ARADI, a 128-bit block cipher with a 256-bit key, designed for low-latency applications (e.g. memory encryption). Several low-latency block ciphers have already been proposed (see for example [3, 10]), but we believe ARADI is particularly well suited to our current use case while delivering good overall performance.

Focusing on latency requires that some other design constraints be relaxed. For example, since we're now primarily interested in an unrolled hardware implementation, we needn't make all the rounds of the cipher identical. This increases the area of a round-based implementation, but allows for faster mixing.

Although we have designed ARADI to have *sufficiently* low latency for the memory encryption application, we have not endeavored to produce the lowest latency block cipher possible. Instead, subject to the given latency constraints, we have attempted to produce a design with reasonably good characteristics across a wide variety of performance criteria.

3.1. The ARADI Round Function

ARADI is a substitution-permutation network (SPN) with round function acting on a state consisting of four 32-bit words (w, x, y, z). In this representation, the s-box layer consists of 32 identical 4-bit s-boxes, with the i-th s-box acting on the i-th bit level of the words of state. The s-box is a composition of four Toffoli gates, i.e. 3-bit to 3-bit gates of the form $(a, b, c) \mapsto (a, b, c \oplus ab)$ (see Figure 3.1 for a diagram).

This s-box has optimal statistical properties for its size. Namely, the largest entry in the Difference Distribution Table is $1/4$ and the largest magnitude of any entry in the Linear Approximation Table is $1/2$. It is easy to see that any 4-bit s-box that is a composition of Toffoli gates requires at least four such gates if we want acceptable linear/differential properties.

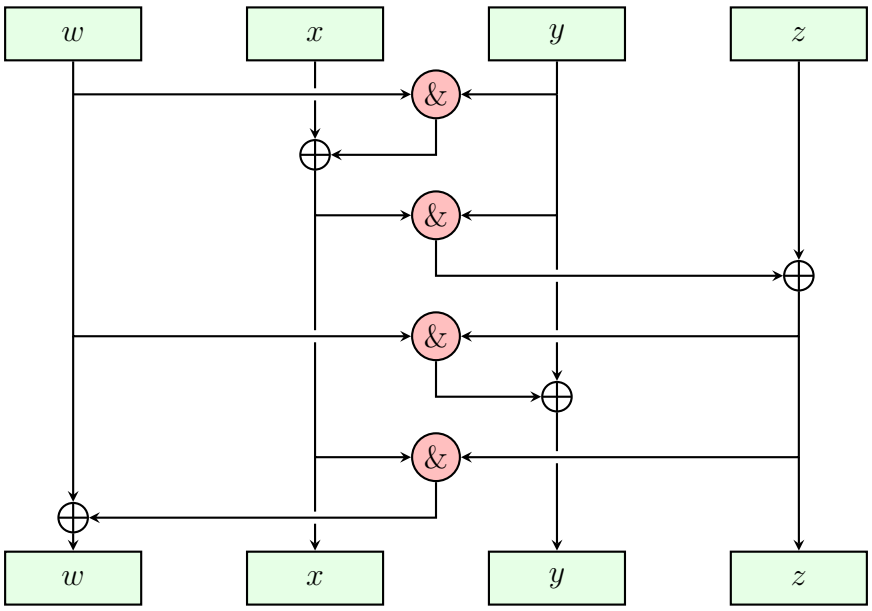

Figure 3.1: ARADI s-box layer

The linear maps for ARADI vary on a cycle of four, which makes it possible to improve resistance to linear and differential cryptanalysis. By acting independently on the words w, x, y, and z, the linear maps operate in a way complementary to the s-box layer, which operates independently on the different bit levels. So, we can write the ARADI linear maps in the form

$$(w, x, y, z) \mapsto (L_i(w), L_i(x), L_i(y), L_i(z)),$$

where L_i is a linear map on 32-bit words.

The L_i are involutions constructed from 16-bit circular shifts and `XOR`s. If the 32-bit input is composed of two 16-bit halves u, l, then L_i has the form

$$(u, l) \mapsto (u \oplus S_{16}^{a_i}(u) \oplus S_{16}^{c_i}(l), l \oplus S_{16}^{a_i}(l) \oplus S_{16}^{b_i}(u))$$

(see Figure 3.2 for a diagram). As a matrix each L_i has row density three, so that each output bit requires two `XOR`s (or one 3-input `XOR`). This, along with the fact that L_i is an involution, allows for an efficient implementation of both the encrypt and decrypt cipher.

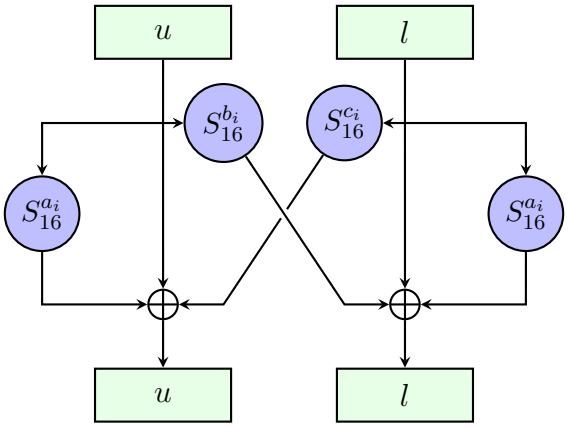

Figure 3.2: ARADI linear map applied to a 32-bit word $w = u \,||\, l$ (u and l represent the upper and lower 16 bits of w. S_{16} is a left circular shift on a 16-bit word).

The sequence of shift amounts for the ARADI linear maps is

$i \mod 4$	a_i	b_i	c_i
0	11	8	14
1	10	9	11
2	9	4	14
3	8	9	7

These parameters were chosen by a limited search over the possible values. No attempt was made to select shift values close to multiples of eight, so we anticipate ARADI is more well suited to 16-bit rather than 8-bit microprocessors.

Let π be the ARADI s-box layer and Λ_i the i-th linear map. If we denote translation by the 128-bit value v by τ_v, then the ARADI encryption function has the form (reading from right to left)

$$\tau_{k_{16}} \circ (\Lambda_{15}\pi\tau_{k_{15}}) \circ \cdots \circ (\Lambda_2\pi\tau_{k_2}) \circ (\Lambda_1\pi\tau_{k_1}) \circ (\Lambda_0\pi\tau_{k_0}),$$

where the indices of the Λ_i are reduced modulo four, and the k_i are the expanded keys (16 round keys plus a post add). The method for generating the k_i from the 256-bit base key is described in the next subsection.

3.2. The ARADI Key Schedule

The ARADI key schedule operates on an array of eight 32-bit words K_j. It makes use of two invertible linear maps M_0, M_1 operating on pairs of 32-bit words. These maps have the form

$$\begin{aligned} M_0(x,y) &= (S^1_{32}(x) \oplus y, S^3_{32}(y) \oplus S^1_{32}(x) \oplus y) \\ M_1(x,y) &= (S^9_{32}(x) \oplus y, S^{28}_{32}(y) \oplus S^9_{32}(x) \oplus y) \end{aligned}$$

where S_{32} is a left circular shift on a 32-bit word. The key schedule update varies on a period of two. Two steps (starting on an even index round) are illustrated in Figure 3.3.

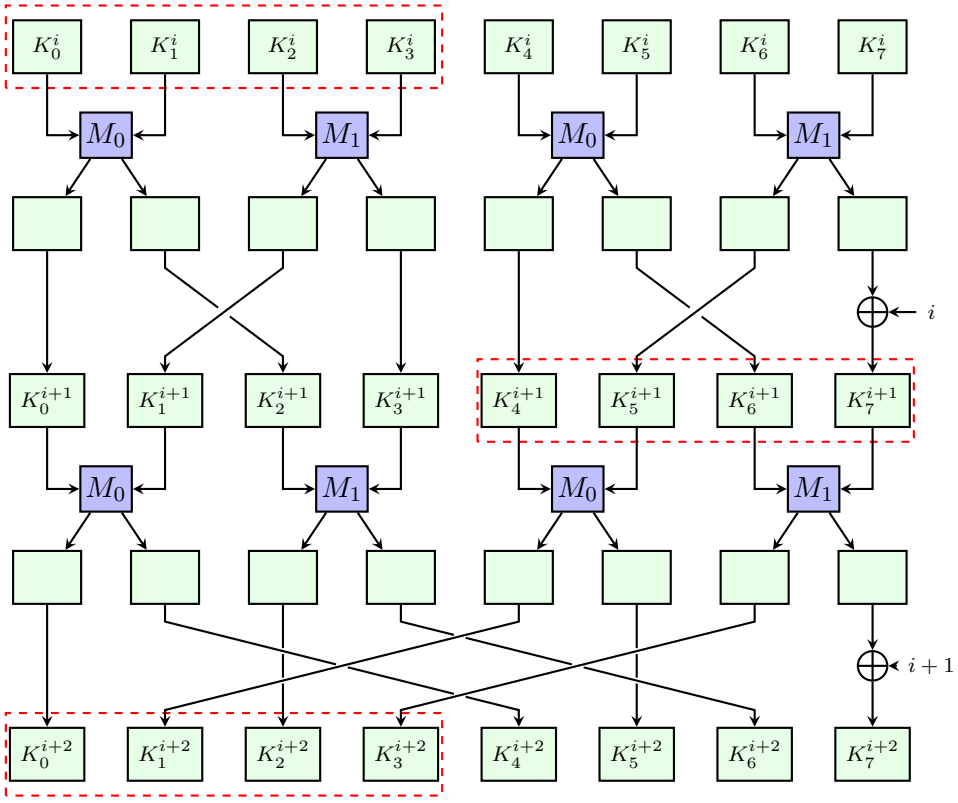

Figure 3.3: ARADI key schedule. Two consecutive rounds shown. Round keys are drawn from the boxed regions.

If we denote the register values at step i by $(K_0^i, K_1^i, K_2^i, K_3^i, K_4^i, K_5^i, K_6^i, K_7^i)$, then the i-th round key is the concatenation $K_0^i||K_1^i||K_2^i||K_3^i$ for even index rounds and $K_4^i||K_5^i||K_6^i||K_7^i$ otherwise. At each step, the four pairs of consecutive words are mixed by applying either M_0 or M_1. Then a word permutation P_j is applied, where $P_0 = (12)(56)$ and $P_1 = (14)(36)$ (j is the round modulo 2). The register is initially loaded with the eight 32-bit words of key $K_0, \ldots K_7$, and a counter is XORed into K_7 at each step in order to block slide/rotational attacks [1].

3.3. ARADI Test Vector

```
Aradi Test Vector 1
Key:                    K[7]     K[6]     K[5]     K[4]     K[3]     K[2]     K[1]     K[0]
                        1f1e1d1c 1b1a1918 17161514 13121110 0f0e0d0c 0b0a0908 07060504 03020100
-------------------------------------------------------------------------------------------

                        w        x        y        z            Round key
-------------------------------------------------------------------------------------------
Plain:                  00000000 00000000 00000000 00000000
Round 0  subcipher      367f232b 25252020 4a4a4040 7c35636b     03020100 07060504 0b0a0908 0f0e0d0c
Round 1  subcipher      ee64f20f f9bce360 418d0976 1042f571     31323734 2b2c2d2a 89829f94 eaddccfb
Round 2  subcipher      e65da996 564e30aa 8ebffad6 2cfea43d     19181312 49484342 bfb2b5b8 efe2e5e8
Round 3  subcipher      19fb2d3e 2ea0ff0a 5f80e087 eab056a4     93d8dd96 49bbf102 12918d0e 2caf0292
Round 4  subcipher      48ff5cca 5747215c 587a96c3 5c895983     7c795e5b 6e0a4a2f 708952ab 0fb51eb7
Round 5  subcipher      99f5db6e b376d237 35c04785 11c1fbe7     73be37f3 b12de15c 6d10261a 63fa1fb1
Round 6  subcipher      05dd6b05 ba589c3f 9705656e c46926d9     30e1a565 56518eba 38a4dc70 43b62b6b
Round 7  subcipher      a4a55ef4 9a71c3e1 239f293b c7ab0eba     6ff94bf4 a1525d49 960d690a f40ac5e6
Round 8  subcipher      f2ca9329 ac68354a cba990dc efec06a6     652b43fa 7ea0caa1 8356eca6 eed8d0ca
Round 9  subcipher      2b4f661f 1f94aecd 8572fae6 79ccb74a     1e8816b8 eaf40402 bf1911db d2ed83c3
Round 10 subcipher      ec3a6302 9ca4753c 91c92f12 a0ff38f9     2aed0767 d7e42972 0ddcac43 e0ce34bd
Round 11 subcipher      4205949d 0e2828c7 bba29cde 7bf46c7f     e587db6f d93a728e e7a79043 54e47c4c
Round 12 subcipher      7ea3e1a5 4f7fafe6 6673f583 e469266b     5deafddf 1235c451 b9420597 1bc4fb83
Round 13 subcipher      27e6107f 1a3e9f60 e6f1261c ad5374a4     f95881fc a9cbae8e 266a00c2 64230546
Round 14 subcipher      c621be33 d8aa33dc cf025fb6 93c87cda     cc0fab2e 5b7aad77 32495539 b022810a
Round 15 subcipher      9b4aaecf 69d197fa eb8df6a0 f60a35ba     71c5c046 8ab9aa02 d8fb0856 b7dfa119
After post-add          3f09abf4 00e3bd74 03260def b7c53912     a443053b 69322a8e e8abfb4f 41cf0ca8
Cipher:                 3f09abf4 00e3bd74 03260def b7c53912
```

Figure 3.4: ARADI test vector

3.4. ARADI Pseudocode

In this section we provide pseudocode implementing the ARADI block cipher. Here, as elsewhere, the notation || indicates concatenation of bitstrings.

```
------------------------- definitions -------------------------
w,x,y,z    = plaintext words
K[7]..K[0] = key words
M(i,j,X,Y) = (S^i_32 Y ⊕ S^j_32 X ⊕ X, S^i_32 Y ⊕ X)
L(a,b,c,x||y) = (x ⊕ S^a_16 x ⊕ S^c_16 y)||(y ⊕ S^a_16 y ⊕ S^b_16 x)
a = [11,10,9,8], b = [8,9,4,9], c = [14,11,14,7]
------------------------ key expansion ------------------------
for i = 0..15
   j ← i mod 2
   k[i][3] ← K[4j+3], k[i][2] ← K[4j+2], k[i][1] ← K[4j+1], k[i][0] ← K[4j+0]
   (K[1], K[0]) ← M(1, 3,K[1],K[0])
   (K[3], K[2]) ← M(9,28,K[3],K[2])
   (K[5], K[4]) ← M(1, 3,K[5],K[4])
   (K[7], K[6]) ← M(9,28,K[6],K[7]), K[7] ← K[7] ⊕ i
   if (j = 0)
      T ← K[1], K[1] ← K[2], K[2] ← T
      T ← K[5], K[5] ← K[6], K[6] ← T
   else
      T ← K[1], K[1] ← K[4], K[4] ← T
      T ← K[3], K[3] ← K[6], K[6] ← T
   end if
end for
k[16][3] ← K[3], k[16][2] ← K[2], k[16][1] ← K[1], k[16][0] ← K[0]
------------------------- encryption --------------------------
for i = 0..15
    z ← z ⊕ k[i][3], y ← y ⊕ k[i][2], x ← x ⊕ k[i][1], w ← w ⊕ k[i][0]
    x ← x ⊕ (w & y), z ← z ⊕ (x & y), y ← y ⊕ (w & z), w ← w ⊕ (x & z)
    j ← i mod 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)
end for
z ← z ⊕ k[16][3], y ← y ⊕ k[16][2], x ← x ⊕ k[16][1], w ← w ⊕ k[16][0]
```

Figure 3.5: ARADI encrypt pseudocode.

3.5. ARADI Performance

In this section, we discuss the performance of ARADI. Given the way we envision the algorithm being used, our main focus is on hardware/FPGA performance. Figure 3.6 below provides performance numbers for various ARADI implementations on a Xilinx Virtex7-3 FPGA, (part no. `Xc7v585tffg1157-3`) alongside reference numbers for an implementation of AES (Helion data from [8] is an estimate based on AES-CTR implementation).

Algorithm	**LUTs**	**Freq (MHz)**	**T'put (Gbps)**	**Latency (Cycles)**	**Latency (ns)**
Helion AES(Giga) [8]	9400	312.00	40.0	14	44.9
ARADI 16d1u	8453	516.00	66.0	16	31.0
ARADI 8d2u	6434	355.49	45.5	8	22.5
ARADI 4d4u	6750	194.17	24.9	4	20.6

Figure 3.6: ARADI FPGA performance (Xilinx Virtex7-3)

Here and elsewhere, the notation xdyu indicates an x-deep pipeline with y unrolled rounds between pipeline stages. In Figure 3.7, we provide a similar comparison for an Ultrascale+ FPGA (part no. `xcvu9p-flga2104-2L-e`).

Algorithm	**LUTs**	**Freq (MHz)**	**T'put (Gbps)**	**Latency (Cycles)**	**Latency (ns)**
Xiphera [11]	14662	777.00	99.46	14	18.02
Design Gateway [4]	1462	525.00	4.48	15	28.57
ARADI 16d1u	8599	793.65	101.59	16	20.16
ARADI 8d2u	8015	463.61	59.34	8	17.26
ARADI 4d4u	6630	239.69	30.68	4	16.69
ARADI 2d8u	5924	154.63	19.79	2	12.93

Figure 3.7: ARADI FPGA performance (Xilinx Ultrascale+)

In Figure 3.8, we provide performance data for ARADI in a hardware implementation, along with an AES implementation for purposes of comparison.

Algorithm	**Gates**	**Freq (MHz)**	**T'put (Gbps)**	**Latency (Cycles)**	**Latency (ns)**
Helion AES(Giga)	125k	237.5	16.0	14	59.0
ARADI 16d1u	110k	892.9	114.3	16	17.9
ARADI 4d4u	133k	333.0	42.6	4	12.0
ARADI 2d8u	218k	188.3	24.1	2	10.6

Figure 3.8: ARADI Hardware performance (65nm CMOS)

Although ARADI was not specifically designed as a software-oriented block cipher, we briefly discuss software implementations.

If the ARADI state is stored in four 32-bit words w, x, y, z, then the s-box layer requires a total of 12 operations (4 `XOR`s, 4 `AND`s, and 4 copies).

The ARADI linear map is not quite as software friendly given this representation, but an efficient implementation can be done if vector operations on 16-bit registers are available. Say we use five 128-bit registers W, X, Y, Z, T (here T is a scratch register) to process four plaintexts at once. For example,

$$W = (w_{1L}, w_{1U}, w_{2L}, w_{2U}, w_{3L}, w_{3U}, w_{4L}, w_{4U})$$

with w_{iU} and w_{iL} the upper and lower 16 bits of the 32-bit word w_i. We want to apply the same linear map L to each of the w_i, and similarly for the x_i, y_i and z_i. If we can perform vectorized circular shifts

$$(S_{16}^{r_1}(w_{1L}), S_{16}^{r_2}(w_{1U}), S_{16}^{r_3}(w_{2L}), S_{16}^{r_4}(w_{2U}), S_{16}^{r_5}(w_{3L}), S_{16}^{r_6}(w_{3U}), S_{16}^{r_7}(w_{4L}), S_{16}^{r_8}(w_{4U}))$$

and word permutations in a single operation, then L can be applied to each of the four words of W using 6 operations (1 copy, 1 word permutation, 2 circular shifts and 2 `XOR`s). This has to be repeated for X, Y, and Z so one encryption round for the four plaintexts requires a total of $12 + 24 + 4 = 40$ vector operations in this example (assuming precomputed round keys).

4. The LLAMA Authenticated Encryption Mode

LLAMA is a low-latency, parallelizable, authenticated encryption mode employing a variant of the "encrypt then MAC" construction. It provides confidentiality by counter mode encryption (`CTR`) and integrity using an authenticator LMAC (similar to PMAC [2]), described below.

Since LLAMA is intended for memory encryption, a padding scheme is unnecessary, as the data to be encrypted can be arranged in fixed-size full-block units. LLAMA is not designed to process additional authenticated data (AAD), further simplifying the mode. To minimize latency, LLAMA uses fully parallelizable operations for both encryption and the generation of a 128-bit integrity tag.

LLAMA makes use of a 128-bit block cipher E. Block cipher encryption with input x and key K is denoted $E_K(x)$. LLAMA uses the same block cipher and key for `CTR` and LMAC, for both encryption and decryption. It does not require the use of the inverse block cipher.

LLAMA operates on plaintexts P consisting of a sequence of 128-bit blocks $P = P_1, P_2, \ldots, P_L$, where L is a constant fixed for the lifetime of the key. Allowable values for L are $2 \leq L \leq 2^{32} - 2$, which correspond to plaintext lengths of $32 \leq |P| \leq 2^{36} - 32$ bytes.

In addition to a secret key K, the security of LLAMA relies on the choice of a non-zero `IV`. Plaintexts encrypted with a given key must have distinct `IV` values. The size of `IV` is application-dependent, but `IV` must fit into a 128-bit block with enough space left over for the counter. Allowable lengths ℓ for `IV` are $96 \leq \ell \leq 128 - \lceil \log_2(L+1) \rceil$. For the test vectors provided in this paper we use a 96-bit `IV`, which can be used with any permissible value of L.

The authenticator LMAC is fully parallelizable. It uses the encrypt block cipher and `XOR` alone, in contrast to modes like GCM [6] that require finite field multiplications. Further, it avoids a tag finalization step that could increase latency. LMAC requires the generation of a set of LMAC keys, but this is done only once per key period so that the work can be amortized.

We now describe LLAMA encryption and decryption in detail (see Figure 4.1 for a diagram of the encryption process).

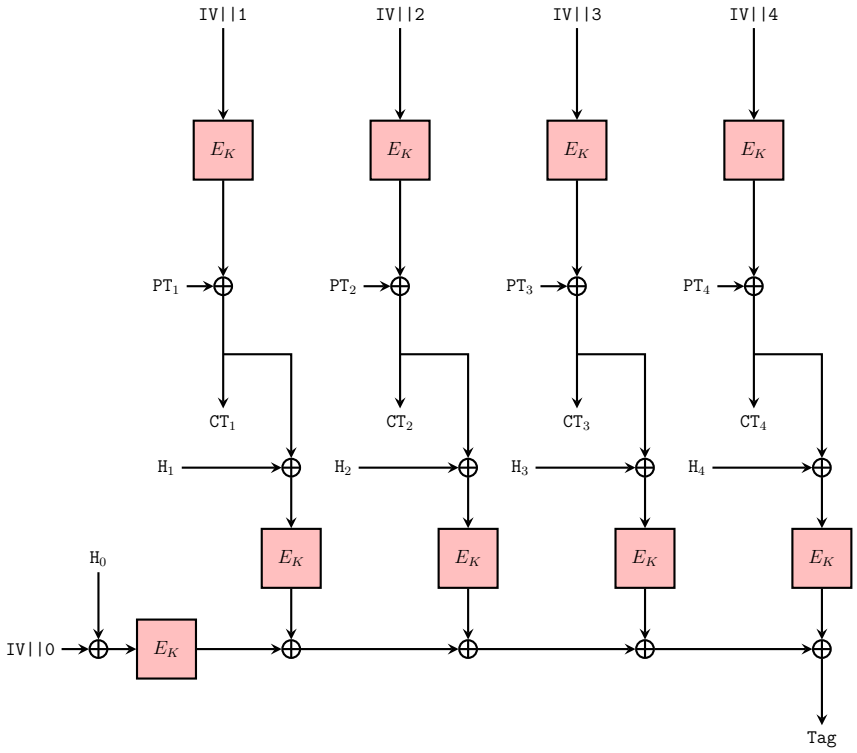

Figure 4.1: LLAMA encryption ($L = 4$, H_i precomputed)

The plaintext is encrypted using Counter Mode (`CTR`) with the given `IV` and key K. In other words, the plaintext blocks are encrypted as

$$C_i = E_K(\texttt{IV} \,||\, i) \oplus P_i,$$

for $1 \le i \le L$. The ciphertext is $C = C_1, C_2, \ldots, C_L$.

LLAMA uses LMAC as the integrity mechanism. The input to LMAC is the key, `IV`, and the ciphertext $C = C_1, C_2, \ldots C_L$. LMAC processes this input to produce a 128-bit integrity tag.

LMAC uses a sequence of 128-bit key-dependent keys, denoted $H_0, H_1, \ldots H_L$. These are computed and (securely) stored when the base key is generated.

Each LMAC key is formed by concatenating two sections of counter mode keystream with `IV` set to zero. If $X[0:7]$ denotes the leftmost (most significant) 8 bytes of a 16-byte input X, then the LMAC keys are

$$H_i = E_K((2i) \,||\, 0^{64})[0:7] \,||\, E_K((2i+1) \,||\, 0^{64})[0:7],$$

for $0 \leq i \leq L$, where $2i, 2i+1$ are encoded as 64-bit integers.

The first step in the LMAC computation is encrypting the `XOR` sum of `IV` and the first LMAC key to form S_0, where `IV` is padded with zeros to fill the 128-bit block: $S_0 = E_K((\texttt{IV} \,||\, 0^{128-\ell}) \oplus H_0)$. Then each block of C is `XOR`ed to an LMAC key and encrypted using the block cipher to get

$$S_i = E_K(C_i \oplus H_i),$$

for $1 \leq i \leq L$. The tag is $T = \bigoplus_{i=0}^{L} S_i$, the `XOR` of the S_i.

Decryption and tag verification is accomplished without using the inverse block cipher. The plaintext is recovered by using counter mode decryption on the ciphertext C initialized with `IV` so that

$$P_i = E_K(\texttt{IV} \,||\, i) \oplus C_i,$$

and the integrity tag is re-computed on the ciphertext and compared with the received tag. If they differ, then the ciphertext fails the integrity check and the plaintext must be rejected.

Note that decryption and verification can be implemented in parallel for Llama. This allows for especially low latency memory *reads*. Since Aradi decryption is efficient, an analogous mode using `CTR` paired with PMAC as an authenticator would perform decryption and verification with similar latency. However, this mode would require that both the encrypt and decrypt block ciphers be implemented, and has the disadvantage of increasing the latency for encryption.

4.1. LLAMA Pseudocode

```
------------------------------- inputs --------------------------------
K = 256-bit key, P = plaintext, IV = initial value (nonce)
---------------------------- LMAC key setup -------------------------
for i = 0 to L
  H_i ← E_K(2i||0..0)[0:7] || E_K((2i+1)||0..0)[0:7]
end for
------------------------------- LLAMA functions ----------------------
LLAMA_ENC(P,IV):
  C ← CTR(P,IV), T ← LMAC(C,IV)
return C,T

LLAMA_DEC(C,T,IV):
  T' ← LMAC(C,IV)
  if T!=T'
    return ERROR
  end if
  P ← CTR(C,IV)
return P

CTR(P,IV):
  P = p_1,p_2...p_L, CTR_Block = IV||0...1
  for i = 1 to L
    c_i ← p_i ⊕ E_K(CTR_Block)
    CTR_Block ← CTR_Block+1
  end for
return C = c_1,c_2...c_L

LMAC(C,IV):
  C = c_1,c_2...c_L, S_0 = E_K((IV||0..0) ⊕ H_0)
  for i = 1 to L
    S_i ← E_K(c_i ⊕ H_i)
  end for
  T = 0
  for i = 0 to L
    T ← T ⊕ S_i
  end for
return T
```

Figure 4.2: LLAMA pseudocode.

4.2. LLAMA Test Vectors

```
IV is 96 bits
ff112233 44556677 8899aabb 00000000

Plaintext is 64 bytes
ffeeddcc bbaa9988 77665544 33221100 00000001 00000002 00000003 00000004
20212223 24252627 28292a2b 2c2d2e2f 30313233 34353637 38393a3b 3c3d3e3f
CV[7..0] = 1f1e1d1c 1b1a1918 17161514 13121110 0f0e0d0c 0b0a0908 07060504 03020100

LMAC Key computation (L=4)
                    w        x        y        z
Aradi Input  00000000 00000000 00000000 00000000 Aradi Input  00000000 00000001 00000000 00000000
Aradi Output 3f09abf4 00e3bd74 03260def b7c53912 Aradi Output 27bfa00d 83349615 3b7095fd e633efb8
H[0] 3f09abf4 00e3bd74 27bfa00d 83349615
Aradi Input  00000000 00000002 00000000 00000000 Aradi Input  00000000 00000003 00000000 00000000
Aradi Output 4cc1c23c 04a204a4 c514133e 329286f8 Aradi Output 5c858e64 45d745af dac0923e 7b2d3272
H[1] 4cc1c23c 04a204a4 5c858e64 45d745af
Aradi Input  00000000 00000004 00000000 00000000 Aradi Input 00000000 00000005 00000000 00000000
Aradi Output 66e19501 b781a2be 44c06864 1bfccbe0 Aradi Output d7000bb1 b3690d84 cb846c76 b4a1d8b3
H[2] 66e19501 b781a2be d7000bb1 b3690d84
Aradi Input  00000000 00000006 00000000 00000000 Aradi Input  00000000 00000007 00000000 00000000
Aradi Output 8b633523 d66cc10a a6f166cd e45d4704 Aradi Output 92d7052d a5e13d30 26db3c87 0f53b391
H[3] 8b633523 d66cc10a 92d7052d a5e13d30
Aradi Input  00000000 00000008 00000000 00000000 Aradi Input  00000000 00000009 00000000 00000000
Aradi Output febd86a9 60e03ce7 5199e241 84747297 Aradi Output ad670931 a473a30e 9d272faa 37959e5d
H[4] febd86a9 60e03ce7 ad670931 a473a30e

Data Block 0 ffeeddcc bbaa9988 77665544 33221100
CTR Input    ff112233 44556677 8899aabb 00000001
CT[1]        a3af1b2b 9d36b762 633f4655 63409c45
Data Block 1 00000001 00000002 00000003 00000004
CTR Input    ff112233 44556677 8899aabb 00000002
CT[2]        7b53af19 e5414dff ed372602 4926ecb3
Data Block 2 20212223 24252627 28292a2b 2c2d2e2f
CTR Input    ff112233 44556677 8899aabb 00000003
CT[3]        7a0eee4b f6be9517 eaf367fb ed6adfe7
Data Block 3 30313233 34353637 38393a3b 3c3d3e3f
CTR Input    ff112233 44556677 8899aabb 00000004
CT[4]        49eab116 9059051a 22929cc3 61251a70

LMAC computation
E_k Input[0] c01889c7 44b6db03 af260ab6 83349615
S[0] 5f738171 9d076e37 be083b74 c278b45b
E_k Input[1] ef6ed917 9994b3c6 3fbac831 2697d9ea
S[1] b0a447f1 3d22df38 3ccc8bf7 b0c4d01b
E_k Input[2] 1db23a18 52c0ef41 3a372db3 fa4fe137
S[2] 1d818f0d daa97bf5 2bd48cfb ab21af2e
E_k Input[3] f16ddb68 20d2541d 782462d6 488be2d7
S[3] f71cb2d2 af42d9eb cab6fe32 9a7a29cf
E_k Input[4] b75737bf f0b939fd 8ff595f2 c556b97e
S[4] c91596a7 66de65fd 35759a39 e680cf7e

Llama ciphertext, tag is
a3af1b2b 9d36b762 633f4655 63409c45 7b53af19 e5414dff ed372602 4926ecb3
7a0eee4b f6be9517 eaf367fb ed6adfe7 49eab116 9059051a 22929cc3 61251a70
cc5f6df8 b31076ec 56d35873 a5672ddf
```

Figure 4.3: LLAMA test vector 1

```
IV is 96 bits
55000000 00000000 00000022 00000000
Plaintext is 64 bytes
88898a8b 8c8d8e8f 90919293 94959697 98999a9b 9c9d9e9f a0a1a2a3 a4a5a6a7
a8a9aaab acadaeaf b0b1b2b3 b4b5b6b7 b8b9babb bcbdbebf c0c1c2c3 c4c5c6c7
CV[7..0] = 10111213 14151617 18191a1b 1c1d1e1f 20212223 24252627 28292a2b 2c2d2e2f

LMAC Key computation (L=4)
                    w        x        y        z
Aradi Input  00000000 00000000 00000000 00000000 Aradi Input  00000000 00000001 00000000 00000000
Aradi Output c974550c 11722bde 40e7d38a c2a94e16 Aradi Output f3e34230 647ea2af 6d3640ac ce145aa9
H[0] c974550c 11722bde f3e34230 647ea2af
Aradi Input  00000000 00000002 00000000 00000000 Aradi Input  00000000 00000003 00000000 00000000
Aradi Output 3b72e799 d991f1fd 07b15cd9 bc5ac5b6 Aradi Output 286cc3b6 4c814dc0 741e9ceb 20517185
H[1] 3b72e799 d991f1fd 286cc3b6 4c814dc0
Aradi Input  00000000 00000004 00000000 00000000 Aradi Input  00000000 00000005 00000000 00000000
Aradi Output 14331840 913663c2 c2ad0ef1 c017f4b0 Aradi Output f5cc3c8b 9861cfdd 16903048 a37bb4e3
H[2] 14331840 913663c2 f5cc3c8b 9861cfdd
Aradi Input  00000000 00000006 00000000 00000000 Aradi Input  00000000 00000007 00000000 00000000
Aradi Output e0e21c2e 4559f34c 8097c345 21547d6b Aradi Output 5f3109aa dfa8febd a10fe784 f9f41e73
H[3] e0e21c2e 4559f34c 5f3109aa dfa8febd
Aradi Input  00000000 00000008 00000000 00000000 Aradi Input  00000000 00000009 00000000 00000000
Aradi Output 0a695d1e 4a8bfdf4 89f7f326 7cb82365 Aradi Output ba163957 14959f3e a6550b33 c910158a
H[4] 0a695d1e 4a8bfdf4 ba163957 14959f3e

Data Block 0 88898a8b 8c8d8e8f 90919293 94959697
CTR Input    55000000 00000000 00000022 00000001
CT[1]        e773398d 759a3153 8e8700b5 ee7b3943
Data Block 1 98999a9b 9c9d9e9f a0a1a2a3 a4a5a6a7
CTR Input    55000000 00000000 00000022 00000002
CT[2]        f1aa7219 0c511297 3838b6d7 e468b02b
Data Block 2 a8a9aaab acadaeaf b0b1b2b3 b4b5b6b7
CTR Input    55000000 00000000 00000022 00000003
CT[3]        d456eaf7 72ef2dca 3393026b ac607d8b
Data Block 3 b8b9babb bcbdbebf c0c1c2c3 c4c5c6c7
CTR Input    55000000 00000000 00000022 00000004
CT[4]        5106365d 4792b3dc 233f1a25 6d60b675

LMAC computation
E_k Input[0] 9c74550c 11722bde f3e34212 647ea2af
S[0] 6301ed8f fc7b87cd 46b3f755 60457a5d
E_k Input[1] dc01de14 ac0bc0ae a6ebc303 a2fa7483
S[1] 1fc71734 6de983f2 ac2ffc80 c86fb701
E_k Input[2] e5996a59 9d677155 cdf48a5c 7c097ff6
S[2] d3546b6d 24ebc200 2f3cc5db 52c16475
E_k Input[3] 34b4f6d9 37b6de86 6ca20bc1 73c88336
S[3] 50145f20 6d5a049c 170564d8 684a7f09
E_k Input[4] 5b6f6b43 0d194e28 99292372 79f5294b
S[4] 5255d9cd 508a3e2d dea9f17d 729479d8

Llama ciphertext, tag is
e773398d 759a3153 8e8700b5 ee7b3943 f1aa7219 0c511297 3838b6d7 e468b02b
d456eaf7 72ef2dca 3393026b ac607d8b 5106365d 4792b3dc 233f1a25 6d60b675
add3173b 88a9fc8e 0c0c5bab e035aff8
```

Figure 4.4: LLAMA test vector 2

```
IV is 96 bits
55000000 00000000 00000022 00000000
Received ciphertext, tag is
e773398d 759a3153 8e8700b5 ee7b3943 f1aa7219 0c511297 3838b6d7 e468b02b
d456eaf7 72ef2dca 3393026b ac607d8b 5106365d 4792b3dc 233f1a25 6d60b675
add3173b 88a9fc8e 0c0c5bab e035aff8
CV[7..0] = 10111213 14151617 18191a1b 1c1d1e1f 20212223 24252627 28292a2b 2c2d2e2f

---------- Decrypt & Verify --------------

Data Block 0 e773398d 759a3153 8e8700b5 ee7b3943
CTR Input    55000000 00000000 00000022 00000001
PT[1]        88898a8b 8c8d8e8f 90919293 94959697
Data Block 1 f1aa7219 0c511297 3838b6d7 e468b02b
CTR Input    55000000 00000000 00000022 00000002
PT[2]        98999a9b 9c9d9e9f a0a1a2a3 a4a5a6a7
Data Block 2 d456eaf7 72ef2dca 3393026b ac607d8b
CTR Input    55000000 00000000 00000022 00000003
PT[3]        a8a9aaab acadaeaf b0b1b2b3 b4b5b6b7
Data Block 3 5106365d 4792b3dc 233f1a25 6d60b675
CTR Input    55000000 00000000 00000022 00000004
PT[4]        b8b9babb bcbdbebf c0c1c2c3 c4c5c6c7

LMAC computation

E_k Input[0] 9c74550c 11722bde f3e34212 647ea2af
S[0] 6301ed8f fc7b87cd 46b3f755 60457a5d
E_k Input[1] dc01de14 ac0bc0ae a6ebc303 a2fa7483
S[1] 1fc71734 6de983f2 ac2ffc80 c86fb701
E_k Input[2] e5996a59 9d677155 cdf48a5c 7c097ff6
S[2] d3546b6d 24ebc200 2f3cc5db 52c16475
E_k Input[3] 34b4f6d9 37b6de86 6ca20bc1 73c88336
S[3] 50145f20 6d5a049c 170564d8 684a7f09
E_k Input[4] 5b6f6b43 0d194e28 99292372 79f5294b
S[4] 5255d9cd 508a3e2d dea9f17d 729479d8

Recovered plaintext is
88898a8b 8c8d8e8f 90919293 94959697 98999a9b 9c9d9e9f a0a1a2a3 a4a5a6a7
a8a9aaab acadaeaf b0b1b2b3 b4b5b6b7 b8b9babb bcbdbebf c0c1c2c3 c4c5c6c7
Computed tag is:
add3173b 88a9fc8e 0c0c5bab e035aff8
Received tag is:
add3173b 88a9fc8e 0c0c5bab e035aff8
Return code is 1 (1=success,0=failure)
```

Figure 4.5: LLAMA test vector 3

4.3. LLAMA Performance

In this section, we provide performance data for LLAMA mode, instantiated with ARADI as the underlying block cipher. Figure 4.6 provides performance numbers for ARADI-LLAMA on a Xilinx Virtex7-3 FPGA, alongside an AES-GCM implementation.

Algorithm	**LUTs**	**Freq (MHz)**	**T'put (Gbps)**	**Latency (Cycles)**	**Latency (ns)**
Helion AES-GCM [8]	12400	200.0	12.0	19	95.0
Helion AES-GCM [8]	21200	200.0	25.0	19	95.0
LLAMA 4d4u	6750	194.2	12.4	8	41.2
LLAMA 4d4u-x2	13500	194.2	24.9	4/8	20.6

Figure 4.6: ARADI-LLAMA performance (Xilinx Virtex7-3)

Figure 4.7 provides performance numbers for ARADI-LLAMA on a Xilinx Ultrascale+ FPGA.

Algorithm	**LUTs**	**Freq (MHz)**	**T'put (Gbps)**	**Latency (Cycles)**	**Latency (ns)**
Xiphera [12]	26564	633.7	81.1	19	30.0
Design Gateway [5]	62864	270.0	138.2	19	70.3
LLAMA 4d4u	6663	299.4	19.2	8	26.7
LLAMA 4d4u-x2	13326	299.4	38.3	4/8	13.4

Figure 4.7: ARADI-LLAMA performance (Xilinx Ultrascale+)

In a software implementation of ARADI-LLAMA, there is very little overhead from the mode. Given an efficient software implementation of ARADI, it is straightforward to produce an efficient ARADI-LLAMA implementation.

References

[1] A. Biryukov and D. Wagner. Slide attacks. In *Fast Software Encryption – FSE'99*, number 1636 in Lecture Notes in Computer Science, pages 245–259. Springer-Verlag, 1999.

[2] J. Black and P. Rogaway. A Block Cipher Mode of Operation for Parallelizable Message Authentication (PMAC). In *Advances in Cryptology – EUROCRYPT'02*, volume 2332 of *Lecture Notes in Computer Science*, pages 384–397. Springer-Verlag, 2002.

[3] J. Borghoff, A. Canteaut, T. Güneysu, E. Kavun, M. Knezević, L. Knudsen, and G. Leander. PRINCE – A Low-Latency Block Cipher for Pervasive Computing Applications. In *Wang X., Sako K. (eds) Advances in Cryptology – ASIACRYPT 2012*, volume 7658 of *Lecture Notes in Computer Science*. Springer, Berlin, Heidelberg, 2012.

[4] Design Gateway AES. `https://dgway.com/products/IP/AES-IP/AES256IP-datasheet-xilinx-en/`.

[5] Design Gateway AES-GCM. `https://dgway.com/products/IP/AES-IP/AES256GCM100GIP-datasheet-xilinx-en`.

[6] M. Dworkin. NIST Special Publication 800-38-D Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC. Natl. Inst. Stand. Technol., 2007.

[7] A. Halderman and S. Schoen. Lest we remember: Cold boot attacks on encryption keys. In *USENIX Security Symposium*, 2008.

[8] Helion Giga AES Cores. `www.heliontech.com/aes_giga.htm`.

[9] M. Knezević, V. Nikov, and P. Rombouts. Low-Latency Encryption – Is "Lightweight = Light + Wait?". In *Cryptographic Hardware and Embedded Systems – CHES 2012*, number 7418 in Lecture Notes in Computer Science. Springer-Verlag, 2012.

[10] G. Leander, T. Moos, A. Moradi, and S. Rasoolzadeh. The SPEEDY Family of Block Ciphers: Engineering an Ultra Low-Latency Cipher from Gate-Level for Secure Processor Architectures. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 4:510–545, 2021.

[11] Xiphera AES. `https://xiphera.com/symmetric-encryption/aes-ctr/`.

[12] Xiphera AES-GCM. `https://xiphera.com/wp-content/uploads/`.