Limdolen

A Lightweight Authenticated Encryption Algorithm

Carl E. Mehner

March 2019

Abstract. This document defines the Limdolen Authenticated Encryption with Associated Data (AEAD) system including the overall AEAD specific algorithm and the core encryption round function. The system takes as input variable-length plaintext, variable length associated data, a fixed-length nonce, and a fixed-length key and creates as output a fixed-length authentication tag and variable-length ciphertext.

Keywords: Lightweight Cryptography , Limdolen , AEAD , AXR

Introduction

Limdolen (meaning “swift secret” in the Sindarin language), is a block based cipher with two defined family members, 128-bit and 256-bit. This paper describes one method of creating cipher text using the Limdolen round function, however, other methods may be defined in the future using the same core round function. Each of the functions and calculations in this algorithm are able to use a maximum size of one 8-bit byte in order to better work with devices that are more constrained in their processor architecture. The AEAD construction that uses the round function is, at a high level, similar to the PMAC SIV [04] algorithm (Parallelizable Message Authentication Code; Synthetic Initialization Vector) created by Phillip Rogaway but has been adapted in several key ways to accommodate the lower power and lower memory footprints of lightweight devices.

Using the algorithm with inputs of 0-bytes length of associated data with a 0-byte length plaintext will return a ciphertext of 0-length, and therefore output just a tag value. Inputs that only contain associated data correspond to running the authentication algorithm for the associated data with padding added to the end of the block size. Doing so allows the algorithm to act as a keyed message authentication function for the associated data.

When run with non-zero lengths of associated data and plaintext, the associated data and plaintext are authenticated by the tag value, and the plaintext is encrypted to create the ciphertext.

Figure 1 shows a high level overview of the data flow within Limdolen.

A high-level overview of Limdolen's data flow

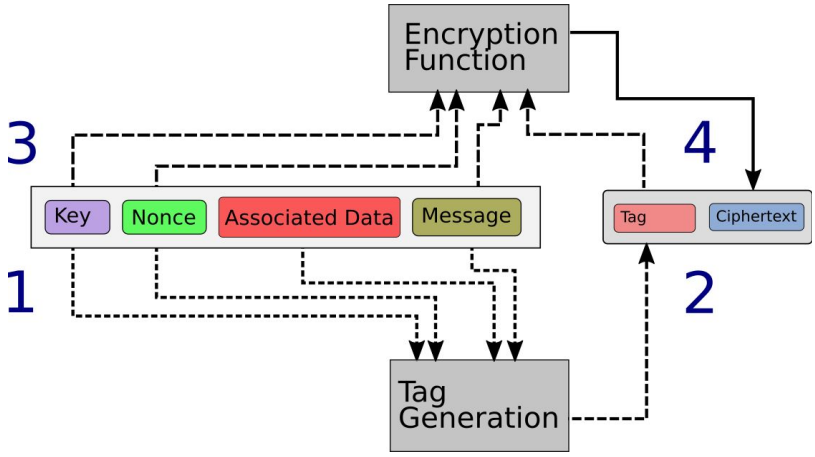

Figure 1

Terminology

Associated Data - data processed by a cipher that remains unencrypted, but is still authenticated along with the cipher text, by the cipher's authentication tag.

MAC - Message Authentication Code, used in AEAD ciphers to facilitate the authentication of the data

tag - Another term for MAC

XOR - exclusive-OR (XOR) function combines two arrays bit-by-bit. Denoted by the ⊕ symbol.

AND - logical AND function combines two arrays bit-by-bit. Denoted by the ⊞ symbol.

Left Shift - Each bit in a bitstring s is shifted one position towards the most significant position. Denoted by s << 1.

Right Shift - Each bit in a bit string s is shifted one position towards the least significant position. Denoted by s >> 1.

Circle Shift - each bit in an 8-bit byte s is shifted by one position to the more significant position in the byte. The most significant byte is moved to the least significant byte.

Denoted by <<< 1 (s).

Additional circle shifts are defined by doing a single shift multiple times, for example, <<< 5 (s) is the same as doing <<< 1 (s) five times.

AXR - A cipher that is made up of AND, XOR, and rotational operations

$E_K(m)$ - Calls the round function with key K and message m.

Encryption and Decryption

Parameters

The design of Limdolen lends itself to tunable parameters, the key and nonce can be any length but will be padded to the length in bits of the Limdolen family name using zeros. The tag may be truncated to a smaller length as well to save space or in proportion to the importance of authenticating the plaintext/ciphertext pairing. Below are the recommended parameters for each defined family member, the expected security strength of Limdolen 128 is 128-bits and the expected security strength of Limdolen 256 is 256-bits. The performance of Limdolen is only affected choosing a different parameter for the number of rounds within a given family member. The performance cost of Limdolen 256 is roughly two times that of Limdolen 128. The primary member chosen is Limdolen 128, as it provides a reasonable balance of security, integrity, and efficiency.

Limdolen Recommended Parameter Lengths

Family Member	Key Length	Nonce Length	Tag Length	Rounds
Limdolen 128	128-bits	128-bits	128-bits	16
Limdolen 256	256-bits	256-bits	256-bits	16

Figure 2

Round Function

Round Key

Each round in the round function uses a unique key derived from the master key provided to the encryption algorithm. Pentagonal numbers[1] are used for the key schedule round constants as a "nothing up my sleeves" series in order to help introduce additional information into each key while keeping the number

[1] This choice for Limdolen was sparked by a reference in Doctor Who story 280, "The Tsuranga Conundrum".

smaller than the maximum value stored in one 8-bit byte for each round. To create the round key, each byte of the master key is combined with the n^{th} generalized pentagonal number, where n is the number of the round. Generalized pentagonal numbers are pentagonal numbers calculated using a sequence of 0, 1, -1, 2, -2, 3… Pentagonal numbers are defined as

```
p = (3n^2 - n) / 2
```

An alternate way to calculate the Generalized Pentagonal Number Sequence is:

```
g_p = ((-5 + (-1)^n - 6n)*(-1 + (-1)^n - 6n)) / 96
```

The sequence for the 16 rounds is as follows:

```
0, 1, 2, 5, 7, 12, 15, 22, 26, 35, 40, 51, 57, 70, 77, 92
```

LimdolenRoundKey

```
Input: Key128||256 , n8 (Round Number)
Output: RoundKey128||256
for i ← 0 to length(Key) - 1 do
      RoundKey[i] = Key[i] ⊕ LimdolenRoundConstant(n)
done
```

LimdolenRoundConstant

```
Input: n8 (Round Number)
Output: RoundConstant8
# for each round. n = 0 through n = 15
return ((-5 + (-1)^n - 6n)*(-1 + (-1)^n - 6n))/96
```

Input - 128-bit

An input array of length 128-bits is XORed with a round key. The 128-bit result is then split into 4 groups of 32 bits, Input = { Q, R, S, T }; R is then combined bit-wise with S using the logical AND function to create an intermediate value Z. Each byte in the value Z is passed through a circular left shift of 2 and combined with Q by XOR resulting in Q'. Each byte in the value Z is passed through a circular left shift of 7 and combined with T by XOR resulting in T'. Q' and T' are combined by AND resulting in Z'. Each byte in the value Z' is passed through a circular left shift of 3 and combined with R by XOR resulting in R'. Each byte in the value Z' is passed through a circular left shift of 5 and combined with S by XOR resulting in S'. The 4 bytes that make up S' are then rotated in a circle shift one byte to the left.

The values chosen in the series of circular shifts performed on Z and Z' are simply the prime numbers less than 8. The values are recombined into an output array Output = { S', T', Q', R' }. The output array is fed back into the function as input. Overall, the process occurs sixteen times.

LimdolenInternalRoundFunction

```
# Input: Key128 , Input128 , Round8
# Output: Round_Output128
{ Q32, R32, S32, T32 } = Input ⊕ LimdolenRoundKey(Key, Round)
Q’ = Q ⊕ <<< 2 (R ⊞ S)
T’ = T ⊕ <<< 7 (R ⊞ S)
R’ = R ⊕ <<< 3 (Q’ ⊞ T’)
{ d8, e8, f8, g8 } = S ⊕ <<< 5 (Q’ ⊞ T’)
S’ = { e, f, g, d }
Round_Output = { S’, T’, Q’, R’ }
return Round_Output
```

LimdolenRounds128

```
# Input: Key128 , Input128
# Output: Round_Output128
for i ← 0 to 16 do
     Input = LimdolenInternalRoundFunction(Key, Input, i)
done
return Input
```

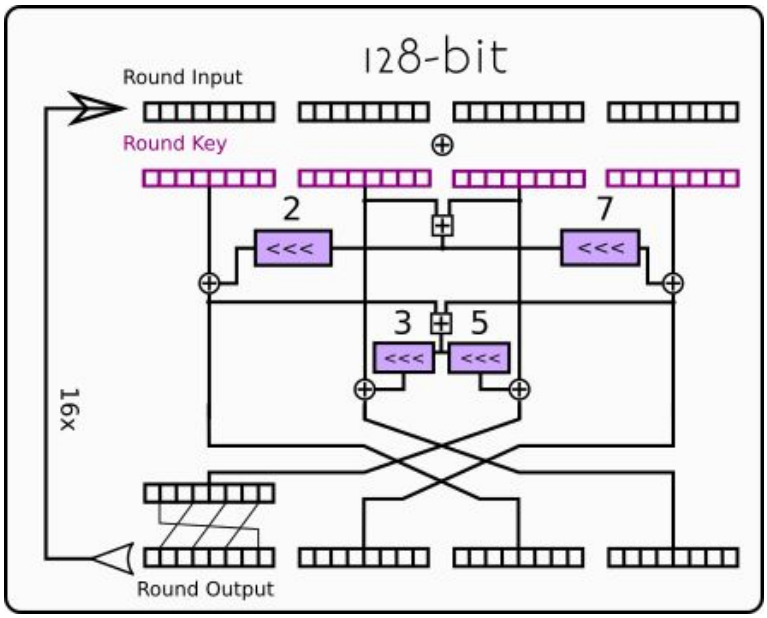

Figure 3

Input - 256-bit

The function for treating 256-bit input uses the 128-bit construction as a basis. The same number of rounds and the same round constants are used in each corresponding round in both 128 and 256-bit members. The 256-bit input and key are split into two equal halves; Input = {u, v}, Key = {k1, k2} and each half is passed through a single round function of the 128-bit construct. At the end of each round, the output {u', v'} is processed by XORing u' into v' and replacing u' with v' such that the round function output is {v', u'⊕v'}. The output is then fed back into the function as the next round's input.

LimdolenRounds256

```
# Input: Key256 , Input256
# Output: Round_Output256
for i ← 0 to 16 do
      { u128, v128 } = { Input[0..127], Input[128..255] }
      u' = LimdolenInternalRoundFunction(Key[0..127], u, i)
      v' = LimdolenInternalRoundFunction(Key[128..255], v, i)
      Input = { v' , u' ⊕ v' }
done
return Input
```

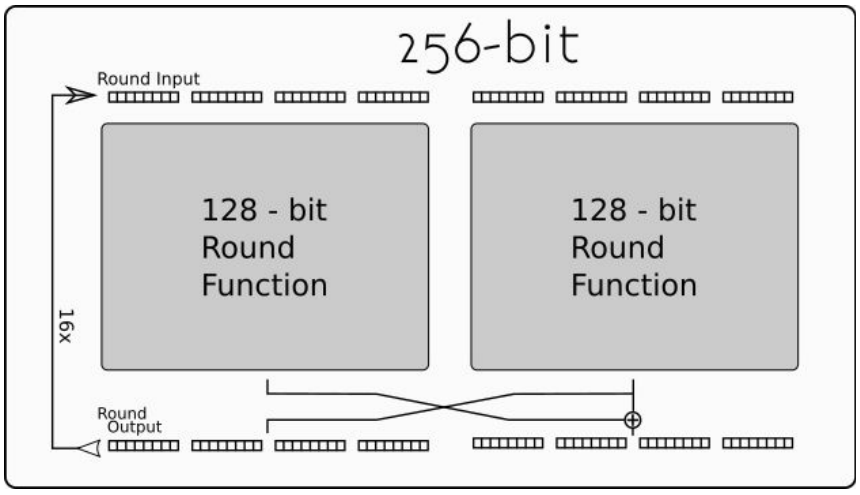

Figure 4

Limdolen AEAD

The Limdolen AEAD function, as mentioned in the introduction, is modeled after the Rogaway PMAC SIV method [04], but with changes to accommodate constrained environments, the full algorithm is laid out in this section.

Calculating the Tag Value

The key used for the PMAC, rather than having a separate key as used in the original algorithm in order to keep the initial amount of memory required low, is derived rather than specified separately. However, if the extra memory is not an issue, a separate PMAC key may be defined and used here as "aeadK" if desired. The derived key aeadK is created by passing in the provided nonce as input to the round function using the provided master key as the round function key, aeadK = E_K(nonce).

To begin, the associated data and plaintext are directly concatenated together to create one input message. On the byte following the final byte of the concatenated AAD and message input, or if the length of the AAD and the message falls on a block boundary (either 128-bit or 256-bit, depending on the algorithm family member chosen), the final byte of the input, a single byte will be added or XORed to the input to mark the end of the datastream and start of the padding. The padding marker will one byte with the most significant bit set to a value of one (if the AAD length is non-zero), or the two most significant bits set to a value of one (if the AAD length is zero). Next, zero bytes are appended to the message until the message is of a length equivalent to the next multiple of the blocklength. By setting one of these two end-of-message markers, Limdolen can distinguishing between two hypothetical invocations: one with just AAD and one with just plaintext, where the AAD and plaintext in those separate messages are equivalent in length and content. Following the concatenation and padding, the message is broken into blocks, b[1] to b[m], equal to the length in bits designated by the family name. The first major departure from Rogaway's PMAC is in

determining the first data to combine with the input block. In the the original PMAC, there is a longer calculation in a Galois field with 2^n points, GF(2^n). This is done by finding, "$\gamma i \cdot L$" for various values of 'i'. (The details and descriptions of these calculations are contained in the Rogaway paper and are not reproduced here.) Due to Limdolen's target of constrained environments, rather than a series of calculations, we will alternate between i=0 and i=1, the two most common[2] values of i in $\gamma i \cdot L$ that were used in the original algorithm to treat the authenticated data and plaintext. Further, α will be split into an array of bytes and the operations against the bytes in array α will be taken against each byte of α rather than done in GF(2^{128}) or GF(2^{256}) so that each byte of α will be operated on individually.

Limdolen simplifies the algorithm to be: $\gamma i \cdot L$; where i=0 is $\alpha = E_{aeadK}(0^n)$ (encrypting a block of only zero values) and i=1 is denoted as $\alpha \cdot \chi = \alpha[n] << 1$ for each byte, *n* in α. The final calculation for the MAC uses i= -1, and will be calculated as $\alpha \cdot \chi^{-1} = \alpha >> 1$. If there is more than one block in b, the first block b[1] through b[m-1] will be combined via XOR beginning with α (for b[1]) followed by $\alpha \cdot \chi$ (for b[2]), and continuing the alternating pattern up to b[m-1].

Next, each of the blocks, 1 through m-1, are passed through the round function using the aead key, $E_{aeadK}(b[x])$ the output is XORed with an aggregate value, originally set to all zeros, such that $0^n \oplus E_{aeadK}(b[1]) \oplus E_{aeadK}(b[1]) \ldots E_{aeadK}(b[m-1])$. The very last block of input is not combined with α or $\alpha \cdot \chi$ nor passed through the round function, rather it is combined with the aggregate value. Penultimately, the aggregate value is combined with $\alpha \cdot \chi^{-1}$ using XOR. Finally, the aggregate value is passed through the round function once more to produce the tag value; tag = E_{aeadK}(aggregate_value).

[2] Exepmlified in https://github.com/miscreant/miscreant.py/blob/master/miscreant/ctz.py

LimdolenTagGeneration

```
# Input: Key_n , Nonce_n , Plaintext_1 , AssociatedData_k
# Output: Tag_n
Tag = 0^n
aeadK = LimdolenRounds(Key, Nonce)
α = LimdolenRounds(aeadK , 0^n )
Input = concatenate ( AssociatedData , Plaintext )
# break input into blocks of blocksize
blockToggle = 1
for each ( block in Input.split(n) ) do
      if(LastBlock)
            Tag = Tag ⊕ addPaddingMarker(block)
      else if(blockToggle == 1)
            Tag = Tag ⊕ LimdolenRounds(aeadK, block ⊕ α )
      else if(blockToggle == 0)
            Tag = Tag ⊕ LimdolenRounds(aeadK, block ⊕ (α<<1))
      blockToggle = blockToggle ⊕ 1
Done
Tag = Tag ⊕ (α >> 1)
Tag = LimdolenRounds(aeadK, Tag)
return Tag
```

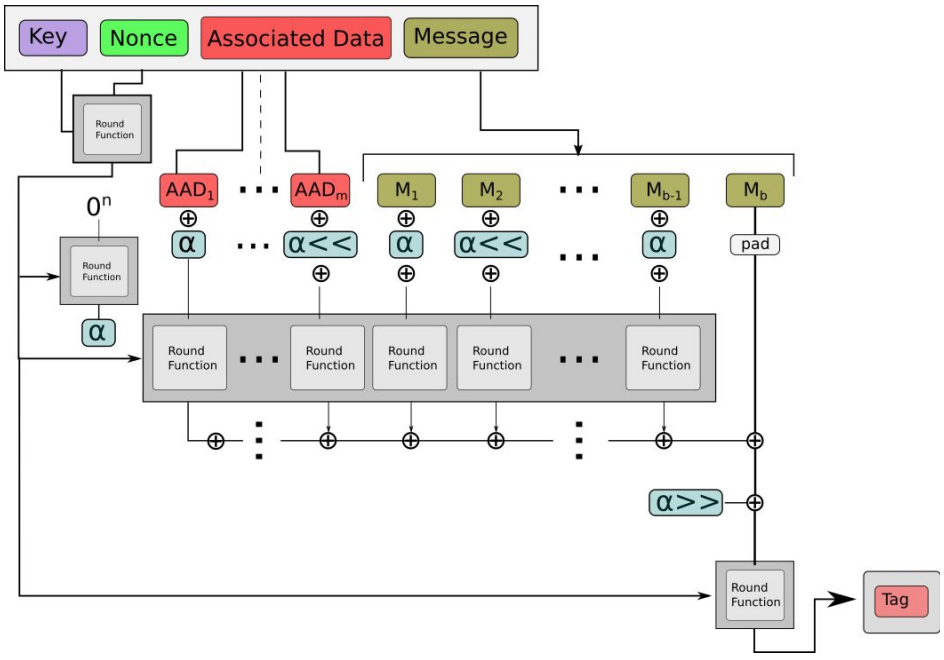

Figure 5

Creating the Ciphertext

First calculate the tag value as described above and combine it with the provided nonce value using XOR to create a starting input to the round function; in_val = tag ⊕ nonce. The plaintext value is split into blocks equal to the size of the key for the particular algorithm family member but is not padded. For each plaintext block, pt[m]; encrypt in_val using the master key k and combine the output with the plaintext block. E_K(in_val) ⊕ pt[m] = ciphertext[m]. For each successive block, add 1 modulo 2^n, where n is the key size, to the previous block's in_val as a counter to create a different initial state for the round function. The final block may not end at a block boundary, therefore the ciphertext will be equal in length to the message. The tag and ciphertext may be stored as a single array, when this is the case the tag must come first, and the ciphertext must follow.

LimdolenEncryption

```
# Input: Key_n , Nonce_n , Plaintext_1 , AssociatedData_k
# Output: Ciphertext_1 , Tag_n
Tag = LimdolenTagGeneration(Key, Nonce, Plaintext, AssociatedData)
RoundIn = Tag ⊕ Nonce
for each ( block in Plaintext.split(n) ) do
      RoundIn = LimdolenRounds(Key, RoundIn)
      Ciphertext.concatenate( block ⊕ RoundIn )
      RoundIn = (RoundIn + 1) % (2^n)
done
return Ciphertext, Tag
```

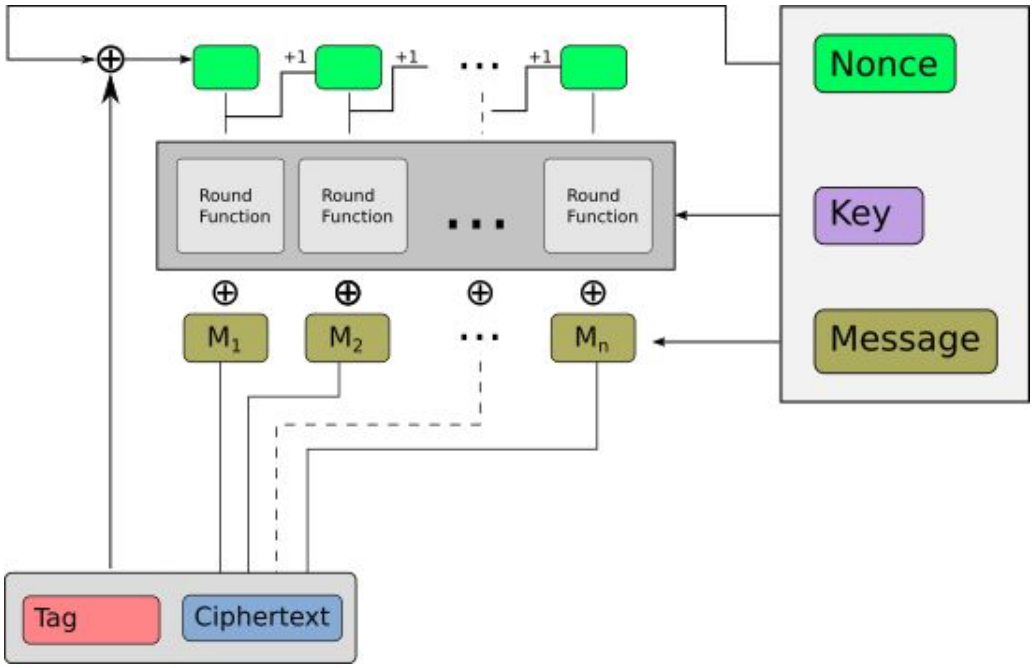

Figure 6

Decryption

To decrypt encrypted ciphertext, take as input the tag and ciphertext, if provided in a single array, the tag will be the first n-bits of the array where n is the tag length parameter for the family member used, which is recommended at either 128-bits or 256-bits and the ciphertext will be the remaining data. The same method used to generate the cipher text will be used except rather than combining the output of the round function with the message, the output will be combined via XOR with the ciphertext and output of that operation will be the unconfirmed plaintext. To authenticate the plaintext, the tag generation algorithm will be used as described above. If the tag value generated by the decrypted plaintext does not match the tag value provided as input to the decryption function, the decryption operation fails, and shall return an error.

LimdolenDecryption

```
# Input: Key_n , Nonce_n , Ciphertext_l , AssociatedData_k , Tag_n
# Output: Plaintext_l
RoundIn = Tag ⊕ Nonce
for each ( block in Ciphertext.split(n) ) do
      RoundIn = LimdolenRounds(Key, RoundIn)
      Plaintext.concatenate( block ⊕ RoundIn )
      RoundIn = (RoundIn + 1) % (2^n)
Done
if(Tag != LimdolenTagGeneration(Key, Nonce, Plaintext, AssociatedData))
      return ERROR
return Plaintext
```

Security Claims

	Bit Strength	
Security Claim	**Limdolen 128**	**Limdolen 256**
Confidentiality of plaintext	128	256
Integrity of plaintext	128	256
Integrity of Associated Data	128	256

Figure 7

Because of the SIV construction in this algorithm it is nonce reuse misuse resistant, that is to say, repeating the nonce with the same key, will not lead to plaintext or key revelation. However, the same Associated Data, Key, Nonce and Plaintext will always result in the same Ciphertext and tag values, therefore, choosing a random nonce for every new invocation is still important.

This cryptosystem also has the property of being side channel resistant in the sense that timing attacks will not apply because each iteration will complete in constant time. Furthermore, this system is resistant to fault attacks, due to the inherent properties of the "Symmetric Mode" design as discussed in the Baksi et al Paper [01]. In addition, any faults in the tag creation process will cause complete failure of decryption and any faults in the cipher/decipher process will cause wrong plaintext or ciphertext without revealing anything about the key.

Because the counter/nonce, { tag $\oplus$ nonce + block_number } is not constrained to a length smaller than the blocksize, and because the nonce is tied to the tag, the nonce will only repeat for a given set of plaintext once every 2^n blocks, where n is the size in bits of the nonce. The probability that a ciphertext will repeat within a set of 2^n distinct nonces is $2^{n/2}$ given the round function being sufficiently random. The amount of plaintext that can be encrypted under one key safely is therefore $2^{n/2}$ n-bits blocks, where n is the key length.

Security Analysis

Because of the keystream-like ciphertext construction which creates a ciphertext the same length as the plaintext, the length of the plaintext will be a known value. If the length of the plaintext is sensitive, the plaintext must first be encoded in such a way as to ensure that it is padded to a set length, such as the cipher family's block boundary.

The round function is designed to be both nonlinear and invertible, which allows for use of the round function in other modes besides the counter-style mode described in this paper.

Each bit in the key is diffused to every other bit in the output after at most 8 rounds. The choice of 16 rounds was to insure that each bit in each 8-bit section the key interacted with every other position of that

byte at least twice. Figure 8 shows the properties after each round of in the round function for each bit position in the 128-bit input from a representative example of each of the four “streams”. Because the output of each round is fed back in as input, each bit of output that is shifted is combined with other bit positions in successive rounds, allowing for quick and efficient diffusion of both the original input and the key bits.

Diffusion of bit 128

00200000000000000100000008000000
00910400880000000420000064010000
00ce3280661100003291040 0bb8c2000
107fd946bbcc2200d9ce3280ff779104
c8ffef3bff779944ff7fd946ffffee32
67fffffdffffee33ffffef3bffffffdd
bfffffffffffffddffffffffdfffffff
ffffffffffffffffffffffffffffffff

Diffusion of bit 64

00910000800000000400000024000000
00ce32006600000012910000b38c0000
007fd946bbcc0000d9ce3200ff769100
c8ffef3bff779900ef7fd946ffffce32
67fffffdffffee33ffffef3bffffffd9
bfffffffffffffddffffffffdfffffff
ffffffffffffffffffffffffffffffff
ffffffffffffffffffffffffffffffff

Diffusion of bit 96

00900000800000000400000025000000
00ee120067000000129000 00bb840000
00ffdd42bbcc0000ddee1200ff779000
48ffffbbff779900ffffdd42ffffee12
77fffffffffee33ffffffbbffffffdd
fffffffffffffddfffffffffffffffff
ffffffffffffffffffffffffffffffff
ffffffffffffffffffffffffffffffff

Diffusion of bit 1

10000000000000800000000000000004
c80200000000004410000002800000b2
77194000880000b3c8020019461000dd
ffee2308661100dd771940eebbc802ff
ffffdd64bbcc22ffffee23ffff7719ff
fffffbbff7799ffffffddffffffeeff
ffffffffffffeeffffffffffffffffff
ffffffffffffffffffffffffffffffff

Figure 8

The security of the tag generation and the encryption relies on a pseudo-random stream of bytes being output from the round function. To test this, an all zero key and all zero nonce were used along with an

input of only zeros (used only to build the "SIV", or tag value) and a single stream of output of the interaction of the nonce and round function (as shown in Figure 5), was used as input to both DieHarder and NIST's Statistical Test Suite. An overview of the results of DieHarder's battery of 114 tests for 128 and 256 bit variants are shown below in Figure 6. The one failed test was the RGB Bit Distribution Test with an n-tuple of 1, in both the 128 and 256-bit families. The same test with n-tuple values 2-12 passed in both the 128 and 256 bit versions.

DieHarder Test Results for Limdolen 128 and 256

(Both family members have identical results)

Passed	113
Failed	*1*

Failed Tests

test_name	ntup	tsamples	psamples	p-value	Assessment
rgb_bitdist	1	100000	100	0	FAILED

Figure 9

The NIST Statistical Test Suite version 1.2.1 [08] was used to test both the 128 and 256 bit variants with 10 streams of 12,800,000 and 25,600,000 bits respectively. Both the 128-bit version and the 256-bit version passed all 188 of the 188 NIST statistical tests.

These statistical randomness tests were then repeated this time with a random key, random nonce, and all-zero input. The results of DieHarder's battery of 114 tests for the 128-bit family member were identical to the tests using all-zero input (Figure 6). The 256-bit family, passed all the DieHarder tests. The NIST Statistical Test Suite for a random key and nonce similarly passed all the tests in the test suite for both family members of Limdolen.

There are no known attacks on Limdolen aside from a bruteforce trial of every key which will succeed in time, $O(2^{n-1})$.

Performance and Implementation

The PMAC construction used in the tag generation is designed to be fully parallelizable, in addition, after the nonce generation, all blocks used for creating the blocked used for encryption are parallelizable, thus in systems that have multiple processors or in specially designed hardware, the system can be made considerably faster.

Overall, there are a total of 2+2m round function calls for a total number of m blocks of plaintext for both family members; every additional *k* blocks of associated data adds *k* additional round function calls (here round function refers to 16 iterations of the Limdolen internal round function). An optimized implementation of the 128-bit round function performs at least 1024 operations (16 rounds * 4 streams * [4 AND operation per stream + 8 XOR operations per stream + 4 rotations per stream]) when performed one byte at a time, as it would be in an 8-bit processor. In hardware, this can be reduced to parallel bitwise operations (two stages of: 1 AND (over 8 bytes), 2 rotations (each over 4 bytes), 1 XOR (over 8 bytes)) using a 265-bit input buffer, 128-bits for the key and 128-bits for the round input, the output buffer is 128-bits. Overall a hardware implementation of the Limdolen Round Function will use a minimum of 33 bytes of RAM (16 byte input buffer, 16 byte key input, 1 byte round counter) and 16 bytes of ROM for the round constants. The round function, implemented purley in logic gates would require at least 1280 NAND gate equivalents, each round would require 5-6 nano seconds to process, making the throughput of a 16-rounds, 21 GBps on a device such as a Virtex-7 FPGA.

Experimentally, the unoptimized 128-bit variant was tested using SUPERCOP. The results show that the performance on one block of data is an average of 3572 and 3532 CPU cycles for the encrypt and decrypt functions respectively. The unoptimized 256-bit variant was also tested and resulted in an average of 6913 and 7133 CPU cycles for encrypting and decrypting one block of data.

Algorithm Features

Limdolen has a number of features that allow it to excel in constrained devices and in hardware or software that allow for parallel processing. The design of the round function is such that it allows for parallel operations within 8-bit increments, while still allowing for operations that operate on larger increments of data if supported by the software or hardware. Because of the AXR construction of the round function and the lack of algebraic computations creating branching decisions, Limdolen is not susceptible to side channel attacks looking at the timing of the function. Similarly, timing attacks manipulating ciphertext attacking Limdolen as a decryption oracle will not cause timing delays since the full algorithm must run, both the decryption and tag generation before any plaintext candidates are accepted or rejected. Under adaptive chosen-plaintext, due to the SIV-style construction, and adaptive forgery attacks, because of the non-malleability of the ciphertext due to the authenticated nature of decrypted ciphertext, the confidentiality and the integrity of the ciphertexts is retained. Further, reusing nonces within Limdolen does not lead to situations where plaintext can be modified, nor to where authentication keys may be retrieved, unlike some implementations of AES-GCM as discussed by Hanno Böck. [05]

References

01. Baksi, A., Shivam Bhasin, S., Breier, J., Khairallah, M., Peyrin, T., “Protecting Block Ciphers against Differential Fault Attacks without Re-keying”, February 2018, https://eprint.iacr.org/2018/085.pdf.

02. Bernstein, D., "ChaCha, a variant of Salsa20", January 2008, http://cr.yp.to/chacha/chacha-20080128.pdf.

03. Bernstein, D., "SUPERCOP", January 2019, https://bench.cr.yp.to/supercop.html.

04. Black, J and Rogaway, P., "A Block-Cipher Mode of Operation for Parallelizable Message Authentication", February 2002, http://web.cs.ucdavis.edu/~rogaway/ocb/pmac.pdf.

05. Böck, H., “Nonce-Disrespecting Adversaries: Practical Forgery Attacks on GCM in TLS”, August 2016, https://www.usenix.org/system/files/conference/woot16/woot16-paper-bock.pdf.

06. Brown, R. G., “Dieharder: A Random Number Test Suite”, Version 3.31.1, May 2017, https://webhome.phy.duke.edu/~rgb/General/dieharder.php

07. Harkins, D., "Synthetic Initialization Vector (SIV) Authenticated Encryption Using the Advanced Encryption Standard (AES)", RFC 5297, DOI 10.17487/RFC5297, October 2008, https://www.rfc-editor.org/info/rfc5297.

08. NIST, “Statistical Test Suite”, Version 2.1.2, July 2014, https://csrc.nist.gov/projects/random-bit-generation/documentation-and-software