Modified Reversible Radix-n Machine Computations with Novel N-state Carry Functions
Applications in Cryptography
Peter Lablans – LabCipher

Abstract

Machine Cryptography uses digital circuitry that performs operations described as modulo-n additions (as in SHA-256 and ChaCha20) and additions over GF(2^k) (as in AES-GCM and many other protocols). These additions are well documented and are used to 'mix' data. By themselves these operations provide no additional security. The security is provided by unknown or secret data "a key" that is mixed with what is called "cleartext" to generate "ciphertext."

Sets of data, like multiple bytes, are treated in this article as a radix-n words of n-state elements. A key-word of k n-state elements and a "cleartext" word of k n-state elements are added radix-n resulting in a "ciphertext" word of k n-state elements. Canonical forms of n-state carry functions for reversible modulo-n additions and additions over GF(n) are provided. This modified and preferably secret radix-n addition increases security in an unpredictable way.

The radix-n modifications are applied in known cryptography such as AES-GCM, ChaCha20 and SHA-256. This improves the security and appears to be resistant against quantum computer (QC) attacks.

Key Contributions:

- Defines canonical n-state carry functions in carry based reversible additions modulo-n and over GF(n)
- Provides modified versions of reversible n-state carry functions
- Applies the carry modifications in machine cryptography
- Increases overall security of standard machine cryptography

Keywords: N-state ripple carry adder, reversibility, radix-n, n-state borrow, non-canonical modification, cryptography, finite fields, Finite Lab-Transform (FLT), encryption, hashing, AES-GCM, ChaCha20, SHA-256

1. The Used Notation Herein

The applied notation herein is derived from the teaching approach of Prof. Dr. Gerrit Blaauw, one of the 3 chief designers of the legendary IBM System/360, as applied in his book "Digital System Implementation," [1]. Therein Blaauw uses APL to describe standard digital components like the AND gate by for instance: $c \leftarrow a \wedge b$ with 'a' and 'b' being binary input operands and 'c' its output. APL allows a symbol like ϴ to be introduced/defined as an operator representing, for instance, a specific look-up table or customized operation. One may also compute, in for instance Matlab, c = ϴ(a,b) wherein ϴ is a predefined lookup table. While unusual, there is

mathematically nothing wrong with this type of representation, as long as one observes the properties of the operation/table, like associativity or lack thereof.

A straightforward table-based notation is applied herein scn for n-state addition-like operations and mgn for multiplication-like operations. Thus, c=scn(a,b) or c=mgn(a,b) become the preferred notation herein. It allows direct replacement in computer programs by relevant look-up tables.

2. Radix-n Addition

The following table provides the lookup table sc5 for modulo-5 addition:

sc5	0	1	2	3	4
0	0	1	2	3	4
1	1	2	3	4	0
2	2	3	4	0	1
3	3	4	0	1	2
4	4	0	1	2	3

Figure 1

Instead of having to repeat or use the table one may use c=sc5(row,column), wherein one convention is that the first operand indicates the row index and the second one is the column index of a look-up table.

The corresponding 5-state carry function car5 is provided in Figure 2 below.

car5	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0	0	1
2	0	0	0	1	1
3	0	0	1	1	1
4	0	1	1	1	1

Figure 2

Figure 3 illustrates the radix-5 carry ripple addition using sc5 or the modulo-5 addition to determine a residue and the 5-state function car5 to determine the carry which is 0 or 1. The carry ripple adder is a positional operation wherein digits of 2 n-state operands are processed. There are always cycles of 2 steps:

step 1: residues of operand elements in corresponding positions are determined, and placed in the position of the operand elements, and

step 2: two elements in a position determine a carry digit, which is placed in a neighboring position.

Each cycle is repeated until no more carry digits are generated. It corresponds often with the way humans do decimal additions, placing a carry to the left of the position of the elements that determine the carry. It coincides with big-endian notation. In arithmetic the generated carry digit is 0 if the sum of two digits is smaller than n. If the sum is n or greater than n the carry digit is a 1. This is independent of a size of n. This rule relates to arithmetic and not to switching, as a switch doesn't know that a sum is equal or greater than n. In n-state machine operations one may use a look-up table like provided in Figure 2 with fixed criteria for generating a carry digit, though one may also implement with circuitry based on a Karnaugh map for instance.

In arithmetic the carry digit ripples from right to left. However, that is because of a desire to make the machine operation similar to our arithmetical approach. For applications like encryption such a direction that is not required and one may ripple from left to right which may have additional benefits.

Figure 3 shows the steps of a 4-digit radix-5 addition of 2 operands. The ripple is from right to left. The selected operands are 5-state: op1= [0 4 4 4 4] and op2 = [0 0 0 0 1]. These are of course 5 digit operands. This is done to illustrate the carry-out.

op1		0	4	4	4	4
op2		0	0	0	0	1
sc5		0	4	4	4	0
car5		0	0	0	1	-
sc5		0	4	4	0	0
car5		0	0	1	-	-
sc5		0	4	0	0	0
car5		0	1	-	-	-
sc5		0	0	0	0	0
car5		1	-	-	-	-

Figure 3

The rows sc5 show the residues addition modulo-5 of the digits of the operands and rows car5 show the generated carry digits generated by the operand digits in a neighboring position. The dash ('-') means that no carry operation is performed and the residue digit in row sc5 corresponding to '-' is the resulting sum digit. The 4-digit sum is [0 0 0 0]. The carry-out is 1, but may not be used in cryptography. If one would have used only the modulo-5 addition, with no carry propagation, the result would have been [4 4 4 0].

Machine arithmetic performs subtraction in binary form and generally not in the subtraction form that humans do, but applies 2's-complement addition. One may do that also for modulo-n subtraction. Rather than doing modulo-n subtraction with a corresponding borrow function one may perform n's complement addition.

3. Radix-n Subtraction

The following table in Figure 4 provides the lookup table for modulo-5 subtraction. The table is derived from out=(row-column) mod-5.

min5	0	1	2	3	4
0	0	4	3	2	1
1	1	0	4	3	2
2	2	1	0	4	3
3	3	2	1	0	4
4	4	3	2	1	0

Figure 4

The corresponding 5-state borrow function bor5 is provided in Figure 5 below.

bor5	0	1	2	3	4
0	0	1	1	1	1
1	0	0	1	1	1
2	0	0	0	1	1
3	0	0	0	0	1
4	0	0	0	0	0

Figure 5

From an arithmetical point of view the borrow table has a borrow 1 if the row-index is smaller than the column-index. Otherwise, the borrow digit is 0.

Rule: From a machine computational point of view, a borrow table corresponding to a carry table is the mirror image of the carry table, mirrored over the horizontal axis so the rows change index. Or index_row_borrow(i)= index_row_carry(n-i-1) for index starting at 0. This is a canonical rule for all reversible additions in this article.

Figure 6 illustrates the radix-5 carry ripple subtraction using min5 or the modulo-5 subtraction to determine a residue and the 5-state function bor5 to determine the borrow which is 0 or 1. The borrow ripple subtraction is also a positional operation with steps identical to the addition but with different functions.

Figure 6 shows the steps of a 4-digit radix-5 subtraction of 2 operands. The ripple is from right to left. The selected operands are 5-state: op1= [0 0 0 0 0] and op2 = [0 0 0 0 1]. These are of course 5 digit operands. This is done to illustrate the borrow-out.

op1		0	0	0	0	0
op2		0	0	0	0	1
min5		0	0	0	0	4
bor5		0	0	0	1	-
min5		0	0	0	4	4
bor5		0	0	1	-	-
min5		0	0	4	4	4
bor5		0	1	-	-	-
min5		0	4	4	4	4
bor5		1	-	-	-	-

Figure 6

The rows min5 show the residues subtraction modulo-5 of the digits of the operands and rows bor5 show the generated borrow digits generated by the operand digits in a neighboring position. The dash ('-') means that no borrow operation is performed and the residue digit in row min5 corresponding to '-' is the resulting difference digit. The 4-digit difference is [4 4 4 4]. The borrow-out is 1, but would not be used in cryptography. If one would have used only the modulo-5 subtraction, with no borrow propagation, the result would have been [0 0 0 4].

4. Radix-n Operations in Cryptography

How would the above be applied in encryption/decryption. Assume a secret key Key and a cleartext CTex. The above then suggests a ciphertext Ciphtex as encryption: Ciphtex = CTex+Key and decryption: CTex = Ciphtex-Key. One may also use subtraction for encryption and addition for decryption, of course.

In most encryption schemes the '+' is performed as an addition over finite field GF(n=2^k), which in implementation is the bitwise XORing of words of k bits, wherein CTex, Key and Ciphtex are all words of k bits. One may replace k bits by their decimal representation and perform encryption by modulo-n with n=2^k addition of the individual n-state elements. This may initially throw off an attacker, but there are only a limited number of replacements of the addition over GF(n). An informed attacker may simply run through the most likely functions. The modification is thus be more of an annoyance than a true barrier.

The radix-n addition provides a further obfuscation, by introducing the carry propagation. The uncertainty herein is formed by the length of the word of n-state elements. The more elements, the greater the probability of a carry propagating through the elements.

However, the fact that a carry is either 0 or 1 works against the uncertainty. It would be better to have a carry that is never 0. However, in that case it is always 1. A carry always being 1 is of course the same as adding a word [1 1 1 0].

Start carry from earlier position

The last carry digit being 0 is caused by being the first step in carry determination. The application in encryption is different from being a truly arithmetical operation. One can 'force' a carry (or borrow) by extending the number of elements. For instance op1=[0 4 4 4 4] may be op1e=[0 4 4 4 4 **0**] wherein the last 0 is not part of the cleartext, but a way to invoke a carry. In this example the extra 0 does not do anything, but that will change further below.

The above also shows why computation of carry digits from left to right may be easier, because there is already an inactive digit of the unused input carry.

4. The Modified Carry Function Modulo-n

A first canonical modification in the modulo-n addition is changing the 0 to any value between 0 and n-1, and changing the 1 to any value between 0 and n-1.

The following example shows the new carry function for n=5 and modulo-5 addition wherein the carry 0 is replaced by 2 and the carry 1 is replaced by 4. It is illustrated in Figure 7 as function can5.

can5	0	1	2	3	4
0	2	2	2	2	2
1	2	2	2	2	4
2	2	2	2	4	4
3	2	2	4	4	4
4	2	4	4	4	4

Figure 7

The corresponding modified borrow function is shown in Figure 8 as function bon5.

bon5	0	1	2	3	4
0	2	4	4	4	4
1	2	2	4	4	4
2	2	2	2	4	4
3	2	2	2	2	4
4	2	2	2	2	2

Figure 8

The modified radix-5 addition of as illustrated in Figure 3, now using modified function can5 still with sc5 is shown in Figure 9.

op1		0	4	4	4	4
op2		0	0	0	0	1
sc5		0	4	4	4	0
can5		2	2	2	4	-
sc5		2	1	1	3	0
can5		4	4	4	-	-
sc5		1	0	0	3	0
can5		4	4	-	-	-
sc5		0	4	0	3	0
can5		2	-	-	-	-

Figure 9

The modified radix-5 subtraction as illustrated in Figure 6, now using modified function bon5 still with min5 is shown in Figure 10.

op1		0	4	0	3	0
op2		0	0	0	0	1
min5		0	4	0	3	4
bon5		2	2	2	4	-
min5		3	2	3	4	4
bon5		2	4	4	-	-
min5		1	3	4	4	4
bon5		4	4	-	-	-
min5		2	4	4	4	4
bon5		4	-	-	-	-

Figure 10

One can see that the cleartext [4 4 4 4] is correctly recovered by the modified radix-5 subtraction. One should of course ignore the borrow-out.

Non-canonical Modifications of Modulo-n Additions

The above modification, substituting a 0 and a 1 in a carry function, with a value (n-1), the substitutions preferably being different, may be considered a canonical modification. While there are many possible modifications when n is large, these modifications are predictable. Even though brute force attacks on these modifications may be extremely time-consuming.

There are some ways to construct non-canonical carry functions. One is pure brute force trial-and-error going through all permutations of an n-by-n table. This runs quickly into limitations of size, as there are n^(n^2) different permutations.

One may build upon this, by further modifying newly found reversible n-state carry tables by applying n-state reversible inverters on the outputs and check if they remain reversible.

As an example of a non-canonical 4-state reversible carry function that corresponds to modulo-4 addition plus4 is the following 4-state carry function cax4 and the corresponding borrow function box4 with subtraction modulo-4 min4.

plus4	0	1	2	3	cax4	0	1	2	3	min4	0	1	2	3	box4	0	1	2	3
0	0	1	2	3		3	3	3	0		0	3	2	1		3	1	2	2
1	1	2	3	0		3	2	3	2		1	0	3	2		3	3	2	3
2	2	3	0	1		3	3	2	3		2	1	0	3		3	2	3	2
3	3	0	1	2		3	1	2	2		3	2	1	0		3	3	3	0

Figure 11

One can perform the carry ripple addition and borrow ripple subtraction as shown in Figures 9 and 10, using the functions of Figure 11 with op1= [3 3 3 3] and op2=[0 0 0 1]. One gets sum=op1+op2= [3 3 0 0] and dif=sum-op2=[3 3 3 3]. Demonstrating that the operations are reversible.

5. Individually per Digit Modified Carry Function Modulo-n

The above illustrates how one simply can modify the carry function for a radix-n addition/subtraction while maintaining reversibility of the operations. It seems a bit trivial to do this for a radix-5 addition. The possibilities of modifications increase dramatically for larger values of n. For instance, for elements represented by 8 bits one has n=256 and there are over 65,000 possible modifications.

Another important opportunity for obfuscation is to use a different carry function for each position in the operands. As long as one applies the corresponding (flipped) borrow function, one is able to fully reverse the addition, using the modification rules as stated above.

6. Carry Function in Carry-less Operations

In cryptography bitwise XORing of words of k bits is much used. This operation may be described as an addition over GF(n=2^k). Such an operation is a carry-less operation as no carry digits are generated. For instance, a word of 32 bits may be considered a word of 4 8-bit words. The bitwise XORing of 32-bits is identical of addition of 4 corresponding words of 8 bits by addition over GF(n=2^8=256). One may compute the individual words of 8 bits as 256-state elements and convert the 256-state result back into a word of 8 bits and reconstitute the word of 32-bits from the 4 256-state elements. This will, of course, create the 32-bits word formed by bitwise XORing.

No carry element is applied by the above addition. Confusion and obfuscation will be created unexpectedly if by some way a carry element is introduced.

Let's first go back to the binary or radix-2 addition. The two binary functions to operate a radix-2 addition are the XOR as representing the modulo-2 addition and the AND function for generating the carry. A binary carry is generated only as both operand bits are 1.

Extension Functions

Extension functions are known in finite field arithmetic, where an element in an extension field is represented by a polynomial over the base field.

For the current extension another approach will be applied. The base functions are the XOR and the AND. The extended functions for n=2^k will be created by doing bitwise XOR and AND of words of k bits and then converting the resulting words of k bits into their decimal representation. Figure 12 illustrates the results for n=2^2=4 and Figure 13 illustrates the radix-4 addition of operands with 4 4-state elements. The function sc4 is the addition over GF(4) and car4 is the corresponding 4-state carry function. The subtraction over GF(4) is also sc4 and the corresponding 4-state borrow function bor4 is the 'flipped' set of rows of car4

sc4	0	1	2	3	car4	0	1	2	3	bor4	0	1	2	3
0	0	1	2	3		0	0	0	0		0	1	2	3
1	1	0	3	2		0	1	0	1		0	0	2	2
2	2	3	0	1		0	0	2	2		0	1	0	1
3	3	2	1	0		0	1	2	3		0	0	0	0

Figure 12

op1		0	3	3	3	3
op2		0	0	0	0	1
sc4		0	3	3	3	0
car4		0	0	0	1	-
sc4		0	3	3	0	0
car4		0	0	1	-	-
sc4		0	3	0	0	0
car4		0	1	-	-	-
sc4		0	0	0	0	0
car4		1	-	-	-	-

Figure 13

The modified radix-4 addition over GF(4) with carry as illustrated in Figure 13 above. One would expect the result of the addition over GF(4) of [3 3 3 3] and [0 0 0 1] to be [3 3 3 0], but in fact

the result is [0 0 0 0]. One should check manually that this is a reversible operation, by using bor4 as the corresponding borrow function.

Modified Carry Functions

There is no single canonical rule for modifying the carry function related to additions over GF(2^k). One may take different approaches. For n=4 one may use a brute force approach and one may find sets of modified carry functions of car4 while keeping sc4 unmodified. One of such a set is provided in Figure 14.

sc4	0	1	2	3	cap4	0	1	2	3	bop4	0	1	2	3
0	0	1	2	3		3	3	2	3		3	2	1	1
1	1	0	3	2		3	2	2	2		3	3	1	0
2	2	3	0	1		3	3	1	0		3	2	2	2
3	3	2	1	0		3	2	1	1		3	3	2	3

Figure 14

The radix-4 addition with ripple addition using sc4 and cap4 is shown in Figure 15.

op1		0	3	3	3	3
op2		0	0	0	0	1
sc4		0	3	3	3	2
cap4		3	3	3	2	-
sc4		3	0	0	1	2
cap4		1	1	1	-	-
sc4		0	1	1	1	2
cap4		3	3	-	-	-
sc4		3	2	1	1	2
cap4		2	-	-	-	-

Figure 15

The result of the modified radix-4 addition of [3 3 3 3] with [0 0 0 1] is then [2 1 1 2].

Figure 16 shows the modified subtraction using sc4 and bop4, using [2 1 1 2] and [0 0 0 1] as operands.

op1		0	2	1	1	2
op2		0	0	0	0	1
sc4		0	2	1	1	3
bop4		3	3	3	2	-
sc4		3	1	2	3	3
bop4		2	0	1	-	-
sc4		1	1	3	3	3
bop4		3	2	-	-	-
sc4		2	3	3	3	3
bop4		1	-	-	-	-

Figure 16

One can see that the cleartext operand is recovered with the modified radix-4 operation. A test program has verified that reversibility for this operation applies to all possible 4-state operands.

7. Further Extending Non-canonical N-state Functions

In cryptography, uncertainty about used n-state functions will increase security. The greater the number of possible functions, the greater the required effort of attack by brute force and the greater inherent security.

In the case of the addition radix-n using addition over GF(n=2^k) the extension of the carry function by element-wise application of the AND function as illustrated in Figure 12, has (of course) only a single extension for each further power of k. The function of Figure 12 is obtained by doing a bitwise AND of words of 2 bits. One may create the extension for n=2^3=8 by doing the same for words of 3 bits and so on. One may also extend the 4-state function of Figure 12 to 4^2 or 16-state elements. One may do this by taking all words of 2 4-state elements and do an element-wise application of the 4-state carry function to all possible 2 4-state element words. And then convert each 2 4-state element word to its decimal representation.

This approach does NOT make a difference if one starts with the same bit-wise AND operation. Doing a 4-bit word AND or first a 2-bit word AND followed by a 2 4-state word operation will create the same result, of course.

The following tables in Figures 17 and 18 illustrate the effect of extensions. Figure 17 is the canonical carry table of the 16-state carry function car16 related to the bitwise XOR car4 expressed as addition over GF(16).

car16	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
2	0	0	2	2	0	0	2	2	0	0	2	2	0	0	2	2
3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3
4	0	0	0	0	4	4	4	4	0	0	0	0	4	4	4	4
5	0	1	0	1	4	5	4	5	0	1	0	1	4	5	4	5
6	0	0	2	2	4	4	6	6	0	0	2	2	4	4	6	6
7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7
8	0	0	0	0	0	0	0	0	8	8	8	8	8	8	8	8
9	0	1	0	1	0	1	0	1	8	9	8	9	8	9	8	9
10	0	0	2	2	0	0	2	2	8	8	10	10	8	8	10	10
11	0	1	2	3	0	1	2	3	8	9	10	11	8	9	10	11
12	0	0	0	0	4	4	4	4	8	8	8	8	12	12	12	12
13	0	1	0	1	4	5	4	5	8	9	8	9	12	13	12	13
14	0	0	2	2	4	4	6	6	8	8	10	10	12	12	14	14
15	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Figure 17

The table of Figure 18 below illustrates the 16-state extension table cap16 of the 4-state table cap4 of Figure 14.

cap16	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	15	15	14	15	15	15	14	15	11	11	10	11	15	15	14	15
1	15	14	14	14	15	14	14	14	11	10	10	10	15	14	14	14
2	15	15	13	12	15	15	13	12	11	11	9	8	15	15	13	12
3	15	14	13	13	15	14	13	13	11	10	9	9	15	14	13	13
4	15	15	14	15	11	11	10	11	11	11	10	11	11	11	10	11
5	15	14	14	14	11	10	10	10	11	10	10	10	11	10	10	10
6	15	15	13	12	11	11	9	8	11	11	9	8	11	11	9	8
7	15	14	13	13	11	10	9	9	11	10	9	9	11	10	9	9
8	15	15	14	15	15	15	14	15	7	7	6	7	3	3	2	3
9	15	14	14	14	15	14	14	14	7	6	6	6	3	2	2	2
10	15	15	13	12	15	15	13	12	7	7	5	4	3	3	1	0
11	15	14	13	13	15	14	13	13	7	6	5	5	3	2	1	1
12	15	15	14	15	11	11	10	11	7	7	6	7	7	7	6	7
13	15	14	14	14	11	10	10	10	7	6	6	6	7	6	6	6
14	15	15	13	12	11	11	9	8	7	7	5	4	7	7	5	4
15	15	14	13	13	11	10	9	9	7	6	5	5	7	6	5	5

Figure 18

One can see the differences between the two 16-state functions. One remarkable difference is that the function of Figure 18 has only 1 situation wherein no carry digit or rather a carry digit 0 is generated. This means that in almost all situations one will have carry propagation. The above functions may be extended to 256-state functions, and so on.

One may use a similar approach for the carry function of modulo-n based addition. But, it has to be kept in mind that this will only work with an extension of the modulo-n addition. The

extension of an addition over GF(2) with a factor k is of course per definition the addition over GF(2^k). However, the addition over GF(n=5^2=25) is different from the addition modulo-25.

8. Other Methods of Modifications of Reversible N-state Carry Functions

A reversible n-state carry function herein is one for which an n-state borrow function exists in combination with an n-state subtraction that reverses the radix-n addition with the n-state carry function. It does specifically NOT mean that the carry function itself is reversible, which it is usually not.

Several methods for constructing reversible n-state carry functions from found reversible n-state carry functions have been developed. Extending functions is one way. But other modification methods have been applied. These methods, applied in cryptographic circuitry, are taught in pending US Patent Applications.

The purpose of this article is to show that numerous modifications of reversible n-state carry ripple addition computer implementations can be created. It is not intended to teach all methods of modification. However, feel free to contact the author at info at labcipher dot com if you want to learn more about these methods.

The Finite Lab-Transform (FLT) Modification

The number of possible modifications of reversible n-state carry functions becomes quickly very large for n=8 and n=16 and greater. Even for n=8 that number may be greater than a standard computer can generate in a reasonable time. This is of importance if one applies the modifications in cryptography. A dedicated attacker with unlimited or close to unlimited computer power may be able to generate many if not all possible 16-state modifications. This puts a cryptographer at a disadvantage. It is still unlikely that an attacker will reasonably be successful at a brute force attack, but the risk is not entirely zero.

Application of the Finite Lab-Transform or FLT [2] and [8] can overcome that risk. The FLT is an n-state transformation that transforms the numerical appearance of an n-state function but preserves its meta-properties. The number of possible transformations is a factor related to the factorial of n (n!). It depends somewhat on the function that is transformed, but in all cases is at least (n-3)!. For n=256 or 8-bit representation that means a factor greater than 10^400. It is an immense number which renders it impossible to find the selected FLT with all computer power in the world during the life-time of our universe.

The factorial factor has as a strange effect that for small n, like n=4 the numbers are limited. For n=4 n!=24, and one may say: “so what?” But for n=8 that number is already 8!=40,320 and for n=16 one has 16!= 20,922,789,888,000 variations.

Application of the FLT to a modified radix-n addition/subtraction is simple. Apply the FLT to all functions, the n-state addition, the n-state carry, the n-state subtraction and the n-state borrow functions and perform the operations as one would do in un-FLTed form.

As an example, use the 4-state functions as shown in Figure 14. Using the 4-state inverter inv4=[3 2 1 0], one gets the functions by FLT as shown in Figure 19.

sn4	0	1	2	3	can4	0	1	2	3	bon4	0	1	2	3
0	3	2	1	0		2	2	1	0		0	1	0	0
1	2	3	0	1		3	2	0	0		1	1	1	0
2	1	0	3	2		1	1	1	0		3	2	0	0
3	0	1	2	3		0	1	0	0		2	2	1	0

Figure 19

Using as input inn=[3 3 3 3] and key=[0 0 0 1] using the above functions one gets as ciphertext: cipt=inn+key = [1 2 2 1]. One can recover inn from inn=cipt-key. It is to be understood that '+' and '-' in the expression mean the modified functions. The ciphertext using the functions of Figure 14 would result in cipher text [2 1 1 2].

This illustrates how one can generate different ciphertext (and recover clear text) by using a modified radix-n addition, followed by an FLT of the functions.

9. Application in Cryptography

Why this modification of radix-n addition? One reason is to make the data flow numerically more unpredictable, while maintaining the proven and tested data flow of existing cryptographic methods, and dramatically increasing security against for instance Quantum Computer attacks.

One may apply the modified radix-n addition in several parts of well-established encryption protocols, of which AES-GCM (Advanced Encryption Standard-Galois Counter Mode) and ChaCha20, both part of TLS 1.3 are among the most widely used.

Advanced Encryption Standard (AES)

The Advanced Encryption Standard (AES) has several operational modes. These modes have an encryption side to generate cipher text and a decryption mode to recover the clear text from the cipher text.

Most of the AES modes have a forward implementation Cipher() and an inverse implementation InvCipher(). There are 4 major modules in AES Cipher():1) SubBytes() applies a substitution table (S-box) to each byte. 2) ShiftRows() shifts rows of the state array by different offsets. 3) MixColumns() mixes the data within each column of the state array. and 4) AddRoundKey() combines a round key with the state.

AddRoundKey() is a bitwise XOR of a state array with a round key array. It does so by bitwise XORing the columns of 4 bytes of the arrays. See AES [3] section 5.1.4. A bitwise XOR of bytes may be described as an addition over GF(2^8=256). One may then describe the bitwise XORing of words of 4 bytes as a carry-less addition of a word of 4 256-state elements radix-256.

This addition is reversed in InvCipher() for decryption, using the same round key, but now using as state array, the one created in the corresponding AddRoundKey() operation in the encryption in Cipher().

Because the addition over GF(256) is an involution (or self-reversing operation), AddRoundKey() in InvCipher uses the same instructions as the AddRoundKey() in Cipher().

Reversibility

The addition can be modified into a reversible addition with carry radix-256 with 4 256-state elements. Let's use the above functions as illustrative examples. Use inn=[41 123 204 177] as a 256-state representation of a word of 32-bits and key=[21 121 221 232]. Using the bitwise XOR approach with no carry, the result is sux=[60 2 17 89]. Using car4 as displayed in Figure 14, and extending it 4-state wise by representing a 256-state element as 4 4-state elements and thus do a 4-state element wise application of car4 one gets a novel carry function car256 corresponding to sc256 the addition over GF(256).

Applying this carry function on the radix-256 addition of inn and key provides as result sux=[204 159 10 89]. One can see that the last element is the same as the unmodified addition, because no carry was induced at the last element in a classical radix-n addition. But this may be changed as discussed earlier above. Because the element 0 also induces a carry, one may create a starting carry from 0s added to both inn and key, but which are (of course) not used as part of the cipher text.

The example has used an addition of 4 256-state elements. However, the AES state array consists of 16 bytes. Accordingly, one may also perform a radix- 256 addition of 16 bytes or in the alternative 2 additions of 8 bytes. An advantage of using more elements is a broader propagation of carry values, but at a cost of a longer ripple.

The above example is provided because it has a corresponding 256-state reversing borrow function. The InvCipher part requires a reversing function. It has been tested in a program and it works fine.

AES-256 for instance, has 14 rounds. Because of the avalanche effect one does not really need to modify AddRoundKey() in all 14 rounds. Probably, a modification in a single round should be sufficient to generate a completely different ciphertext.

Other Reversibility Applications

Popular encryption methods AES-GCM and ChaCha20 apply bitwise XOR in a final stage of encryption. These methods generate a keystream which is bitwise XORed with the cleartext to create ciphertext. In that case one may replace the bitwise XORing of bitstreams with the modified radix-n operations as described above.

Irreversible Applications

The AES part of AES-GCM [4] is actually a one-way application of AES to purely generate a secret keystream. The secret keystream is the same in encryption and decryption mode. Decryption is achieved by the earlier mentioned bitwise XOR. So only the Cipher() part of AES is applied and no InvCipher() is required. (One may of course also apply InvCipher() to generate the keystream.) This dictates that the AES part should be repeatable for encryption and decryption, but no requirement exists that the AES part should be reversible.

Sum-Space

The above necessitates the introduction of, what is herein called, the sum-space of an addition or more generally of an operation. The n-state sum-space of an operation may be defined as the totality of sums of c=a⊕b wherein ⊕ is an n-state operation. For ⊕ being the addition modulo-n or the addition over GF(n=2^k) the sum-space is the set of all possible n-state elements and the distribution of the elements is uniform and flat. That is, no bias exists related to a particular outcome. That means for instance, that if no bias exists in the original AddRoundKey() operation, then no bias will be introduced by applying an alternative function with the same sum-space as the bitwise XORing. The sum-space of a conventional modulo-n ripple addition is called a complete sum-space.

Without direct proof but based on programming examples, it is asserted that with any n-state carry function (including non-reversible n-state carry functions) a combination with an n-state reversible function in radix -n ripple carry mode, has a uniform (non-biased) complete sum-space.

Continue Irreversible Applications

AES-GCM

In AES-GCM one may then modify for instance the n-state function in AddRoundKey() from a carry-less addition over GF(2^k) to a ripple carry radix-n addition still using a function described by addition over GF(2^k) but in combination with a random (any random) n-state carry function. One merely has to make sure that the same n-state carry function is applied for both encryption and decryption. Again, in this situation there is no need for reversibility.

The total number of for instance possible 256-state carry functions is immense. (256^(256^2)). Certain conditions may be pre-set for such functions, for instance that no 0 carry element is generated or that at least 100 different 256-state carry states are applied. And so on.

ChaCha20

ChaCha20 [5] is another encryption method that is widely used and part of TLS 1.3 [6]. Similarly, as in AES-GCM, the main part of ChaCha20 is to generate a secret keystream that is identical for encryption and decryption.

ChaCha20 has 20 so called quarter-rounds operated on a state array of 16 * 32 bits. Each quarter-round includes a bitwise XOR operation on words of 32 bits and an addition of 2 words of 32-bits modulo-2^32. The quarter round expressions are provided and explained in section 2.1 of RFC 7539 [5].

Both operations may be modified as described above. For instance, by processing each block of 32-bits as 4 256-state elements and by performing a 256-state ripple adder operation using any useful 256-state carry table. As with AES, it is not needed to modify all operations, but for

example merely 1 or several quarter-rounds. The internal strength of the avalanche effect of the quarter-rounds guarantees a complete change in output due to the modification.

SHA-256

Virtually all hashing methods use at some point bitwise XORing, or a description by an addition over GF(2^k). One may modify this function as explained above. This creates a customization of most popular hashing methods, including SHA-256 [7]. SHA-256 applies bitwise XORing as defined in FIPS 180-4 section 4.1.2 expressions (4.4)-(4.7) on words of 32 bits. As explained earlier one may break-up each bitwise XORing of words of 32 bits into a radix-256 carry ripple addition with a custom 256-state carry function, which needs NOT be reversible. It is not needed to do this for all expressions, nor is it required to do this for all rounds of SHA-256 to create an entirely different but still repeatable and valid hash value.

SHA-256 also applies an addition modulo-2^32, as defined in section 6.2.2 of FIPS 180-4, which may also be modified as explained above. This creates a strong and privatized hashing method that may be used to generate one-step exchange of private keys.

10. Conclusions

Security of cryptographic machines is usually obtained from two aspects:
1) a set of well described operations that in combination create an output that is intractable to be inverted to an input operand; and
2) at least one operand that is so large that brute force attacks in the context of the combined operations are infeasible.

A different approach is used herein. The data-flow as in standard cryptographic operations is maintained, but certain functions are replaced. This results in a data-flow that has the same structural properties (and at least the security) as the unmodified cryptographic method, but with an entirely different numerical output. This allows for customization of cryptographic methods and inherently increases its security against attacks. It appears that the modifications are resistant against Quantum Computer attacks.

The modifications in the above case are based on novel n-state carry functions that preserve the reversibility of a radix-n carry ripple-like machine addition, but with a different numerical output than a standard canonical addition.

11. License to Use the Modified Radix-n Machine Operations for Limited Purposes

Certain aspects of the above machine operations are claimed in pending USPTO Patent Applications. The applications are assigned to LCIP jv. LCIP jv provides explicit license to use the claimed invention for trial, research and educational purposes only. Any use of the above modified machine operations in cryptography is explicitly not part of this license. This specifically pertains to operational data encryption and hashing in operational storage and/or exchange of data.

We are well aware of disapproval and/or concern of certain groups about the assertion and/or use of Intellectual Property Rights. So be it. Some are not aware of the IPR exception of the Creative Commons Attribution 4.0 International License. You are hereby notified of that exception.

Permission and license for operational use can only be obtained by written permission of Peter Lablans. Such license may require a reasonable royalty or license fee. Please contact Peter Lablans at info at labcipher dot com for further information.

References

[1] Gerrit A. Blaauw, Digital System Implementation, 1976, Prentice-Hall, Inc., Englewood Cliffs, NJ

[2] Peter Lablans, The Finite Lab-Transform (FLT) for Invertible Functions in Cryptography, 2024 at https://engrxiv.org/preprint/view/3570/6377

[3] National Institute of Standards and Technology (NIST). (2001, November). Advanced Encryption Standard (AES) (Federal Information Processing Standard (FIPS) 197). https://csrc.nist.gov/pubs/fips/197/ipd

[4] National Institute of Standards and Technology (NIST). (2001, November). Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC for Advanced Encryption Standard (AES) (Special Publication 800-38D). https://csrc.nist.gov/pubs/sp/800/38/d/final

[5] ChaCha20 and Poly1305 for IETF Protocols, Request for Comments: 7539, May 2015, https://datatracker.ietf.org/doc/html/rfc7539

[6] The Transport Layer Security (TLS) Protocol Version 1.3 at https://datatracker.ietf.org/doc/html/rfc8446

[7] National Institute of Standards and Technology (NIST). (2015, August). Secure Hash Standard (SHS) (Federal Information Processing Standard (FIPS) 180-4). https://nvlpubs.nist.gov/nistpubs/fips/nist.fips.180-4.pdf .

[8] The Finite Lab-Transform (FLT), a website https://www.labtransform.com/

Biography:

Peter Lablans received a B.Sc. and M.Sc. (Ir.) degrees in electrical engineering from the Technische Hogeschool Twente in Enschede, The Netherlands. He was a student of Prof. Dr. G.A. ("Gerry") Blaauw, one of the 3 co-architects of the legendary IBM System/360, who greatly influenced Lablans' thinking about computing machines. Lablans is a prolific inventor and is the named inventor of over 50 US Patents. He lives in New Jersey.