Novel N-state Commutative (Full) Involutions NOT Being Additions Over GF(2^k)

Applications in Cryptography

Peter Lablans – LabCipher

Abstract

Applications of full commutative n-state involutions are ubiquitous in standard cryptographic operations in current encryption and hashing. The full involutions are generally implemented as bitwise XOR of words of k bits and may be generally described as additions over GF(2^k).

Novel n-state full involutions are described herein, including methods how to generate them. The novel n-state full involutions can NOT be described as additions over GF(2^k). Application of these n-state involutions creates a level of uncertainty for attackers and increases security of encryption and hashing when replacing conventional involutions.

Application of the Finite Lab-Transform (FLT) on these novel involutions maintains their self-reversing properties and further increases security. Examples of use of novel involutions in encryption (AES-GCM,) and hashing (SHA-256) are provided.

Key Contributions:

- Defines canonical n-state full 2-operand involutions in reversible computer operations -
- Provides modified versions of n-state full 2-operand involutions that are not additions over GF(2^k)
- Increases overall security of involutions by applying the Finite Lab-Transform (FLT)
- Applies the involution modifications in machine cryptography

Keywords: n-state involutions, 2-operand involutions, self-reversing, n-state commutative involution, n=2^k, extensions by element-wise function application, self-reversing n-state inverters, finite fields, GF(2^k), Finite Lab-Transform (FLT), 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 ← a∧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.

 © Copyright 2024, Peter Lablans

A straightforward table-based notation is applied herein, like 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. Full and Partial Involution

A full n-state involution is a commutative self-reversing computer operation herein and is called a full n-state involution or an n-state commutative involution. The best known involution function is the (binary) XOR operation, of which the look-up table sc2 is shown in Figure 1.

sc2	0	1
0	0	1
1	1	0

Figure 1

This is a full involution because: if c=sc2(a,b), then a=sc2(c,b) and a=sc2(b,c) as the function sc2 is commutative. The involution may also be called a self-reversing function.

In cryptography (and in other applications like error-correcting computer coding) often words of k bits are XORed bitwise. This may be described as an addition over finite field GF(n=2^k). In Figure 2 a look-up table of the addition over GF(2^2=4) is shown as table sc4. The table sc4 is achieved by using words of 2 bits and XOR all possible 2 words of 2 bits and display the result in a decimal integer form. The function sc4 is a full or 4-state commutative involution.

sc4	0	1	2	3
0	0	1	2	3
1	1	0	3	2
2	2	3	0	1
3	3	2	1	0

Figure 2

Partial Involution

An involution function is a partial involution if the function is non-commutative. For instance the function expressed as dif=key-clt is a partial involution. In that case clt=key-dif and only for that arrangement the function is self-reversing. Figure 3 shows the modulo-5 function min5 is dif=row-column as a partial involution.

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 3

 © Copyright 2024, Peter Lablans

Addition over GF(2^k)

The addition over GF(2^k) has some attractive properties, like being commutative, being associative, having an additive inverse and having a corresponding function (commonly called a multiplication) that distribute over GF(2^k). On that basis, the n-state function for n=2^k and using as base function XOR, is what is called "a law of composition" of a finite field GF(n).[13]

3. The Reversible N-state Inverter

An important concept in non-binary logic operations herein is the reversible n-state inverter. Its notation is as a function y=inv(x) wherein x, y ∈ $\mathbb{Z}$n. One representation of the n-state inverter is provided by its transformation from input to output states. The input states are all possible input states, represented by an index number starting at 0. So, input= [0 1 2 3 4 … n-1]. And thus input(0)= 0, input(3)=3 and input(n-1)=n-1. This is also explained in an earlier website [8].

Using a 4-state inverter inv4, we may have an inverter transformation input→output as [0 1 2 3] → [3 0 1 2]. Because the input is always input= [0 1 2 … n-1], with predetermined indices, one may also say: inv4= [3 0 1 2]. Because of the above convention one knows that inv4(0)=3, inv4(1)=0, inv4(2)=1, inv4(3)=2.

Each reversible n-state inverter invn has its reversing n-state inverter rinvn, with as property that invn(rinvn(i)) = i and rinvn(invn(i)) = i.
The reversing 4-state inverter rinv4 of the above inverter inv4 is then: rinv4= [1 2 3 0].

The above is explained in mathematical terms. In programming such as APL and C, it relates to array indexing starting at origin 0.[14] Matlab starts array indices at origin-1 and statements like rinv4(0) = 1 will generate an error message. Using indexing starting at origin-1 is by itself not a problem as one may increase all inverter states with 1, as in Matlab. But it requires careful management of inverter statements.

Figure 4 illustrates an n-state inverter in a diagram. An n-state inverter herein is a machine operation, not a mathematical concept.

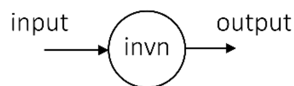

Figure 4

Program-wise, an n-state inverter is (in APL or Matlab terms) a one-argument or monadic operation. It is represented by a one-dimensional table or array. Because numerical and symbolic notation are used, some people may want to maintain that the machine operations herein are mathematical operation. This is expressly and explicitly not the case. A computer or machine does not "know" what it is processing. There are no 0s and 1s in a computer for instance. But only gates that are placed in physical state HIGH or LOW. For CMOS technology voltage levels between 0 and 1.5 Volt may be applied as LOW and between 3.5 V and 5 V as HIGH. For a

© Copyright 2024, Peter Lablans

basic explanation on how to describe physical states of a device with human labels like 0 or 1 is explained in Claude Shannon's 1938 Master Thesis [9].

There are n^n different n-state inverters, ranging from [0 0 … 0] to [(n-1) (n-1) … (n-1)]. There are factorial of n or (n!) different reversible n-state inverters, including identity. For n=3 one has as 3! = 6 reversible inverters: [0 1 2]; [0 2 1]; [1 0 2]; [1 2 0]; [2 0 1]; and [2 1 0].

The 3-state inverter [0 1 2] or in general invn = [0 1 2 … n-1] is the identity or identity inverter.

Furthermore inv3 = [1 0 2] has as reversing inverter rinv3 = [1 0 2] and inv3=rinv3 and is called self-reversing.

4. Self-reversing N-state Inverters

A self-reversing n-state inverter is a reversible inverter for which the reversing inverter is the same as the inverting inverter: or invn=rinvn.

For n=2 there are 2 self-reversing inverters. 1) inv21=[0 1], which is identity and 2) inv22=[1 0].

For n=4 there are 10 self-reversing 4-state inverters, including identity. The list of 4-state self-reversing inverters is shown in Figure 5.

self-reversing 4-state inverters
[0 1 2 3]→[0 1 2 3]
[0 1 2 3]→[0 1 3 2]
[0 1 2 3]→[0 2 1 3]
[0 1 2 3]→[0 3 2 1]
[0 1 2 3]→[1 0 2 3]
[0 1 2 3]→[1 0 3 2]
[0 1 2 3]→[2 1 0 3]
[0 1 2 3]→[2 3 0 1]
[0 1 2 3]→[3 1 2 0]
[0 1 2 3]→[3 2 1 0]

Figure 5

By analyzing the n-by-n lookup tables of an n-state full involution, one observation is that all rows in the involution representing n-state table are self-reversing. This is sufficient for a function to be at least a partial involution. For a full involution it dictates that the kth column of the representing table has to be identical to the kth row of the table, because the table has to be commutative.

© Copyright 2024, Peter Lablans

An n-state involution may be represented by an n-by-n table. If one keeps a counter (which may be a modulo-10 counter) one may create a partial involution where for each element in a series of elements a pre-defined self-reversing n-state inverter is applied. In that case one may apply any mix of n-state self-reversing inverters, with or without repeat. One may randomly create a long sequence of numbers selected from 10 numbers for the 4-state case and each element in the sequence indicates the use of a particular 4-state self-reversing inverter. Other arrangements are fully contemplated.

However, in general, in an n-state operation a function represented by an n-by-n table is preferred. Applying the above, a series of 4-state full involutions have been constructed. A limitation herein is that kth rows and columns have to be identical. For n=4 one may do that manually, but preferably a computer program is applied. In fact, a Matlab program has been created to go through all possible n-by-n tables constructed from available n-state self-reversing inverters, wherein each row is different, and each kth column is identical to the kth row wherein k ranges from 1 to n in Matlab or from 0 to n-1 in a zero-origin computer language like C.

One may facilitate the process by creating sets of self-reversing n-state inverters based on their first element. In the 4-state case that means Group 0-start {[0 1 2 3], [0 1 3 2], [0 2 1 3] and [0 3 2 1]}; Group 1-start {[1 0 2 3], [1 0 3 2]}; Group 2-start {[2 1 0 3], [2 3 0 1]}; and Group-3 start {[3 1 2 0], [3 2 1 0]]}. One may always start with a first row being selected from the Group 0 start. Because of commutativity, this determines automatically from which groups the next inverters have to be selected. However, this is done for programming convenience and one may start with any appropriate self-reversing n-state inverter.

A simple exhaustive procedure is to select from each Group an inverter that has no elements in common in any column position with earlier selected inverters above.

Doing this manually for n=4:
1) select inverter [0 1 2 3] from Group 0, this tells that the next inverters have to be selected from Group 1, Group 2 and Group 3.
2) select [1 0 2 3] from Group 1. This inverter in row 1 has element 3 in common in column 3 with the selected inverter in row 0 and should be disqualified. Skip and select [1 0 3 2] in Group 1. This inverter has no elements in common in corresponding column positions with the inverter in row 0. Move on to:
3) select [2 1 0 3] in Group 2. This inverter is disqualified because of element 3 in column position 3. Skip to inverter [2 3 0 1] in Group 2. This works and move to:
4) select [3 1 2 0] in Group 4. This does not work and try [3 2 1 0] and this works.

By the above steps the 4-state involution sc4 of Figure 2 has been constructed.

Using the same procedure, but starting with different self-reversing 4-state inverters from Group 0 one may create the following 4-state full involutions as shown in Figure 6.

 © Copyright 2024, Peter Lablans

sc4	0	1	2	3	sn4	0	1	2	3	sp4	0	1	2	3	sr4	0	1	2	3
0	0	1	2	3		0	1	3	2		0	2	1	3		0	3	2	1
1	1	2	3	0		1	0	2	3		2	3	0	1		3	2	1	0
2	2	3	0	1		3	2	1	0		1	0	3	2		2	1	0	3
3	3	0	1	2		2	3	0	1		3	1	2	0		1	0	3	2

Figure 6

Of the above full involutions only sc4 is associative and can possibly be an "addition" function in a finite field GF(4). The other functions (sn4, sp4 and sr4), may be used as a replacement involution of sc4. However, they are not usable for defining a Finite Field (or as a "law of composition" formally) [13]. Clearly, the outcome of using such novel involution functions instead of sc4 will provide a different ciphertext from a cleartext message.

It may seem unlikely, under strict limitation of being commutative, that sufficient self-reversing inverters exist to generate different n-state commutative involutions. But they actually do exist.

A rule that creates a novel n-state full involution from an existing n-state full involution from its n-by-n table is as follows.

An n-state full involution n-by-n table has n rows, each row being an n-state self-reversing invertor.
1) select one of the n n-state self-reversing inverters as the first or top row of the new n-by-n table.
2) because of the requirement of an n-state full involution being commutative, the first **column** of the new n-by-n table is the same as the first **row**.
3) complete each row by selecting from the old table the row that starts with the first n-state element of the new row as set by the commutative requirement
4) the new n-by-n table is also an n-state full involution.

Accordingly, one can create n-1 new n-state full involution tables from a given n-state full involution table.

As an example use as start the table of sn4 in Figure 6 and create the other 3 4-state tables as shown in Figure 7.

sn4	0	1	2	3	sn41	0	1	2	3	sn42	0	1	2	3	sn43	0	1	2	3
0	0	1	3	2		***1***	***0***	***2***	***3***		***3***	***2***	***1***	***0***		***2***	***3***	***0***	***1***
1	1	0	2	3		***0***	1	3	2		***2***	3	0	1		***3***	2	1	0
2	3	2	1	0		***2***	3	0	1		***1***	0	2	3		***0***	1	3	2
3	2	3	0	1		***3***	2	1	0		***0***	1	3	2		***1***	0	2	3

Figure 7

This is a highly effective way to generate quickly a potentially large set of different n-state full involutions. However, applying a limited FLT [2] would also generate the above functions.

 © Copyright 2024, Peter Lablans

A disadvantage being that if a cryptography designer can do it, then so can an attacker.

5. The N-state Extension Tables

There are additional ways to extend an existing n-state involution table to a higher p-valued one. Again, based on having at least one n-state commutative involution, an n-state FLT (for instance as explained in https://engrxiv.org/preprint/view/3570) would create close to factorial n (n!) different variations of such an involution. This may be beneficial for larger values of n, like n=16 or n=32 where there may be too many self-reversing n-state inverters to rapidly find n-state commutative involutions not being a ‘law of composition” over GF(n).

Extending by Element Wise Application of Base Table

A first way is to use a base table as an n-state n-by-n look-up table as and use n-state elements as base elements. Form a new p-state word, for instance from 2 n-state elements. Next, create all possible p-state words from 2 n-state elements. Then perform the n-state element-wise combination of all possible 2 n-state words with the n-state involution function. The result will be for each combination a 2 n-state element word. Convert the 2 n-state element word to its n^2 p-state equivalent.

If the base n-state n-by-n table is a full involution then the p-state extension with p=n^k is a p-state p-by-p full involution table.

This is demonstrated by applying table sn4 of Figure 6 as a base table and extend it to a 16-state table as explained above, by creating all words of 2 4-state elements, combine the words with sn4 into resulting 2 4-state words and convert all resulting 2 4-state elements to a 16-state element. This will create the table sn16 as shown in Figure 8.

sn16	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	1	3	2	4	5	7	6	12	13	15	14	8	9	11	10
1	1	0	2	3	5	4	6	7	13	12	14	15	9	8	10	11
2	3	2	1	0	7	6	5	4	15	14	13	12	11	10	9	8
3	2	3	0	1	6	7	4	5	14	15	12	13	10	11	8	9
4	4	5	7	6	0	1	3	2	8	9	11	10	12	13	15	14
5	5	4	6	7	1	0	2	3	9	8	10	11	13	12	14	15
6	7	6	5	4	3	2	1	0	11	10	9	8	15	14	13	12
7	6	7	4	5	2	3	0	1	10	11	8	9	14	15	12	13
8	12	13	15	14	8	9	11	10	4	5	7	6	0	1	3	2
9	13	12	14	15	9	8	10	11	5	4	6	7	1	0	2	3
10	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
11	14	15	17	13	10	11	8	9	6	7	4	5	2	3	0	1
12	8	9	11	10	12	13	15	14	0	1	3	2	4	5	7	6
13	9	8	10	11	13	12	14	15	1	0	2	3	5	4	6	7
14	11	10	9	8	15	14	13	12	3	2	1	0	7	6	5	4
15	10	11	8	9	14	15	12	13	2	3	0	1	6	7	4	5

Figure 8

© Copyright 2024, Peter Lablans

One may use a simple computer program to check that sn16 is indeed a 16-state full involution table.

The above method is powerful to create higher-state involutions. However, the steps are big. One only can get from 4-state to 16-state but not to 8-state involutions from a 4-state base involution. One could arrive at a 16-state involution by using words of 4 bits and doing a bitwise XOR. But clearly, the number of created 16-state involutions would be limited.

Extending by Doubling

There are two useful novel rules related to self-reversing n-state inverters. Both rules relate to creating a self-reversing 2*n state inverter from an n-state self-reversing inverter.

Rule 1: In Matlab notation. If one has an n-state inverter invn, then form a second inverter invnb=invn+n. (remember, all arrays in Matlab are origin-1). Then for k=2*n create invk=[invn invnb]. Or, one modifies the original inverter by adding n to all elements and make this set of elements the second half of the k-state inverter. If the original n-state inverter is self-reversing then the newly created k-state inverter is also self-inverting.

An example: use 4-state self-reversing inverter inv4= [3 4 1 2] in origin-1. Then inv4b is formed by adding 4 or inv4b=[7 8 5 6] in origin-1. The new 8-state inverter is then inv8=[3 4 1 2 7 8 5 6] and is also self-reversing.

This doubling takes care of the first half of required rows of the p-state commutative involution.

Rule 2: Take above invn and invnb and create invkb= [invnb invn]. The newly formed k-state inverter invkb is also self-reversing.

Doubling rule: One may, based on the above 2 rules, form a k-state full involution with k=2*n by taking the original table scn, form the extended n rows by rule 1 of above applied to each row in scn and then apply rule 2 to all newly formed rows to now have n new rows. Combining the two sets of n rows creates a k-state full involution when the original scn was a full involution.

An advantage of this method that one may create novel involutions that are a linear factor bigger than the original ones rather than a power of n.

sn8	0	1	2	3	4	5	6	7
0	0	1	3	2	4	5	7	6
1	1	0	2	3	5	4	6	7
2	3	2	1	0	7	6	5	4
3	2	3	0	1	6	7	4	5
4	4	5	7	6	0	1	3	2
5	5	4	6	7	1	0	2	3
6	7	6	5	4	3	2	1	0
7	6	7	4	5	2	3	0	1

Figure 9

© Copyright 2024, Peter Lablans

The above example in Figure 9 by applying these rules illustrates how an 8-state full involution function sn8 is created from table sn4 in Figure 6 by applying the doubling rule.

This opens an additional way to create n-state full involutions. It also increases the total number of different involutions that can be created. When applied in machine cryptography, the total number of different n-state full involutions will be significant, making successful attacks after applying the modifications unlikely.

6. The Finite Lab-Transform (FLT) Modification

The number of different n-state full involution increases significantly with the above extension methods. Especially if one uses 256-state or larger-state involutions. However, a committed and dedicated attacker may make an inventory or library of possible modifications. Such a library may become unfeasible perhaps for n=16 but almost certainly for n=32 or greater.

An additional and secure modification that places the possible number of modification outside the feasibility of brute force attack is the application of the Finite Lab-Transform or FLT [2] and [8].

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 believed to be 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 for attackers to find the selected FLT, even with all computer power in the world during the life-time of our universe.

The factorial factor has as 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.

As an example, use the 4-state functions as shown in Figure 6. Using the 4-state inverter inv4=[3 2 1 0], one gets the functions by FLT as shown in Figure 10.

sc4f	0	1	2	3	sn4f	0	1	2	3	sp4f	0	1	2	3	sr4f	0	1	2	3
0	2	3	0	1		3	2	1	0		1	0	3	2		2	1	0	3
1	3	0	1	2		2	3	0	1		0	2	1	3		1	0	3	2
2	0	1	2	3		1	0	2	3		3	1	2	0		0	3	2	1
3	1	2	3	0		0	1	3	2		2	3	0	1		3	2	1	0

Figure 10

One can easily see that if cipher=scnf(inn,key) in an encryption, then the above functions all give different ciphertext outputs. The cleartext inn may be retrieved with the same functions such as inn=scnf(cipher,key) in a decryption.

© Copyright 2024, Peter Lablans

This illustrates how one can generate different ciphertext (and recover clear text) by using a modified involution followed by an FLT of the function.

9. Application in Cryptography

Why this modification of involutions? One reason is to make the data flow numerically more unpredictable, while maintaining the proven and tested data flow of existing cryptographic methods with previously known functions such as additions over GF(n=2^k). It is believed that it dramatically increases security against for instance Quantum Computer attacks.

One may apply the modified involution in several parts of well-established encryption protocols, of which AES-GCM [4] (Advanced Encryption Standard-Galois Counter Mode) and ChaCha20 [5], both part of TLS 1.3 [6] are among the most widely used.

Advanced Encryption Standard (AES)

The Advanced Encryption Standard (AES) [3] 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 and is a commutative involution. 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 modify the bitwise XORing of words of 4 bytes as explained above.

Other Reversibility Applications

Popular encryption methods AES-GCM [4] and ChaCha20 [5] 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 involution machine operations as described above.

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. The quarter round expressions are provided and explained in section 2.1 of RFC 7539 [5].

© Copyright 2024, Peter Lablans

One may apply a modification, for instance, by processing each block of 32-bits as 4 256-state elements and by performing a 256-state modified involution as explained above. 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 modified 256-state involution. 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.

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 by specifically modified ones. In this article well known n-state commutative involutions have been replaced with novel n-state commutative involutions. 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 involution functions that preserve the self-reversibility, but with a different numerical output than a standard bitwise XORing.

One is cautioned not to use the modified n-state involutions that are not characterized as being associative. In that case they are not additions over GF(2^k) and have not a corresponding n-state multiplication.

***

 © Copyright 2024, Peter Lablans

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/

[9] Claude Shannon, A symbolic analysis of relay and switching circuits. MIT Thesis, 1938, available at https://dspace.mit.edu/handle/1721.1/11173

[10] US Patent Ser. No. 11,336,425B1 to Peter Lablans, entitled Cryptographic machines characterized by a Finite Lab-Transform (FLT), issued on May 27, 2022

[11] US Patent Ser. No. 11,093,213B1 to Peter Lablans, entitled Cryptographic computer machines with novel switching devices, issued on Aug. 17, 2021

[12] US Patent Ser. No. US 10,650,373B2 to Peter Lablans, entitled Method and apparatus for validating a transaction between a plurality of machines, issued on May 12, 2020

[13] Iain T. Adamson, Introduction to Field Theory, 2nd edition, Dover Edition, 2007

[14] Gilman et al. APL/360. An Interactive Approach, 1970, John Wiley & Sons, Inc.

Biography:
Peter Lablans received a M.Sc. (Ir.) degree in electrical engineering from the Technische Hogeschool Twente in Enschede (now "University of Twente'), The Netherlands. Lablans is a prolific inventor and is the named inventor of over 50 US Patents. He lives in New Jersey.

© Copyright 2024, Peter Lablans