Ternarylogic

THE FINITE LAB-TRANSFORM (FLT)

Peter Lablans

One is referred to US Patent Application Publication S/N 20170169735 published on June 15, 2017 for detailed description of the Lab-transform (FLT) and application is certain cryptographic applications. Specifically FIG. 48 provides a screenshot of a Matlab® program listing of a Lab-transform (FLT).

Cryptography is non-binary

Most of our real-world data is non-binary: text, sound and video all apply non-binary representation (be it with words of bits). Text is famously coded as a byte per character. Sound samples are coded as 16 bits per sample. And pixels may be coded as 24-bit words.

It will not come as a surprise that many computer encryption routines apply non-binary switching functions, though largely hidden from the general public. Unfortunately, these functions are generally based on bit-wise XORing of binary words. While called "additions over an extension finite field GF(n=2^k)," they are basically a series of XORs. See for instance the FIPS Standard for AES or Advanced Encryption Standard. Another cryptography application that operates on bytes or words of bytes is the hashing or message digest standard SHA-2. Other computer cryptography applications that use non-binary functions are elliptic curve cryptography and public key exchange.

Virtually all computer based cryptographic methods and cryptographic machines are designed using a mathematical explanation. That is a strength, because it enables to expose relations between entities that would otherwise be hard to understand. It is also a weakness, because arithmetic is universally understood and thus enables creating creative attacks on cryptographic machines.

Machine Arithmetic

Computers and electronic calculators are able to perform almost all arithmetical operations so quickly and with such a precision that it seems that doing calculations are an inherent capability of digital machines. But that is not the case. Machine arithmetic is fundamentally a process of applying switching devices to process signals, wherein the signals are assigned a meaning of a symbol. Machine arithmetic on digital machines is fundamentally discrete and finite.

In designing arithmetical circuits descriptive mathematical functions are applied that coincide with the switching functions. The binary XOR function is often described as a mod-2 addition and the AND function as a mod-2 multiplication. This has the beneficial effect that we can design switching processes in terms of relatively easy to understand mathematical symbols.

The negative effect is that people seem to believe that the computer performs arithmetic in a human sense, which is not true. Computers merely process signals. An often made assumption is that a computer implemented "arithmetical" operation has to comply with standard human arithmetical operations and functions. Most cryptographic operations are designed and operated on the basis of standard human "arithmetical functions."

Many of the cryptographic functions are often modulo-n multiplications wherein n is very large. Unfortunately, with improved computer power, n is often not large enough and even larger values of n have to be applied to protect the security of cryptographic methods and machines. For instance 2048 bit keys are now recommended for RSA. The ultimate threat is formed by quantum computers which will (presumably) be able to successfully attack many current cryptographic applications.

Almost every n-state switching function in cryptography has some common properties. 1) every 'addition' has a zero element 0: 0+x=0. Every multiplication has a zero element 0: 0*x=0 and a one element: 1*x=x. This is so common that most people do not even think about it. One could substitute 0 for another number and do the same for 1 also. However, this most likely will break up some important properties of a multiplication and addition: associativity and distributivity, rendering the operations pretty much useless for cryptography.

So, the challenge is to create a cryptographic device wherein properties of machine addition and multiplication are maintained, but the interpretation of signals has changed, and changed in such a way that it has become highly unpredictable what the signals mean and thus how they are being processed by the computer. Enter the Finite Lab-transform (FLT).

The Finite Lab-transform (FLT)

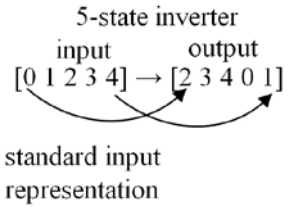
The FLT, hereinafter "Lab-transform", is a discrete and finite machine transformation of one or more n-state switching functions that characterize a switching performance of a switching device that has certain meta-properties such as commutativity, associativity, distributivity, existence of an inverse, etc. , wherein the transformation leaves the meta-properties unchanged while modifying the expression of these properties. For instance a Lab-transformed multiplication still has a multiplicative inverse, but that inverse may no longer be 1. One very unusual result of the FLT may be that 0 is no longer the zero element and 1 is no longer the one-element of an arithmetical operation.

The Lab-transform is applied to switching functions with a finite number of states. Binary switching has 2 states. Operations selected for cryptographic operations are often defined as operations over a finite field. For n being prime, like 3, 5, 7, ... 492,876,863, ... etc. the operations over the finite field (named GF(n), G=Galois and F=Field) the operations are generally the modulo-n addition and modulo-n multiplication.
For n being a power of 2, the operations over GF(2^k) are a bit more complicated, but generally are well known. (the reason for finding an alternative for mod-4 multiplication being that for instance 1*2mod-4 = 2 and 3*2mod-4 = 2, so the operation is not invertible).

N-state Inverters

The Lab-transform of an n-state switching operation applies an n-state inverter. An n-state operation (including an n-state inversion) is assumed to have one of n states as at least one input. An n-state inverter is a single input/single output operation. For convenience, an n-state input is assumed to be able to assume one of the states in [0 1 ... (n-1)]. So an input of a 5-state inverter can have one of the states [0 1 2 3 4]. For convenience the states 0, 1, 2, ... (n-1) are used. This has some advantages in being able to use common algebra in describing operations. However, 0, 1, 2, ... (n-1) are merely labels for signals in a machine and for instance (n-1) may be represented by a word of 3 bits ([100] for 4 for instance, wherein a 0 may be 0.2 Volt and a 1 may be 4.7Volt.

The following figure illustrates a 5-state inverter using states 0, 1, 2, 3 and 4.



The input is an ordered representation of possible input states: [0 1 2 3 4] followed by an arrow indicating inversion to output states. The output following the arrow indicates the output states as a result of input states. the position in the output corresponds to a similar position in the input states as further illustrated by the arrows. So input state 0 is inverted to output state 2, input state 1 to output state 3, etc, and input state 4 to output state 1. The traditional binary inverter is [0 1]→[1 0].

The 5-state inverter may be represented as inv5= [0 1 2 3 4] →[2 3 4 0 1]. Because every inverter is characterized by input states [0 1 ... (n-1)], the input states will be dropped and we will talk about inverter inv5 = [2 3 4 0 1]. The inverter is now a vector of 5 elements.

In functional computer language such as Matlab it is desirable to define the inverter as a vector:

inv5=[2 3 4 0 1].
The actual inversion is realized by executing:
inv5(i), wherein i is one of the input states. So inv5(0) = 2 and inv5(4) = 1.

The above shows it would be nice if a position in [0 1 2 3 4] is the same as the actual state. That is: position 0 is state 0, position 4 is state 4. In computer language it requires a program to run in 'origin 0.' That means that a vector or array starts indexing from position 0. Unfortunately, Matlab works in origin 1.

For operational purposes the inverters will be represented in origin 1. That is: the inverters inv5=[3 4 5 1 2] and the input states are [1 2 3 4 5]. So inv5(1)=3 and inv5(5)=2.

Reversing inverters

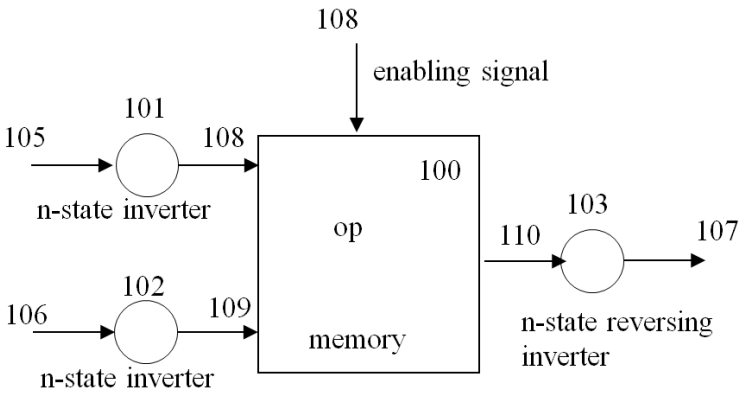
If all n states in an n-state inverter are different, the inverter is reversible. The inverter inv5 is reversible. Assume the name of the inverter that reverses 'inv5' is 'rinv5.' The effect of applying rinv5 after applying inv5 is that the original input state is generated: or rinv5(inv5(i))=i. The reversing inverter rinv5 of inv5=[2 3 4 0 1] is rinv5=[3 4 0 1 2]. One can check that rinv5(inv5(2))=2 (in origin 0).

Number of reversible n-state inverters

There are n! (factorial n) reversible inverters. That may be underwhelming for n=4 with just 1*2*3*4= 24 inverters. However, for n=8 there are in the order of 40,000 reversible inverters and for n=256 (operations on 8-bit words) there are over 10^{100} reversible inverters.

The Lab-transform based on n-state inverters

The following figure illustrates the machine Finite Lab-transform (FLT).



For this embodiment (as it is called in patents) a switching table, for instance a modulo-5 addition or a modulo-5 multiplication is stored in a memory 100. The stored switching table is addressed by two indices, which are the operands or data provided on the inputs, i.e. input 105 and input 106. The input on 105 is inverted by inverter 101, which in the example is inv5, and outputted on 108 which is the new index to 100. The input on 106 is modified by inverter 102 which is also inverter inv5 and outputted on 109. The result of operation 100 (addition or multiplication) is outputted on 110 which is inverted by reversing inverter 103 which in the example is rinv5 and the result is outputted on 107.

The Lab-transform is not intuitive

The counter-intuitive aspect of the Lab-transform is the use of inverters at inputs and the reversing inverter at the output. An immediate reaction may be: that does not do anything: a conversion is applied in one direction and then reversed before a result is generated. It may appear that a net effect is identity. But as the following examples will show: that is definitely not the case.

The 5-state example

The following tables show the switching tables for addition mod-5 and multiplication mod-5.

$+\bmod -5$	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

$*\bmod -5$	0	1	2	3	4
0	0	0	0	0	0
1	0	1	2	3	4
2	0	2	4	1	3
3	0	3	1	4	2
4	0	4	3	2	1

The + and * both have as zero element 0 (x+0 = x and x*0 = 0) and one element 1 (x*1=x).
Applying inv5=[2 3 4 0 1] and rinv5=[3 4 01 2] generates the following Lab-transformed tables:

*sc*5	0	1	2	3	4
0	2	3	4	0	1
1	3	4	0	1	2
2	4	0	1	2	3
3	0	1	2	3	4
4	1	2	3	4	0

*m*5	0	1	2	3	4
0	2	4	1	3	0
1	4	2	0	3	1
2	1	0	4	3	2
3	3	3	3	3	3
4	0	1	2	3	4

The new, Lab-transformed, functions are commutative, associative and distribute. The zero element is 3 and the one element is 4. So we have now a set of 5-state functions that look nothing like the mod-5 operations and that follow laws of algebra and define a finite field aGF(5), which will be called an alternate finite field.

An 8-state example

The following tables are an 8-state addition and multiplication over GF(n=2^3). The finite field is defined by polynomial $x^3 + x + 1$. The addition is formed by bitwise XORing of binary representation of the symbols and the multiplication is done modulo-($x^3 + x + 1$). (in case you want to check).

+GF(8)	0	1	2	3	4	5	6	7
0	0	1	2	3	4	5	6	7
1	1	0	3	2	5	4	7	6
2	2	3	0	1	6	7	4	5
3	3	2	1	0	7	6	5	4
4	4	5	6	7	0	1	2	3
5	5	4	7	6	1	0	3	2
6	6	7	4	5	2	3	0	1
7	7	6	5	4	3	2	1	0

**GF*(8)	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7
2	0	2	4	6	3	1	7	5
3	0	3	6	5	7	4	1	2
4	0	4	3	7	6	2	5	1
5	0	5	1	4	2	7	3	6
6	0	6	7	1	5	3	2	4
7	0	7	5	2	1	6	4	3

Consider an 8-state inverter inv8=[3 6 2 7 1 4 0 5] and its reversing inverter rinv8=[6 4 2 0 5 7 1 3]. Applying the Lab-transform (FLT) to the above tables generates:

*sc*8	0	1	2	3	4	5	6	7
0	6	7	4	5	2	3	0	1
1	7	6	5	4	3	2	1	0
2	4	5	6	7	0	1	2	3
3	5	4	7	6	1	0	3	2
4	2	3	0	1	6	7	4	5
5	3	2	1	0	7	6	5	4
6	0	1	2	3	4	5	6	7
7	1	9	3	2	5	4	7	6

*m*8	0	1	2	3	4	5	6	7
0	7	4	1	2	0	3	6	5
1	4	2	3	5	1	7	6	0
2	1	3	5	7	2	0	6	4
3	2	5	7	0	3	4	6	1
4	0	1	2	3	4	5	6	7
5	3	7	0	4	5	1	6	2
6	6	6	6	6	6	6	6	6
7	5	0	4	1	7	2	6	3

The switching tables are associative, commutative and distribute in combination. The zero element is 6 and the one element is 4.

Rule based FLTs

For extremely large values for n, it may not be practical to store inverters and/or switching tables on memory. Computers can be programmed and circuits like FPGAs

can be configured to perform BigInteger operations. The inverters can be programmed as rules. For instance a BigInteger n-state inverter rule in GF(n) operations can be a multiplication ‘k’ followed by an addition ‘a’. Such a rule is reversible so that the BigInteger reversing rule can also be applied.

Ternarylogic LLC Portfolio

How to apply the Lab-transformed (FLT) switching tables is the subject of patents and patent applications, which are not all published.

The Ternarylogic LLC IP portfolio can be reviewed at www.ternarylogic.com/portfolio.pdf.

Peter Lablans, January 2018