The Finite Lab Transform (FLT) and PKI Based Inverter Distribution

Peter Lablans-December 7, 2025 – info@labcyfer.com

The exceptional expansion of the solution space of symmetric encryption security like AES-GCM and ChaCha20 ($>10^{480}$) is achieved by:
1) application of a simple functional transformation
2) treating computational functions as n-state (n>2) rather than binary
3) benefiting from **factorial/combinatorial explosion**
4) easy distribution of large parameters by known PKI

How big is the solution space " $>10^{480}$ " of the FLT? Even if each atom in our universe (10^{80}) were a quantum computer, running during the lifetime of our universe, you'd still be nowhere near exhausting the solution space of the FLT.

1. Architecture and Implementation

The FLT is a **functional computational transformation** — a transformation of a computer-implemented function.

Its foundation lies in the digital design framework developed by Dr. Gerrit "Gerry" A. Blaauw, who distinguished between:

- **Architecture**: The abstract definition of a system's behavior.
- **Implementation**: The specific realization of that architecture.
- **Realization**: The physical instantiation of the implementation.

Blaauw's purpose was to create a common architecture that preserves defined outputs across different implementations, ensuring compatibility between generations of machines. This principle is now universally applied in computer design, though Blaauw's name is not widely known. He was one of the principal designers of the legendary IBM System/360, arguably the first truly general-purpose computer and an immense commercial success, due to its compatibility design.

FLT applies a "twisted" form of this framework:

- The **architecture** of a cryptographic system (e.g., AES-GCM) is preserved.
- The **implementation** of its functions is modified so that identical inputs ("plaintext") yield **different outputs ("ciphertext")**.

Its development has been largely computational, with minimal reliance on formal mathematical theory. However, since academic cryptography values mathematical rigor, the FLT can be articulated in the language of classical number theory.

1.1 FLT as a Structure-Preserving Conjugation

The FLT is a novel number-theoretical transformation that injects **real-time, high-entropy cryptographic agility** into existing symmetric ciphers (AES, ChaCha20).

The Finite Lab Transform (FLT) is best understood not as a new cipher, but as a **Master Key** that can dynamically change the internal functionality of an existing cipher like AES-GCM, while leaving the wiring unchanged.

Imagine current AES as a massive, complex engine. Every time you run the engine today (encrypt a message), it uses the exact same gears (the implementation) to get the result. Attackers study this static gear structure.

FLT works by introducing a **secret set of high-entropy modifications of core functionality,** that preserve the important properties of the function but modifies its numerical (computational) output.

Cryptographers will recognize the FLT as a structure-preserving **conjugation**—a deep concept from abstract algebra. For non-specialists, a key insight is that the transformations *do not* simply cancel each other out in the context of the cipher's internal mathematics.

1.2 Formal Mathematical Definition

Let:

- g: a base finite-state operation (e.g., addition or multiplication modulo n)
- h: a secret, arbitrary n-state reversible inverter (permutation)
- $S = \{0, 1, \ldots, n\text{-}1\}$: the finite set of states

Then: $f = h^{-1} \circ g \circ h$

For a 2-operand operation $g(a, b) = c$, the transformed operation is: $f(a, b) = h^{-1}(g(h(a), h(b)))$

Key Components:

- g: Original operation (e.g., XOR, addition over GF(n)).
- h: Secret permutation chosen from $n!$ possible maps (the computational inverter).
- h^{-1}: Inverse permutation.
- f: FLT-transformed operation — structurally identical but numerically distinct from g.

1.3 Isomorphism and Automorphism

Because *f* is a conjugation of *g*, it preserves all essential algebraic meta-properties (associativity, commutativity, invertibility).

1. **Isomorphism**
   - The algebraic structure defined by *f* is isomorphic to that defined by *g*.
   - This ensures functional correctness and cryptographic integrity.
2. **Automorphism**
   - When *g* defines a group, ring, or field, and *h* maps the structure onto itself, FLT acts as an **automorphism generator**.
   - The resulting *f* is an automorphic implementation of the original primitive.

1.4 Cryptographic Implications

- The vast space of possible permutations yields a solution space exceeding: 10^{480}
- This scale defeats long-term adversarial harvesting.
- FLT strengthens resilience against both classical and quantum attacks (Shor's, Grover's).
- Crucially, it does so by enhancing **mature and proven ciphers** like AES-GCM and ChaCha20, not replacing it. It does so, without the cumbersome expansion of keywords or internal states.

2. The FLT in Detail

This section explains the concept of *n*-state inverter, reversible *n*-state inverters and reversing *n*-state inverters. The *n*-state reversible inverters enable the implementation of the Finite Lab-Transform (FLT). The FLT is demonstrated on lookup tables of 8-state ("3-bit") operations.

2.1 The *n*-state inverters

An *n*-state inverter is represented by a sequence of *n* *n*-state symbols. Each symbol in the sequence has a value (an *n*-state value, selected from *n* possible states) and a position in the sequence. For instance: $invn=[a_0\ a_1\ a_2\ \ldots\ a_{n\text{-}1}]$.

For convenience, the set of n-state elements is generally assumed to be {0, 1, 2, … , *n*-1}. For instance 8-state elements are selected from S_8={0, 1, 2, …, 7}.

The position *i* of a symbol a_i is determined starting at a position *i*=0.

One is reminded that the description is for convenience only. In computers, symbols don't exist. Only physical states exist, in datasheets usually provided as LOW (L) and HIGH (H). An *n*-state signal with *n*>2, in current computer technology, is generally a set (or "word") of signals. An 8-state signal may be a representation of a word of 3 bits ($8=2^3$).

An 8-state inverter *inv*8 may be: *inv*8= [7 0 3 0 6 4 5 5]. The meaning is: *inv*8(0)=7; *inv*8(1)=0, … , and *inv*8(7)=5. In this example, the inverter has duplicative states, such as *inv*8(6)=*inv8*(7)=5.

An *n*-state inverter is reversible (or invertible) when there is a rule that undoes the *n*-state inversion. This requires that the *n*-state inverter has exactly *n* different elements in *n* different positions in the representation. While the *n*-state inverter is called *invn*, the reversing inverter is called *rinv*, and *invn*(*rinvn*(*x*))=*x*. The reversible inversion is called a *bijection*.

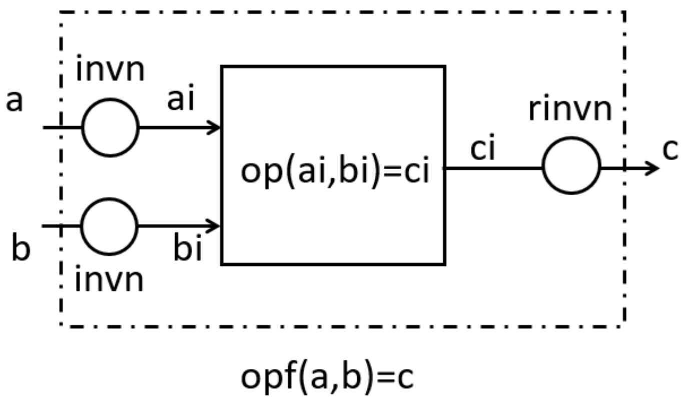
For instance, an 8-state reversible inverter is: *inv*8= [7 0 3 6 2 4 1 5], The corresponding reversing inverter is rinv8=[1 6 4 2 5 7 3 0]. For instance, *inv*8(*rinv*8(0))=0 and *inv*8(*rinv*8(5))=5.

A reversible n-state inverter is a permutation of the identity *invi*=[0 1 2 3 … *n*-1]. There are factorial of *n* (*n*!) different permutations of *invi*, including *invi*.

The growth of number of permutations as a function of *n* is enormous. While starting relatively slow (2!=2; 3!=6, 4!=24, etc), *n*! growth explosively and for *n*=256 one has over 10^{500} different permutations. This effect is generally known as combinatorial or factorial explosion. Factorial growth is faster than exponential growth.

2.2 The Finite Lab-Transform (FLT)

The following diagram illustrates the FLT.



The above diagram illustrates the FLT of a 2-operand *n*-state operation. The core operation is *n*-state operation *ci*=*op*(*ai*,*bi*), wherein *ai*, *bi* and *ci* are *n*-state elements. The operands *ai* and *bi* are created by inverting operands *a* and *b*, respectively, by *n*-state reversible inverter *invn*. The output *ci* of the operation *op* is inverted with the reversing operation *rinvn*.

The FLT transforms the operation *op* to operation *opf*, while preserving all meta-properties of *op*. One may represent the operation op in a look-up table. By running through all *n* input states one may generate also the lookup table for *opf*. While that seems a difficult task, in software it is actually very easy. And surprisingly, for practical applications like for *n*=256 or byte-wise operations, it occupies relatively little storage space. A 256-state 256 by 256 table requires only 64 KB. Which is really small in current computers, where memory is at least in the order of GB and storage often exceeds 1TB.

The FLT applies to any 2-operand *n*-state operation. That means that it applies to operations over a base-field $GF(q)$ with q being prime, as well to operations over extension fields $GF(q^p)$. This by itself is highly unusual, as in general the literature seems to dictate that only one base-field $GF(q)$ exists, usually defined by addition and multiplication modulo-q. This has consequences for cryptographic operations that apply known base-fields such as Elliptic Curve Cryptography (ECC) and PKI methods such as Diffie-Hellman (DH) and RSA.

2.3 Examples for *n*=8 of FLTed Look-up Tables

One application of the FLT is to dramatically enhance security of symmetric encryption like AES-GCM/CTR and ChaCha20. These cryptographic primitives work with bytes (in AES) and words of 32-bits (in ChaCha20). For that reason, the following example uses $n=2^k$. AES-GCM works on *n*=256. But tables of that size are unwieldy as examples, even though easy to manage in software. For that reason, the following examples are provided as 8-state tables. The general properties are fairly easy assessed without being overwhelmed by size of the tables.

The tables are related to the addition and multiplication over $GF(8)$. The addition over $GF(8)$ is formed by bitwise XORing words of 3 bits (2^3=8) and replacing the 3 bit words with equivalent decimal value. The specific multiplication is created by computing all multiplications of polynomials of degree 3 modulo-x^3+x+1.

The addition table is *sc*8 and the multiplication table is *mg*8.

*sc*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

*mg*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

The addition table has as zero element z=0. The zero element means that $sc8(a,z)=a$ for all a. One can see that z=0 in the table of *sc*8, as the column/row in the table for position 0 is identity

[0 1 2 3 4 5 6 7]. A similar effect is shown in table mg8. Because *mg*8(*z*,*a*)=*z* for all *a*, and the column/row in position 0 is [0 0 0 0 0 0 0 0]. Similarly, in *mg*8 *mg*8(*e*,*a*)=*a* for all *e* provides identity for *e*=1.

The tables, on review, identify quickly *z* and *e*.

There are 40,320 different 8-state reversible inverters. Selecting
*inv*8=[5 2 6 7 4 0 1 3] and applying the FLT using this inverter, generates the FLTed tables *sn*8 and *mn*8, as shown below.

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

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

Reviewing identity rows/columns, one finds that *z*=5 in *sn*8 and *e*=6 in *mn*8. And *z*=5 in *mn*8 provides a row/column with all identical elements.

While not directly obvious, one can check that *sn*8 and *mn*8 both are associative, and that *sn*8 and *mn*8 distribute. The functions clearly are commutative and the inversing rules of finite field apply. Accordingly, the functions *sn*8 and *mn*8 establish a finite field *GF*(8).

Also, *sc*8 is a 2-operand self-reversing 8-state function or involution. That is: *c*=*sc*8(*a*,*b*) and *b*=*sc*8(*a*,*c*) and *a*=*sc*8(*b*,*c*). The property of involution is preserved by the FLT and *sn*8 is also a 2-operand involution.

2.4 Application in AES-GCM

The Advanced Encryption Standard is the leading encryption primitive globally. AES is specified in publication NIST FIPS-197. AES comes in different modes. The most widely used mode is AES in Galois Counter Mode (AES-GCM) as defined in TLS 1.3. While standard documents are often notoriously difficult to read, the NIST documents related to AES are exceptionally well written and fairly easy to understand.

AES-GCM and AES-CTR (AES Counter Mode) are different from other AES modes in that the AES part is used to generate a keystream in a one-way approach, The keystream is combined with the plaintext to generate the ciphertext. And the ciphertext is decrypted by combining it again with a locally generated (AES based) keystream. Thus, the keystream generation in AES-GCM should be repeatable, and not necessarily reversible, as in other AES modes.

AES-GCM/256 works with a 256-bit key and an Initial Vector (IV). It applies 5 confusion/diffusion modules or transformations as they are called, to modify an input.
1) the Key Expansion Module, which expands the initial 256-bit (32-byte) key to a key array of 240 bytes. It is computed once at the start of the AES process;
2) the SubBytes() step, applied to the State Array, which is a substitution step using an S-box
3) the ShiftRows() step, which permutes the State Array by rotation;
4) the MixColumns() step, which performs a vector/matrix multiplication; and
5) the AddRoundKey() step, which combines the State Array with a Round Key Array.

Except for the ShiftRows() step, all of the above may be further transformed by the FLT.

A simple modification is in AddRoundKey() in at least one round. The input of AddRoundKey() is the State Array (a 4 by 4 set of bytes or 256-state elements) and a 4 by 4 array of 256-state elements called the Round Key Array. In standard AES the arrays are combined column-wise by bitwise XORing, which may be called standard combining. Using an FLT one may transform the bitwise XORing, which is an addition over *GF*(256), which one may call *sc*256, into a FLTed 256-state function called *sn*256, which is also an addition over *GF*(256).

The effect is illustrated below in the following 256-state arrays as applied to round 9 in AES-GCM.

Round State Array

110	56	156	154
17	110	173	238
148	165	82	191
120	8173	154	92

Round Key Array

50	37	219	129
59	133	161	39
32	28	148	246
180	87	93	25

Combined with *sc*256

92	29	71	27
42	235	12	201
180	185	198	73
204	95	199	69

Combined with *sn*256

131	46	182	49
111	53	81	254
2	20	92	112
109	149	30	147

In an AES-GCM example, the following inputs are applied:
plainText='I like to read! I like to read!'
iv='4392367e62ef9aa706e3e801'
key='44a74c1a57da2bf6d6838956cdca13f1b67cc6ad87d459bff544784083868171'

The effect of this change completely changes the ciphertext that is generated from:
ciphertext_standard =
'6deb6e66165c0f8d85369bb6d2051d4ca7f25733d8432306e112413bff4a2a' to

ciphertext_mod = '4813ca8a7a5e6b78c33ba82bf7e90a4d3ed06b02203ed32f471d32e2644dca'

This illustrates the significant change imposed by the FLT. It is beyond the scope of this introduction to deep-dive in how different FLTs affect the ciphertext. However, it is an indication how the imposed change will ripple through the entire flow of the symmetric encryption and established strong diffusion and confusion of the AES data flow or architecture creates a highly secure modification.

One may apply the FLT to the other modules or transformations in AES-GCM as identified above.

3. Interactive and Dynamic FLT Update

Repetition is the enemy of cryptographic security. An FLT that depends on a pre-stored n-state reversible inverter is less secure than an FLT that is created per cryptographic session, for instance.

3.1 PKI Based Inverter Generation

The US 12,476,789 patent addresses that issue. It does that by the following steps. Keep in mind that a 256-state reversible inverter has 256 different 256-state elements.
1) a shared sequence of n-state elements (like 32 bytes) is received;
2) the shared sequence is expanded to at least 256 bytes. It has with high certainty duplicate bytes and thus also has a number of missing bytes;
3) replace duplicate bytes with missing bytes; and
4) a 256-state reversible inverter has been created and may be applied in an FLT.

A 32-byte sequence is of course a 256-bit sequence. Such a shared sequence may be created as part of a Public Key Infrastructure (PKI) process between 2 computers. A Post Quantum (PQ) secure PKI key generation process is for instance Kyber, defined in NIST FIPS 203.

Thus, there has been created a PKI based method to use a tested key exchange mechanism to establish a common 256-state reversible inverter. This can take place per cryptographic session. One can use the PKI key for both shared key and for inverter generation. One may also apply multiple key generations, one for a symmetric key and one for an n-state reversible inverter. This is already part of for instance TLS 1.3 protocols. One may also easily expand the TLS protocol to generate more than one key.

3.2 Self-propagating Inverters

Repetition is the enemy of security. To prevent repetition, an n-state inverter will generate its own successor. This is done in the following way.

Using the Matlab expression: *invn*=*invn*(*invseed*) for self-propagation. One initializes *invseed* and *invn* with for instance the 256-state reversible inverter generated from a shared key.

Using as example an 8-state inverter *inv*8=[4 2 1 5 6 3 7 0].
Applying *invn*=*invn*(*invseed*), one generates consecutively:
inv81=[6 1 2 3 7 5 0 4];
inv82=[7 2 1 5 0 3 4 6];
inv83=[0 1 2 3 4 5 6 7].

For relatively short *n*-state reversible inverters, the cyclic nature of self-propagation limits the number of different *n*-state reversible inverters that are generated. But, as with other aspects of factorial explosion, these numbers increase dramatically for greater values of *n*, like *n*>200. In that case one routinely can generate 100s of millions of different *n*-state inverters by self-propagation, up to over a billion in case of *n*=256.

As a relatively simple way to further secure extremely large cycles of self-propagation, one may generate for instance a 512-state reversible seed inverter. Self-propagation is achieved by *invn*=*invn*(*inv*512). And the resulting 512-state inverter is simply reduced to a 256-state inverter by dropping all elements greater than 255 (in origin-0). The cycle length of self-propagation for *n*=512 is generally greater than1 billion.

This approach allows real-time change of encryption implementation, per session, per message, per packet and per block, if so desired.

4. Other Transformations/Modifications

The FLT as explained above is actually one of several transformations that expand or enhance the solution space of cryptographic primitives. Following are more, without detailed explanation.

Radix-n transformation

The addition over GF(2^k) is a carry-less operation. One may include a reversible carry-propagating function. In AddRoundKey() in AES, one operates on columns of 4 bytes. One may consider that as a carry-less addition, like [*c*0 *c*1 *c*2 *c*3]=[*a*0 *a*1 *a*2 *a*3]+[*b*0 *b*1 *b*2 *b3*]. Transforming the carry-less addition into a carry-propagating (or radix-n) addition, creates sums that are significantly different from the ones created by carry-less addition.

Radix-n transformation with random carry

A carry propagating addition, like the carry ripple addition, seems to require a fixed carry function. This is not the case in cryptography. In AES-GCM the AES portion is used for key-stream generation and doesn't have to be reversible. However, it must have an output ("sum-space") that is unbiased for any input. It can be shown that as long as the "residue" generating function is reversible, one may select any random carry function. and maintain a sum-space that has a uniform distribution of outcomes.

2-operand *n*-state Involutions Not Being Addition over $GF(n=2^k)$

The XOR based, self-reversing, addition over $GF(2^k)$ is a 2-operand involution. It is ubiquitous in cryptography. It is the final function also in AES-GCM and ChaCha20 that combines plaintext/ciphertext with a keystream. One can construct 2-operand n-state involution operations that are not additions over $GF(2^k)$ and provide different outcomes.

Extremely Large Bit-sequence Reversible Inverters

Some cryptographic primitives use elements that are represented by 100s and sometimes 1000s of bits. Creating n-state inverters seems a daunting task. One relatively simple approach, but requiring some deep preparation work, is the use of *n*-state maximum length feedback shift registers ("FSR").

A content of a k-state shift register is the *n*-state element that is being inverted. So, for instance, a 1024-state inversion can be achieved with a maximum length 256-state FSR with 4 256-state shift register element. An inversion may be running the FSR in p steps forward and capturing the new shift register content, The reversing inverter is running the FSR in reverse direction. And an FLTed FSR may be applied.

One issue is to find primitive polynomials of degree 4 (in the above example) over $GF(256)$. This is quite easily done in MAGMA Calculator. It requires the translation of the specific field notation of Magma into an executing program like in C or Python.

Other Modifications

One possible modification to both AES and ChaCha20 is to expand the size of the state array. In AES, the state array is traditionally defined as 4×4×8=128 bits. This could be generalized to a $k \times k$ array of bytes, where $k>4$, thereby increasing the internal state size and potentially strengthening resistance against exhaustive search and certain structural attacks.

Another modification involves altering the unit size of the processed words in AES. Conventionally, AES operates on 8-bit bytes. By extending this to p-bit words, where $p>8$, one could explore new design spaces with larger substitution-permutation structures.

Yet another modification is by increasing the number of rounds.

All of these modifications—whether expanding the state array, increasing the unit word size, or adding more rounds—ultimately serve as patches rather than true architectural improvements. They do not fundamentally strengthen the underlying structure of the cipher. This situation mirrors the historical struggle with RSA, where continually increasing the modulus size was necessary to stay ahead of successful attacks. Larger parameters may delay brute-force feasibility, but they fail to address deeper structural vulnerabilities.

This distinction is precisely what makes the Finite Lab-Transform (FLT) stand out. It keeps the architecture unchanged, but deeply modifies the functional implementation.

5. Some Limitations

The solution spaces provided by the FLT and others are gigantic. However, they are still finite. The nature of the operations have some inherent repetition. It was already indicated that self-propagation of inverters for relatively small n, may create short or at least shorter self-propagating cycles.

One of the significant limitations is caused by the nature of the addition over GF(2^k). One can see above in the 8-state tables of $sc8$ that $sc8(a,a)=z$ for all a. This inherently limits the number of different functions that can be generated. As with all cases of factorial/combinatorial explosion, the effects appear to be extremely limiting for smaller values of n. For instance, for n=4, 8 and 16. But for n=32 the number of different functions under FLT explode.

A best estimate of a bound of available distinct additions over $GF(n=2^k)$ is:

$$p = (2k)! \,/\, |GL(k,2)|$$

where

$$|\mathrm{GL}(k,2)| = \prod_{i=0}^{k-1}\left(2^k - 2^i\right).$$

This establishes p=240 for n=8; and by “explosion” $p=10^{28}$ for n=32.

For n=256, the number of different additions over GF(256) is greater than 10^{480}, rendering the above limitation utterly irrelevant for practical cryptographic purposes.

The limitations are mentioned to alert potential users to possible duplicates and to carefully watch size of selected parameters. With these conditions sufficiently addressed, one practically will see no effect of any duplicates during the lifetime of our universe.

The solution space of the Finite Lab Transform (FLT) is immense, and can be further expanded by introducing multiple points of transformation within a cryptographic primitive. These enhancements are multiplicative, pushing the effective solution space beyond astronomical magnitudes.

Yet, in cryptography, sheer solution space size is only one—albeit important—dimension of security. Equally critical is theoretical resistance against mathematical analysis (cryptanalysis). Preliminary evaluations using Hamming distance metrics demonstrate excellent diffusion and nonlinearity properties, strengthening resilience against statistical attacks.

Side-channel vulnerabilities, such as timing analysis, can be mitigated through equal-time realization of inverters and operations. This defense is reinforced when all computational steps are implemented with uniform timing, including those that perform no modification (realized as identity inverters). Under this design, an adversary cannot distinguish whether an FLT modification has been applied, thereby neutralizing timing-based leakage.

As cryptographic history consistently illustrates, the disciplined application of secure operational practices is as vital to overall system strength as the choice of secure primitives themselves. The FLT framework, when combined with rigorous operational safeguards, offers a promising foundation for robust cryptographic design.

6. It Works!

Yes, it works — without qualification. By consistently using lookup tables, side-channel attacks do not expose the FLT. One should employ a lookup table together with identity inverters when no change is desired. This ensures that all variations execute at the same speed.

Standard cryptographic primitives have been successfully transformed using the FLT, including both encryption and hashing. Reference implementations in **C**, **Python**, and **Matlab** are available for trial, educational, and research purposes only, and may be downloaded from: https://lcip.in/

7. Info + Contact

One may look up the patents in the USPTO Patent Database or in Google Patents.
For additional information send request to info@labcyfer.com.

8. Intellectual Property (IP) Rights

Peter Lablans
December 7, 2025