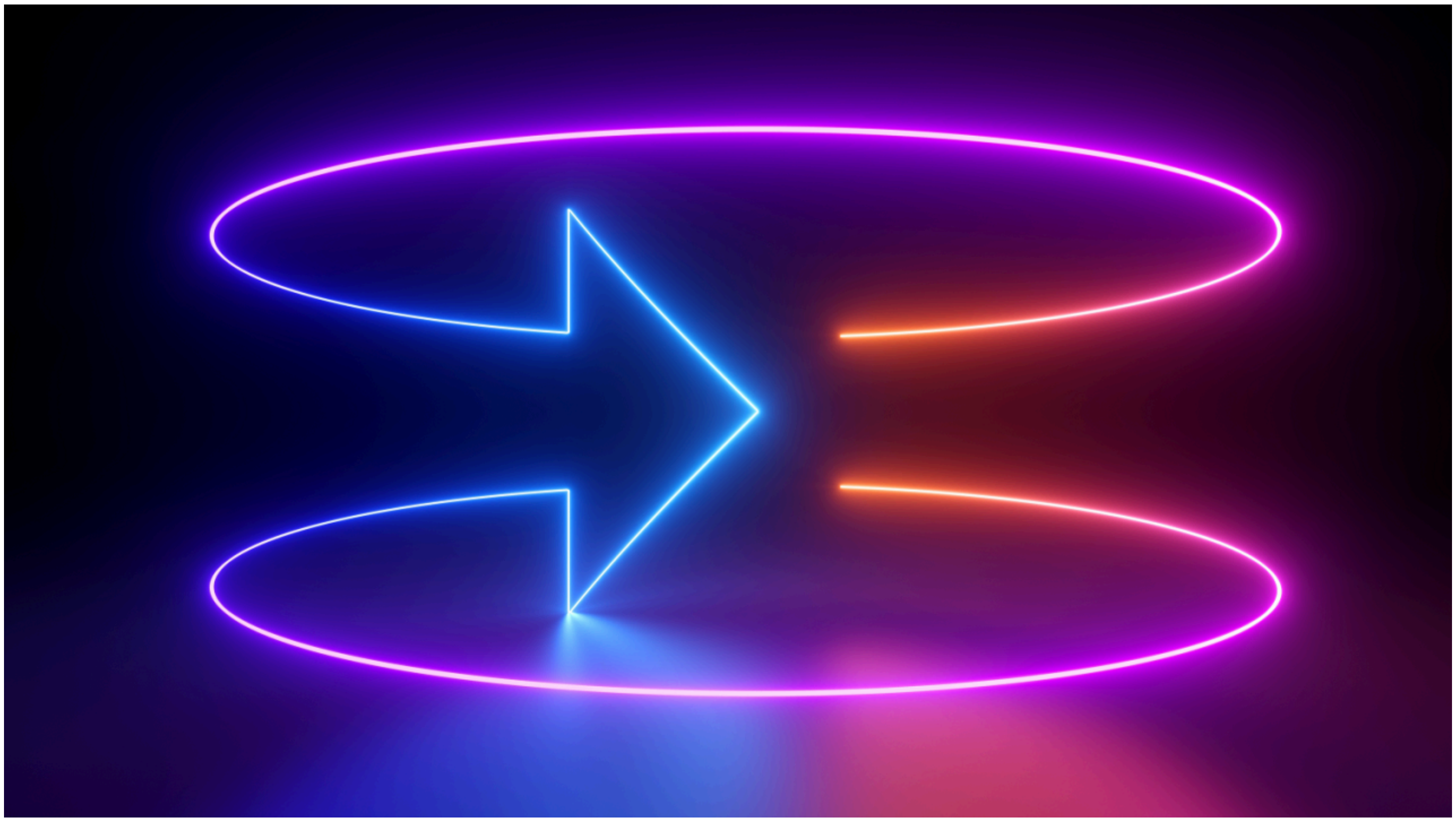


Novel Involution Functions in Cryptography

Peter Lablans | CEO/CTO at LabCyfer | Published December 20, 2024

What Are Involution Functions?

Involution functions are the ubiquitous workhorse function in cryptography such as encryption and hashing. They commonly perform the initial step of ‘entering’ user data into the cryptographic method. One essential property is that no bias towards a particular data value is created. And commonly one applies the XOR function, often mathematically described as the modulo-2 addition, as the involution of choice.

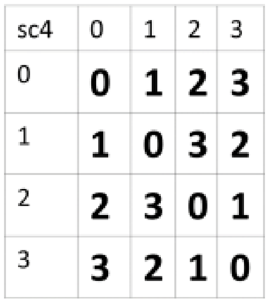
An involution function is its own inverse. One may call them "self-reversing." Assume a 2 operand function ‘*sc*’ so that *c=sc(a,b)* with input operands ‘*a*’ and ‘*b*.’ Then for a commutative involution it applies: *a=sc(c,b)* and also *a=sc(b,c)*. In binary logic there are two functions that perform such a commutative involution: 1) the XOR function and 2) the EQUAL function. Because computers apply computer arithmetic, generally the XOR function is used.

The bitwise XOR is applied in SHA-256 hashing and in AES and ChaCha20 encryption.

N-state Representation of Involutions

One generally uses in cryptography words of bits in bytes in their hexa-decimal representation. One may also use the decimal value of any word of bits. A word of *k*-bits has a 2^*k*-state decimal representation. For instance 8-bits word *num256*=[1 1 1 1 1 1 1 0] in base 2 is 255 decimal and *num*4=[1 1] is decimal 3.

For instance, a table *sc4*, as shown below, representing bitwise XORing of words of 2 bits, may be represented by the following 4-state table:

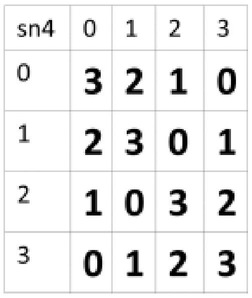


sc4	0	1	2	3
0	**0**	**1**	**2**	**3**
1	**1**	**0**	**3**	**2**
2	**2**	**3**	**0**	**1**
3	**3**	**2**	**1**	**0**

This function or table *sc4* is a commutative involution. This table has certain properties. For instance, it has a zero-element *z=0*. And for all input operands *x* being 0, 1, 2 or 3 *sc(z,x)=x*. And this applies ONLY for zero-element *z=0*.

One question that arises: is or are there other 4-state involution functions, for instance *sn4* for which *z=3*, for instance? This means: *sn4(3,x)=x* for all valid values of *x*. (0, 1, 2, and 3).

Yes, there are. here is one:



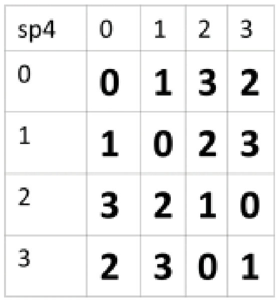
sn4	0	1	2	3
0	**3**	**2**	**1**	**0**
1	**2**	**3**	**0**	**1**
2	**1**	**0**	**3**	**2**
3	**0**	**1**	**2**	**3**

Finding such a function for *n=4* is a bit of a challenge as there are 4^(4^2) (about 4 billion) different 4 by 4 4-state tables. But, by setting condition, it is certainly doable to find them.

Another question is: is there a 4-state involution *sp4* that doesn't have a consistent zero-element, but is still a 4-state commutative involution? Yes, there are several involutions that fit the requirement. The following 2-operand 4-state function *sp4* is an involution.

A Novel 2-operand Involution

There are several involutions that fit the above requirement of a non-consistent zero-element. The following 2-operand 4-state function *sp4* is such an involution.



sp4	0	1	2	3
0	**0**	**1**	**3**	**2**
1	**1**	**0**	**2**	**3**
2	**3**	**2**	**1**	**0**
3	**2**	**3**	**0**	**1**

The above table shows that *sp4(0,z)=0* and *sp4(1,z)=1* for *z=0*; and *sp4(2,z)=2* and s*p4(3,z)=3* for *z=1*. This is a simple illustration for *n*=4. It may appear a bit underwhelming. But the function *sp4* which looks at first blush a bit like *sc4*, will completely alter a combination of 2 streams of binary data, treated as consecutive words of 2 bits.

Looking Behind the Computational Curtain

In general, we accept certain computational operations as foundational. Our natural inclination is to determine patterns from basic building blocks. And we combine the building blocks to see or distinguish the bigger picture, such as the data-flow in encryption for instance. And because the dataflow is so complex, we sometimes forget to take an in-depth look at the building blocks. And we may discover that in a number of cryptographic operations we over-specify requirements of computer functions. And we may actually modify these functions without affecting security. On the contrary, by making modifications we actually increase security.

One reason for this reliance on "standard" or "classical" computer implementations, is a misconception that computers perform mathematics. They don't. We know this since Claude Shannon published his Master Thesis in 1938 entitled "A Symbolic Analysis of Relay and Switching Circuits." What Shannon taught us, was that we may describe switching circuits in terms of mathematics. Unfortunately, this is often interpreted that computer or switching devices perform mathematics, which is not the case.

Powerful Computational Function Transformation

The ability to create novel modified *n*-state involutions is very powerful. It allows processing words of *k* bits, (like bytes of 8 bits) with basically currently unknown involution functions. As the number of bits in a word of bits increases, the number of possible modifications increases exponentially or factorially.

As an example: one may consider In AES the module *AddRoundKey()*. This is a self-reversing operation, which applies a byte-wise XOR (involution) on columns of the state-array and the round-key array. This operation is a known involution. It explains why the same module is applied in both the encryption and decryption operations.

The byte-wise XORing may be replaced by a novel 256-state (=2^8) involution, without changing fundamental and proven properties of AES. But it will dramatically and undetectably to attackers (due to the avalanche effect of AES) modify the output ciphertext. The new involution does not change the statistical properties of the ciphertext. Thus, the modification itself does not leak information into a ciphertext.

In AES-GCM and ChaCha20, the bitwise XOR is also applied in encrypting/decrypting the plaintext respectively the ciphertext with the keystream. A modification of the bitwise XOR with novel involutions offers simple and immediate increase of security.

How to generate these modified involution functions?

Yes, that is the trick of course: how does one find these tremendously useful but incredibly hard to determine modifications of n-state involutions? Especially if one wants to find these modified novel involutions for *n*=256 or greater.

It is incredibly hard when one tries it through brute force. The numbers are just against it. While for *n=4* one has a relatively small set of 4 billion to test, for *n=8* which is slightly more useful, one has 6*10^57 8-state tables to work through. And for *n=256* (using bytes) there are 10^150,000 tables. Brute force doesn't work.

Some modeling may be applied to minimize the process, but not enough to be really helpful, even for *n*=8. The solution that we at LabCyfer found and developed was in Computational Function Transformations. These Transformations are based on modifying known basic computer or machine functions, and creating completely novel and non-obvious computer functionality. They are deterministic and unlock an incredibly large set of computer functions that are largely infeasible to break by brute force and yet deterministic to generate.

And yes, we have developed this for *n*=256 and greater as computer implemented processes.

It is not practical to describe the methods/devices we developed in this introductory description. That is not to keep it secret. It is just that the tables and numbers will be too large. If you want to learn more about how to develop large novel involutions you can find details in the article Novel N-state Commutative (Full) Involutions NOT Being Additions Over GF(2^k).

Contact Us

The purpose of this article is to provide a high-level impression of commutative and non-commutative involutions. Not to provide a way to generate them. You may contact us at info@labcyfer.com for more detailed descriptions.