# Review of the White-Box Encodability of NIST Lightweight Finalists

Alex Charlès[1] and Chloé Gravouil[1,2]

[1] Univ Rennes, CNRS, IRMAR - UMR 6625, F-35000 Rennes, France
[2] EDSI, 8 rue du Bordage, 35510 Cesson-Sévigné, France

**Abstract.** One of the main challenges cryptography needs to deal with is balancing the performances of a cryptographic primitive with its security. That is why in 2015, the National Institute of Standards and Technologies (NIST) has begun a standardization process to solicit the creation of new lightweight cryptographic algorithms. We then wondered which of this standardization finalists would suit the best to a white-box implementation.
To this end, we studied different algorithms structures on their encodability to later develop our white-box encoding solution. Afterwards, we reviewed the standardization finalists on the applicability of our solution to those algorithms, and finally apply it to GIFT, the permutation of GIFT-COFB.
**Keywords :** *White-Box Cryptography, Lightweight Cryptography, Encodings, GIFT-COFB.*

## 1 Introduction

Lightweight cryptography is the field of cryptography that aims to develop cryptographic primitives adapted to devices with limited capacities. To meet with the growing need in this area, the NIST has initiated in 2015 a process to "solicit, evaluate, and standardize" cryptographic algorithms fulfilling this requirement ([12]).

Furthermore, the growing of the number of these active devices goes hand in hand with the need of connecting them with each other or with other devices, for instance in the IoT domain. Hence, the need for lightweight cryptographic primitives eligible for white-box implementations has emerged.

**Definition 1 (from [13]).** ***White-Box Cryptography*** *is an obfuscation technique intended to implement cryptographic primitives in such a way that even an adversary with full access to the implementation and its execution platform is unable to extract key information.*

To protect the key from a white-box attacker, one of the first and most known method proposed by Chow *et al.* [4] in 2002 is to tabularize then encode the implementation. Their original idea was to turn the AES algorithm into a giant look-up table mapping each possible plaintext to its corresponding ciphertext,

hence avoiding any manipulation of the key. Nevertheless, such a table for 128-bit plaintexts and ciphertexts would be too heavy and thus unrealistic ($2^{128} * 128 = 2^{135}$ bits). Hence, the chosen solution was to build a network of encodings look-up tables.

Our approach was to study different structural characteristics of algorithms and their implications on potential encodings, to present a solution that would settle most of the noticed challenges. Then, we reviewed the NIST Lightweight standardization finalists to choose the most suitable to white-boxability by our solution.

**Outline of the paper** In Section 2.1, we describe the principle of encoded networks of look-up tables. Then, in Section 2.2. we make a classification of cryptographic algorithmic structures depending on how their outputs can be encoded, and present how the two different cases can be merged. Thereafter, we present in Sections 2.3. and 2.4. our encoding solution and its design rationale. Then, in Section 3, we review the NIST lightweight finalists to choose GIFT-COFB as the most suitable to the application of our method. Finally, we present this application on GIFT-COFB permutation GIFT.

## 2 A new kind of encoded network of lookup tables

### 2.1 Principle of an encoded network of lookup tables

Implementations with look-up tables aim to gather the operations of the cipher algorithm into small groups of operations to tabularize them. Tabularizing consists in replacing a group of operations $G$ with a lookup-table that links each possible input of $G$ to its corresponding output. These groups of operations need to be small-sized, as an n-bit input to m-bit output group of operations has a corresponding lookup table of $2^n * m$ bits.

Once the whole algorithm has been separated onto small groups of operations, we apply an encoding scheme to it. In order to preserve the original ciphering, we then apply the corresponding decoding to each input of the following group of operations.

We can finally tabularize all the group operations along with their input decoding and output encoding operations. A white-box attacker does not know these encoding and decoding operations, so cannot retrieve the state being ciphered. Ultimately, as this implementation avoids a white-box attacker accessing the state being encoded and protects the operations through encodings, it might succeed in preventing the attacker deducing a key being used inside of some groups of operations.

### 2.2 Choice of the encoding of a group of operations

For each $j^th$ group of operations of round $i$ $G^i_j$, we need to choose an output encoding operation that fits with the usage of the output. Indeed, the encoding

cannot be the same depending if the output bits of the group of operations are used **altogether** or **separately**, which can be represented as two main cases:

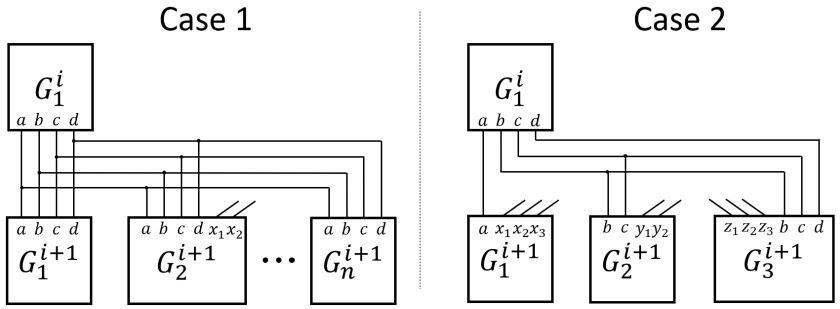


**Fig. 1.** Two cases of groups of operations

**In the first case,** all $n$ output bits of a group of operations $G_j^i$ are used **altogether** in different following groups of operations, which means that they receive all of $G_j^i$ output bits at once. An encoding solution in this case would be to create a revertible $n$-bit lookup table that would be applied to the $n$ output bits. Then, the corresponding $n$-bit decoding lookup table would be applied to each input of the following groups of operations. However, the attacker will know that the same input decoding lookup table is applied in these different following groups of operations, which weakens their encoding by giving him complementary information. To avoid that, a better solution would be to merge the group of operations to each of its following groups of operations.

For instance, in the case 1 of Fig. 1, all $i+1^{th}$ round groups of operations $G_j^{i+1}$ have for input all the output bits $a$, $b$, $c$ and $d$ of the group of operations of round $i$ $G_1^i$. So its output bits are used altogether in the following groups of operations $G_j^{i+1}$. In this case, we can create $G_1'^{i+1} = G_1^{i+1} \circ G_1^i$, that will have the same inputs as $G_1^i$, and the same outputs as $G_1^{i+1}$. Even if a group of operations has some input bits that are not outputs of $G_1^i$ such as $G_2^{i+1}$ with $x_1$ and $x_2$, we can still consider a new group of operations $G_2'^{i+1}$ such that $G_2'^{i+1} = G_2^{i+1}(a, b, c, d, x_1, x_2)$.

However, most cryptographic algorithms tend to have as many input bits as output bits. So, if the output bits of a group of operations are duplicated, the state will have more bits, which implies that its number of bits will be reduced in other groups of operations elsewhere, which is often done with a XOR, which can be problematic (§2.6).

Furthermore, if the output is duplicated then used in $n$ different lookup tables, we need to create $n$ different merged lookup tables. Hence, the number of tables would grow exponentially with the number of rounds.

**In the second case,** the bits of a group of operations $G$ are used **separately** in the following groups of operations, which means that none of its following groups of operations have all the output bits of G as input. In this case, we cannot create a $n$-bit lookup table that will encode the $n$ output bits of a group of operations, as reversing this operation would imply to transmit the $n$ encoded bits, while needing only some of them. Besides increasing the size of the input, an attacker can observe which bits are not used in the following groups of operations, weakening their encoding. The solution is rather to encode the outputs bits per *bundles*.

For instance, in the case 2 of Fig. 1, the output bits $a$, $b$, $c$ and $d$ of the group of operations $G_1^i$ are used separately as each following group of operations $G_j^{i+1}$ do not have all of these four bits for input. Thus, we cannot create a 4-bit lookup table to encode the output of $G_1^i$. In this case, we would rather create three lookup tables to encode $\{a\}$, $\{b, c\}$ and $\{d\}$. Indeed, $a$ is used only in $G_1^{i+1}$, separately from $b$, $c$ and $d$, justifying its solo encoding. We should also create a 2-bit encoding for $b$ and $c$ as $b$ is not used separately from $c$ in any following groups of operations. Lastly, even if $b$, $c$ and $d$ are used in the input of $G_3^{i+1}$, we cannot create a 3-bit encoding for them, as $d$ is not used with $b$ and $c$ for $G_2^{i+1}$ input, so we would rather create another 1-bit encoding for $d$.

The first weakness of encoding separately the output bits is that the output encodings are smaller thus weaker to brute-force attacks, as there exists $(2^n)!$ $n$-bit lookup tables. In particular, a 1-bit encoding correspond to a XOR of a constant bit $\in \{0, 1\}$, which is weak. The second is that an attacker can deduce more information from the output. In fact, in that case we need to encode the 4-bit output of $G_1^i$ as three separate groups $\{a\}$, $\{b, c\}$ and $\{d\}$. So, even if the input of $G_1^i$ is encoded, an attacker can observe which inputs modify only $a$ or $d$ and reciprocally, which gives him information on $G_1^i$.

Finally, the output bits of a group of operations can be used separately in some following groups of operations and altogether in some other groups. In such case, as described in the first case explication, we generate the different merged groups of operations each time the bits are used altogether, to thereafter relate this case to a only separated one, and apply the associated method.

### 2.3 Our new encoding scheme

As seen previously (2.2), the usual encoding methods have intrinsic weaknesses in each case. We developed a new method that can be applied to resolve some of the problems shown above. This method consists of creating *Tboxes*, each protecting a group of operations using pseudo-random bits to encode their outputs. We will show that this method strengthens the brute-force attack resistance, allows small-sized encoding and avoids the attacker getting information on a group of operations using its output bits *separately*, which makes this method well-suited for this case. We present below the steps to apply this method to a general case, and an application on an example.

**Step 1:** For each group of operations, determine how their outputs are used: their different *bundles*. Indeed, we can determine a partition of the output bits of a group of operations, that corresponds to which output bits go to which following group of operations; we call each element of this partition a *bundle*.

In our example of Fig. 2 we want to create a Tbox $T$ for $G_1^i$. The input bits of $G_1^i$ come from two previous groups of operations: three bits from $G_1^{i-1}$ and two from $G_2^{i-1}$, so $G_1^i$ has two input *bundles* of respectively three and two bits. Similarly, $G_1^i$ has three output *bundles*, two 1-bit that go to $G_1^{i+1}$ and $G_3^{i+1}$, and a 2-bit one that goes to $G_2^{i+1}$. In our scheme, we suppose that the previous and following groups of operations of $G_1^i$ use their bits on other groups of operations of the step $i$, otherwise we are in the *altogether* case (see Fig. 1), and we could merge them with $G_1^i$.

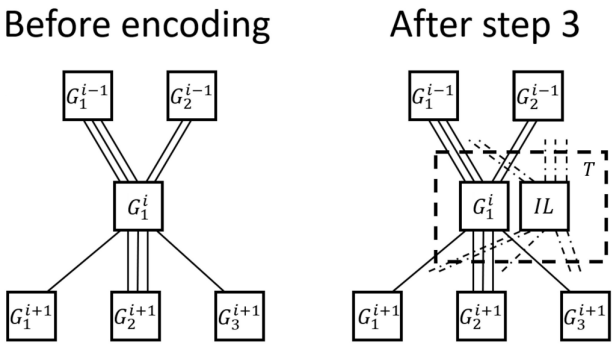


**Fig. 2.** An example of a group of operations being Tboxed with the first three steps

**Step 2:** For each group of operations, we determine the needed size of their *intern lookup table* (noted $IL$ in the scheme), that will be used to modify the input pseudo-random bits. To apply our method, we need to encode each output *bundle* with at least one pseudo-random bit, which determines the minimum of bits that the *intern lookup table* needs to output. Furthermore, the overall Tbox $T$ input needs to have a reasonable size, as a $n$-bit Tbox costs $2^n * n$ bits. For those memory and performance reasons, we considered that the Tbox needs to have a 10-bit input maximum size, which implies that the *intern lookup table* input needs to have a maximum of $10 - |G|$ bits, with $|G|$ the input size of the group of operations being encoded. Lastly, the *intern lookup table* needs to have at least as much input bits as output bits, because if not, the lookup table will not be surjective, which causes statistical bias.

In our example, firstly, $G_1^i$ is a 5-bit to 5-bit group of operations, so the *intern lookup table* can have a maximum of $10 - 5 = 5$ bit of input. Furthermore, $G_1^i$ receives two *bundles* for input, and outputs three *bundles*. So its *intern lookup table* will receive a minimum of two pseudo-random bits, and outputs a minimum

of three. As it needs to have at least as much input bits as output ones, the *intern lookup table* is at minimum a 3-bit to 3-bit one. It implies that either $G_1^{i-1}$ or $G_2^{i-1}$ needs to encode their *bundles* transmitted to $G_1^i$ with at least two pseudo random-bits. For those reasons, here we will choose (arbitrarily) a 5-bit to 5-bit *intern lookup table*.

**Step 3:** Once the *intern lookup table* has been determined, we need to attribute its input and output pseudo-random bits to each input and output *bundle*. Firstly, each *bundle* needs to have at least one pseudo-random bit. Secondly, each *bundle* needs to have the same number of pseudo-random bits attributed by both of its previous and following groups of operations.

As we chose a 5-bit to 5-bit *intern lookup table* for $G_1^i$, we can for instance attribute 2 pseudo-random bits (represented in dashed lines) to the received $G_1^{i-1}$ *bundle*, and hence three to $G_2^{i-1}$. Likewise, we choose to transmit two pseudo random bits for the bundles sent to $G_1^{i+1}$ and $G_3^{i+1}$, and thus one for $G_2^{i+1}$.

**Step 4:** Then, we generate the *bundle* encodings and decodings. For each output *bundle* of $G$ and its associated pseudo-random data, we need to generate a lookup table that has for input size the number of bits of the *bundle* coming from $G$ plus the number of its associated pseudo-random bits. Then, for each output bundle, we need to generate the corresponding decoding lookup table that will be applied accordingly to the input of the following groups of operations.

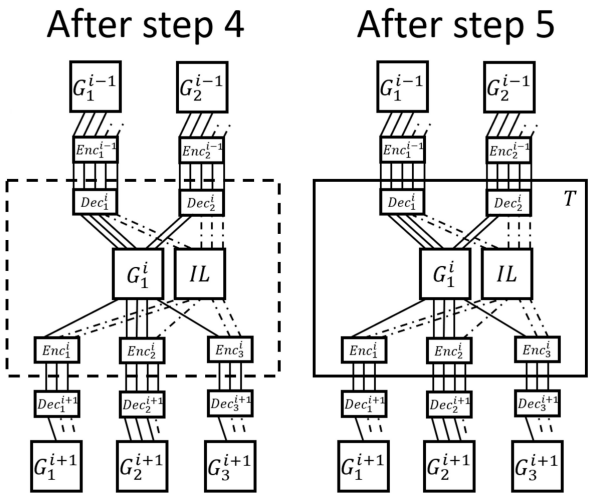


**Fig. 3.** The last two steps of our example being Tboxed

In our case, as shown in Fig. 3, we will generate a 4-bit encoding $Enc_2^i$ for the *bundle* of $G_1^i$ transmitted to $G_2^{i+1}$, as it is a 3-bit *bundle* associated with one

pseudo-random bit. We proceed the same way for the other *bundle* encoding and their corresponding decoding.

**Step 5:** Tabularize each group of operations with their *intern lookup table*, input decodings and output encodings to create its corresponding Tbox.

In our case, we represented in solid line the TBox $T$ of $G_1^i$.

### 2.4 Design rationale

We designed the Tbox to allow an encoding implementation on cases that did not have one. Indeed, to encode the second case described in Fig. 2.2, we cannot simply encode all the output bundles of a group of operations separately, as it is not resistant to brute-force attacks, and allows a white-box attacker to get information on the group of operations.

With a Tbox, the brute-force attack resistance is greatly enhanced, as an attacker needs to enumerate all the possible encoding, decoding and intern lookup tables. This values grows exponentially as there exists $(2^n)!$ possible lookup tables of $n$ bits.

Furthermore, the Tbox has been made to be resistant to differential attacks, as a single bit modification of the input of a Tbox has an impact on the output bits of its bundle decoding. Thus, some bits of the decoded bundle or some pseudo-random bits are modified. If pseudo-random bits or group of operations state bits are modified, then the output of respectively the *intern lookup table* or the group of operations changes, impacting the overall output of the Tbox as each of its output bundles are encoded with at least one pseudo-random bit.

Another possibility for the intern lookup table would be to transmit a copy of the input bits of the group of operations along with its usual pseudo-random bits as input. We avoided this possibility, as the output of the intern lookup table would depend on the input of the group of operations, which will make the overall Tbox output more dependant on the state bits than the pseudo-random ones, which can be observed by a white-box attacker to deduce if he succeeds to modify a state bit rather than a pseudo-random one, judging by its impact on the Tbox output.

As explained in the previous section 2.2, if all groups of operations are using their output bits altogether, the classic encoding scheme will be too heavy when trying to create all the merged groups of operations, so it is a case that we want to avoid. If the groups of operations in the *altogether* case do not generate a too large amount of merged groups of operations, we should relate this case as the *separated* one. Then the Tbox method could be in some cases mandatory, and in others will bring the presented advantages, so must be applied in any cases where it is possible.

### 2.5 The problem of the first and last Tboxes

We suppose that the algorithm will operate with encoded inputs and outputs. As a matter of fact, this is a use case common for instance in the DRM field. Even

with this supposition, our white-box implementation is still exposed to Chosen Plaintext Attacks, if we suppose that the attacker can choose the real plaintext rather than encoded one.

To properly encode the input given to each of the first Tboxes of the implementation, we consider that each of these first Tboxes is receiving a single *bundle* encoded with its needed amount pseudo random bits, and we consider that the last Tboxes are also transmitting a single *bundle*. So, for each of the first Tboxes, we generate a decoding lookup table and tabularize it inside the Tbox to decode its input. We also proceed similarly to the last Tboxes with their output encoding lookup table. However, this obliges the entity which communicates with this implementation to know these encoding and decoding lookup tables, and generate random data. It might not always be feasible, moreover, these random bits could also be seen as an asset to protect.

### 2.6 The weakness of Tboxing the XOR operation

Firstly, the $n$-bit XOR corresponds to a $2n$-bits to $n$-bit group of operations, which is a problem for the pseudo-random data handling. Indeed, as explained in the first case of the section 2.2, reducing the state size with such operation will imply to duplicate some outputs of groups of operations elsewhere as the whole algorithm has often a same amount of input bits than output ones. This duplication is costly to avoid, and if not, implies to decode the duplicated output multiple times with the same decoding lookup table, which can be exploited by an attacker. Furthermore, these following groups of operations that have for input the duplicated output will also have the same pseudo-random bits.

Secondly, encoding a $n$-bit XOR with a Tbox does not prevent an attacker from getting information. Indeed, if the Tbox of the $n$-bit XOR takes for input two encoded $n$-bit vectors $a$ and $b$ and outputs another encoded $n$-bit vector $c$, then for any other input encoded vectors $a'$ and $b'$, if the Tbox still outputs $c$, then we know that the XOR of the decoded values of $a'$ and $b'$ equals the XOR of the decoded values of $a$ and $b$. It can also be deduced that the output pseudo-random bits of the intern lookup table are the same, which gives information on the outputs of the groups of operations of the previous Tboxes.

## 3 Review of the white-boxability of the NIST LWC finalists

We then evaluated which NIST lightweight finalists would be the best to apply to our solution structure-wise.

### 3.1 Overview of the NIST LWC finalists

In April 2019, 56 submissions were admitted to the round 1 of the lightweight standardization process. Then, in August 2019, the NIST announced the 32 most promising candidates qualified for round 2. Finally, in March 2021 the selection of the ten finalists was made public :

- ASCON [7]
- Elephant [10]
- GIFT-COFB [1]
- Grain128-AEAD [9]
- ISAP [6]
- Photon-Beetle [2]
- Romulus [8]
- Sparkle [3]
- TinyJAMBU [11]
- Xoodyak [5]

Amongst these candidates, we want to select the best algorithm for a white-box implementation by using our encoding method. This implementation has for objective to protect the key from a white-box attacker.

### 3.2 Selection of white-boxable NIST LWC finalists

To realize this selection, we studied the finalists given different arguments.

First of all, the dispersion of the key throughout an algorithm forces a white-box attacker to study more parts of it to recover the key([13], §3.2).

Furthermore, following the Kerckhoffs principle we suppose the attacker knows the algorithm except for the key. So a tabularized and encoded operation that depends on the key might be harder to break for a white-box attacker than an operation that does not depend on the key. Furthermore, the disclosure of a state allows an attacker to compute all following operations that are not key-dependent. For these reasons, Isap[6] ascon[7] photon-beetle[2] sparkle[3] xoodyak[5] Grain128-AEAD[9] were eliminated.

We then reviewed in more details the remaining algorithms.

**Why we did not choose Romulus** Romulus uses the block cipher Skinny to perform its own ciphering. The Skinny state can be regarded as a 4x4 table of bytes, with each of its operations being byte-wise and not bit-wise. The MixColumn step uses 8-bit XOR, so this algorithm is in the *altogether* case as described in the section 2.2. The XOR of bytes is too heavy to tabularize, as this 16-bit to 8-bit XOR would create a $2^{16} * 8 = 524$KB lookup table.

To avoid this problem, a solution would be to transform the 8-bit encoding of a byte into two 4-bit encodings to reduce the 16 to 8 bits XOR to two 8 to 4 bits XORs. As we are separating the output bits of the previous operation in two bundles, we are in both *altogether* and *separated* cases. So we have to relate this case as a *separated* case only (2.2). To avoid to reuse the decoding of the duplicated bits used in the MixColumns step, we need to duplicate the previous groups of operations as many times as the number of times their outputs are used in different XOR. Of course, all these duplicated groups operations would have the same input and so the same input decoding, so we have to also duplicate all the previous groups of operations of the algorithm, which is very heavy, considering that Skinny has 40 rounds. Once done, we related this case to an *separated* one, but it still has the problem of the XOR (§2.6). Thus, we decided to not choose Romulus.

**Why we did not choose TinyJAMBU** To perform the permutation step of TinyJAMBU, we consider the 128-bit state as a 128-bit LFSR, which is clocked using 5 bits of state as well as a bit of the key. The state bits cannot be encoded with each other, as the clock operation never uses a same set of state bits. We also cannot encode the state bits without pseudo-random data, as an attacker would try to clock 128 times (i.e. the key size number of times) the LFSR with fixed state value, to deduce at each time whether the encoded result varies or not, deducing the key bit values.

Thus, we would need to apply our method to this algorithm. However, each clock of the LFSR is using four 1-bit XORs (2.6) and one NAND. So, as for Romulus, in order to avoid decoding the bits multiple times we need to duplicate the encoded bit 5 times, as each state is used 5 different times. Because the LFSR can be clocked up to 1024 times, this method would be too heavy. For this reason, we decided to not choose TinyJAMBU.

**Why we did not choose Elephant** Elephant[10] uses a function $mask(K, a, b) = mask_K^{a,b}$ that extends a key $K$ depending on $a, b \in \mathbb{N}$. $a$ and $b$ are dependant on the block indexes of the message and associated data. Thus $mask_K^{a,b}$ depends only on constant inputs, so we want to precompute it to reduce the manipulation of the key. However, the message length and the nonce length are potentially infinite, forcing to restrict their lengths. Furthermore, Elephant proposes two permutations: Keccak and Spongent-$\pi$. We cannot apply our solution to Keccak as it uses many XOR operations.

However, our solution can be applied in the same fashion to Spongent-$\pi$ as GIFT, the permutation of GIFT-COFB. Indeed, if we forgot about GIFT round key, we can observe that both algorithms are using 4-bit Sboxes, followed by a bitwise permutation, that uses the Sbox output as four 1-bit *bundles* (§2.3). Nonetheless, if we achieve a white-box implementation of this permutation, as $mask_K^{a,b}$ varies, each of its calls and their white-box implementations will be different which is very heavy, as we are realising a constant-key implementation.

For those reasons, we choose to focus in the application of our solution on GIFT.

### 3.3 Overview of GIFT

In each of its forty rounds, GIFT uses three operations: *SubCells*, *PermBits* and *AddRoundKey*:

- *SubCells* applies the 4-bit GIFT SBox GS to all 4-bit nibbles of the 128 bit state.
- *PermBits* applies a bitwise permutation to the state.
- *AddRoundKey* XOR two bits of the current round key and one bit of the round constant to each 4-bit nibble of the state.

**Algorithm 1:** GIFT($S$,$RK$)

**Input :** S, the 128-bit state and RK the 64-bit round keys
**Output:** S

**for** $i = 0$ **to** 39 **do**
  $S \leftarrow SubCells(S)$
  $S \leftarrow PermBits(S)$
  $S \leftarrow AddRoundKey(S, RK[i])$
**end for**
**return** $S$

### 3.4 Presentation of our solution on GIFT

Firstly, any operation of GIFT duplicates its output. So GIFT figures among the *separated* case 2.2. The best solution to encode GIFT would be to choose to regroup the 4-bit Sbox application and its 2-bit key bit XOR and 1-bit round constant XOR as a group of operations. That way, each of GIFT forty rounds will contain 32 groups of operations that we want to encode.

Choosing another way to regroup the operations seems impossible at low cost, as regrouping many Sboxes as a groups of operations would be too heavy. Even if separating a Sbox application would allow an heavier encoding, creating this network of groups of operations to replace the Sbox would take the same 4-bit input and return the same 4-bit output, while giving the attacker more possibilities of modifications during the Sbox application.

In order to use the key at any round, PermBits and AddRoundKey need to be swapped. Indeed, $PermBits(S) \oplus E = PermBits(S \oplus PermBits^{-1}(E))$, for S the state and E the key and round constants array. (see Algorithm 2)

**Algorithm 2:** $\widetilde{GIFT}(S,\widetilde{RK})$

**Input :** S, the 128-bit state and $\widetilde{RK} = PermBits^{-1}(RK)$
**Output:** S

**for** $i = 0$ **to** 39 **do**
  $S \leftarrow SubCells(S)$
  $S \leftarrow AddRoundKey(S, \widetilde{RK}[i])$
  $S \leftarrow PermBits(S)$
**end for**
**return** $S$

The SubCells and AddRoundKey operations can now be merged together in a single group of operations. To Tbox it, we can observe that this group of operations has four 1-bit input and output *bundles*, which obliges us to select an *intern lookup table* with four to six input and output bits. The GIFT bit-wise permutation has the particularity to link each $i^{th}$ output bit of an Sbox to a $i^{th}$

input bit of a following one, with $i \in \{0, ...3\}$. Thus, we can encode each *bundle* with a different amount of bits as long as we decode its corresponding *bundle* with the same sized decoding. We decided to encode the first and last *bundle* with a 2-bit encoding, and the two left middle ones with 2-bit or 3-bit one, as shown in the scheme 4.

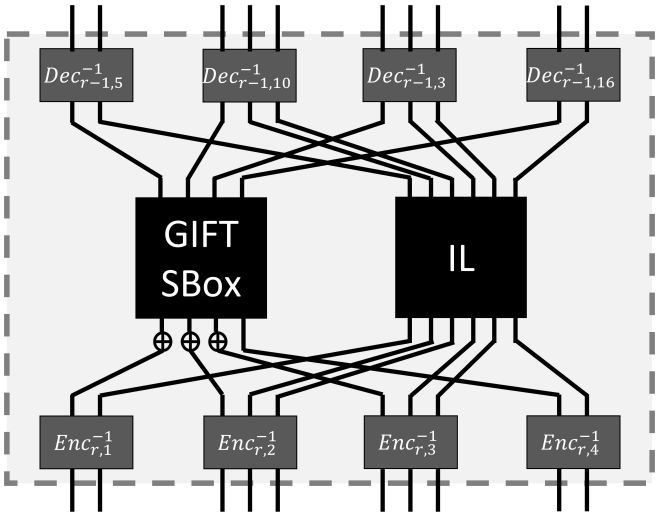


**Fig. 4.** $T_0$, the right-most Tbox of the round r

As GIFT has 1-bit bundles, the Tbox implementation was mandatory and brings all its advantages presented in 2.4. In particular, GIFT is in the *separated* case, ensuring that each *bundle* is encoded and decoded only once. Furthermore, GIFT allows a Tbox implementation with less than 10-bit input. Lastly, if we choose to encode each bundle with only 1 pseudo-random bit, the *intern lookup table* would be 4-bit to 4-bit, leading the brute-force attack to $(2^2!)^4 * (2^2!)^4 * (2^4)! \approx 2^{80}$ possibilities. If we encode the two middle bundles with 3-bit encodings, there are four 2-bit lookup tables, four 3-bit lookup tables and one 6-bit lookup table. Then, a brute force attack would require to go through $(2^2!)^4 * (2^3!)^4 * (2^6)! \approx 2^{375}$ possibilities.

### 3.5 Performances

We propose below a comparison of two versions of the implementation of GIFT, mainly to discuss the impact of our encoding scheme on both the execution time and the size of the implementation considered. In the following, we denote by :

- *GIFTEmbedded* : a bitsliced tabularized implementation of GIFT with constant key ;

– *GIFTEncoded* : a bitsliced encoded version of *GIFTEmbedded* using our encoding scheme previously exposed, with each bundle encoded with only 1 pseudo-random bit.

The processor used for testing is an *11th Gen Intel Core i7-1185G7 @ 3.00GHz 1.80 GHz*, and we used the GCC compiler with optimization options (-Os for binary size measurements and -O2 for execution time measurements).

For the execution time procedure, we generated 4000 random plaintexts and measured the elapsed time of the execution of GIFT on these inputs. These plaintexts are used both for *GIFTEmbedded* and *GIFTEncoded* in order to have the same test context for both implementations. Here are the results :

| Nature of the tests | *GIFTEmbedded* | *GIFTEncoded* |
|---|---|---|
| Execution time (4000 runs) | 15.34 ms | 94.68 ms |
| Size of binary | 132.2 kB | 1.2 MB |

## 4 Conclusion

We classified the structures of cryptographic algorithms into two different cases, whether the output bits of groups of operations are used *separately* or *altogether* in the following groups of operations. Then, we showed that all cases can be related to the *separated* one, and presented our encoding solution to this case. Afterwards, we reviewed the NIST lightweight standardization finalists on the white-boxability of their structure to select the one the most suitable to the application of our solution : GIFT-COFB. Finally, we presented our solution applied to GIFT-COFB base permutation GIFT.

## References


1. Banik, S., Chakraborti, A., Iwata, T., Minematsu, K., Nandi, M., Peyrin, T., Sasaki, Y., Sim, S.M., Todo, Y.: GIFT-COFB. IACR Cryptol. ePrint Arch. p. 738 (2020), https://eprint.iacr.org/2020/738
2. Bao, Z., Chakraborti, A., Datta, N., Guo, J., Nandi, M., Peyrin, T., Yasuda, K.: Photon-beetle authenticated encryption and hash family (2020), https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/round-2/spec-doc-rnd2/photon-beetle-spec-round2.pdf
3. Beierle, C., Biryukov, A., dos Santos, L.C., Großschädl, J., Perrin, L., Udovenko, A., Velichkov, V., Wang, Q.: Lightweight AEAD and hashing using the sparkle permutation family. IACR Trans. Symmetric Cryptol. **2020**(S1), 208–261 (2020). https://doi.org/10.13154/tosc.v2020.iS1.208-261, https://doi.org/10.13154/tosc.v2020.iS1.208-261
4. Chow, S., Eisen, P.A., Johnson, H., van Oorschot, P.C.: White-box cryptography and an AES implementation. In: Nyberg, K., Heys, H.M. (eds.) Selected Areas in Cryptography, 9th Annual International Workshop, SAC 2002, St. John's, Newfoundland, Canada, August 15-16, 2002. Revised Papers. Lecture Notes in Computer Science, vol. 2595, pp. 250–270. Springer (2002). https://doi.org/10.1007/3-540-36492-7_17
5. Daemen, J., Hoffert, S., Peeters, M., Assche, G.V., Keer, R.V.: Xoodyak, a lightweight cryptographic scheme. IACR Trans. Symmetric Cryptol. **2020**(S1), 60–87 (2020). https://doi.org/10.13154/tosc.v2020.iS1.60-87, https://doi.org/10.13154/tosc.v2020.iS1.60-87
6. Dobraunig, C., Eichlseder, M., Mangard, S., Mendel, F., Mennink, B., Primas, R., Unterluggauer, T.: Isap v2.0. IACR Trans. Symmetric Cryptol. **2020**(S1), 390–416 (2020). https://doi.org/10.13154/tosc.v2020.iS1.390-416, https://doi.org/10.13154/tosc.v2020.iS1.390-416
7. Dobraunig, C., Eichlseder, M., Mendel, F., Schläffer, M.: Ascon v1.2: Lightweight authenticated encryption and hashing. J. Cryptol. **34**(3), 33 (2021). https://doi.org/10.1007/s00145-021-09398-9, https://doi.org/10.1007/s00145-021-09398-9
8. Guo, C., Iwata, T., Khairallah, M., Minematsu, K., Peyrin, T.: Romulus v1.3 (2020), https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/finalist-round/updated-spec-doc/romulus-spec-final.pdf
9. Hell, M., Johansson, T., Meier, W., S¨onnerup, J., Yoshida, H.: Grain-128aead, a lightweight aead stream cipher (2020), https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/round-2/spec-doc-rnd2/grain-128aead-spec-round2.pdf
10. KU, T.B., Chen, Y.L., Mennink, C.D.B.: Elephant v2 (2020), https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/finalist-round/updated-spec-doc/elephant-spec-final.pdf
11. Saha, D., Sasaki, Y., Shi, D., Sibleyras, F., Sun, S., Zhang, Y.: On the security margin of tinyjambu with refined differential and linear cryptanalysis. IACR Cryptol. ePrint Arch. p. 1045 (2020), https://eprint.iacr.org/2020/1045
12. National Institute of Standard and Technology: NIST lightweight standardization process, https://csrc.nist.gov/Projects/Lightweight-Cryptography
13. Wyseur, B.: White-Box Cryptography. Ph.D. thesis, Katholieke Universiteit Leuven (2009), https://www.esat.kuleuven.be/cosic/publications/thesis-152.pdf