GIFT: A Small Present

Towards Reaching the Limit of Lightweight Encryption (Full version)

Subhadeep Banik[1,5], Sumit Kumar Pandey[2], Thomas Peyrin[1,2,3]
Yu Sasaki[4], Siang Meng Sim[2], and Yosuke Todo[4]

[1] Temasek Laboratories, Nanyang Technological University, Singapore
bsubhadeep@ntu.edu.sg

[2] School of Physical and Mathematical Sciences
Nanyang Technological University, Singapore
emailpandey@gmail.com,thomas.peyrin@ntu.edu.sg,SSIM011@e.ntu.edu.sg

[3] School of Computer Science and Engineering
Nanyang Technological University, Singapore
[4] NTT Secure Platform Laboratories, Japan
Todo.Yosuke@lab.ntt.co.jp,Sasaki.Yu@lab.ntt.co.jp

[5] LASEC, École Polytechnique Fédérale de Lausanne, Switzerland.

Abstract. In this article, we revisit the design strategy of PRESENT, leveraging all the advances provided by the research community in construction and cryptanalysis since its publication, to push the design up to its limits. We obtain an improved version, named GIFT, that provides a much increased efficiency in all domains (smaller and faster), while correcting the well-known weakness of PRESENT with regards to linear hulls.
GIFT is a very simple and clean design that outperforms even SIMON or SKINNY for round-based implementations, making it one of the most energy efficient ciphers as of today. It reaches a point where almost the entire implementation area is taken by the storage and the Sboxes, where any cheaper choice of Sbox would lead to a very weak proposal. In essence, GIFT is composed of only Sbox and bit-wiring, but its natural bitslice data flow ensures excellent performances in all scenarios, from area-optimised hardware implementations to very fast software implementation on high-end platforms.
We conducted a thorough analysis of our design with regards to state-of-the-art cryptanalysis, and we provide strong bounds with regards to differential/linear attacks.

Key words: lightweight cryptography, block cipher, PRESENT, GIFT

Table of Contents

1 Introduction

In the past decade, the development of ubiquitous computing applications triggered the rapid expansion of the lightweight cryptography research field. All these applications operating in very constrained devices may require certain symmetric-key cryptography components to guarantee privacy and/or authentication for the users, such as block or stream ciphers, hash functions or MACs. Existing cryptography standards such as `AES` [20] or `SHA-2` [35] are not always suitable for these strong constraints. There have been extensive research conducted in this direction, with countless new primitives being introduced [2, 4, 5, 12, 15, 24, 41], many of them getting broken rather rapidly (designing a cipher with strong constraints is not an easy task). Conforming to general trend, the American National Institute for Science and Technology (NIST) recently announced that it will consider standardizing some lightweight functions in a few years [36]. Some lightweight algorithms such as `PRESENT` [12], `PHOTON` [23] and `SPONGENT` [11] have already been included into ISO standards (ISO/IEC 29192-2:2012 and ISO/IEC 29192-5:2016).

Comparing different lightweight primitives is a very complex task. First, lightweight encryption encompasses a broad range of use cases, from passive RFID tags (that require a very low power consumption to operate) to battery powered devices (that require a very low energy consumption to maximise its life span) or low-latency applications (for disk encryption). While it is generally admitted that a major criterion for lightweight encryption is area minimisation, the throughput/area ratio is also very important because it shows the ability of the algorithm to provide good implementation trade-offs (this ratio is also correlated to the power or energy consumption of the algorithm). Moreover, the range of the various platforms to consider is very broad, starting from tiny RFID tags to rather powerful ARM processors. Even high-end servers have to be taken into account as it is likely that these very small and constrained devices will be communicating with back-end servers [6].

While most ciphers take lightweight hardware implementations into account to some extend, `PRESENT` [12] is probably one of the first candidates that was exclusively designed for that purpose. Its design is inspired by `SERPENT` [7] and is very simple: the round function is simply composed of a layer of small 4-bit Sboxes, followed by a bit permutation layer (essentially free in hardware) and a subkey addition. `PRESENT` has been extensively analysed in the past decade, and while its security margin has eroded, it remains a secure cipher. One can note that the weak point of `PRESENT` is the tendency of linear trails to cluster and to create powerful linear hulls [10, 17].

Since the publication of `PRESENT`, many advances have been obtained, both in terms of security analysis and primitive design. The NSA proposed in 2013 two ciphers [4], `SIMON` and `SPECK`, that can reach much better efficiency in both hardware and software when compared to all other ciphers. However, this comes at the cost that proving simple linear/differential bounds for `SIMON` is much more complicated than for Substitution-Permutation-Network (SPN) ciphers like `PRESENT` (`SIMON` is based on a Feistel construction, with an internal function

that uses only a AND, some XORs and some rotations). Besides, no preliminary analysis or rationale was provided by the SIMON authors. Last year, the tweakable block cipher SKINNY [5] was published to compete with SIMON's efficiency for round-based implementations, while providing strong linear/differential bounds.

As of today, SIMON and SKINNY seem to have a clear advantage in terms of efficiency when compared to other designs. Yet, PRESENT remains an elegant design, that suffers from being one of the first lightweight encryption algorithm to have been published, and thus not benefiting from the many advances obtained by the research community in the recent years.

Our contributions. In this article, we revisit the PRESENT construction, 10 years after the original publication of PRESENT. This led to the creation of GIFT, a new lightweight block cipher, improving over PRESENT in both security and efficiency. Interestingly, our cipher GIFT offers extremely good performances and even surpasses both SKINNY and SIMON for round-based implementations (see Table 1). This indicates that GIFT is probably the cipher the most suited for the very important low-energy consumption use cases. Due to its simplicity and natural bitslice organisation of the inner data flow, our cipher is very versatile and performs also very well on software, reaching similar performances as SIMON, the current fastest lightweight candidate on software.

Table 1. Hardware performances of round-based implementations of PRESENT, SKINNY, SIMON and our new cipher GIFT, synthesized with STM 90nm Standard cell library.

	Area (GE)	Delay (ns)	Cycles	TP_{MAX} (MBit/s)	Power (μW) (@10MHz)	Energy (pJ)
GIFT-64-128	1345	1.83	29	1249.0	74.8	216.9
SKINNY-64-128	1477	1.84	37	966.2	80.3	297.0
PRESENT 64/128	1560	1.63	33	1227.0	71.1	234.6
SIMON 64/128	1458	1.83	45	794.8	72.7	327.3
GIFT-128-128	1997	1.85	41	1729.7	116.6	478.1
SKINNY-128-128	2104	1.85	41	1729.7	132.5	543.3
SIMON 128/128	2064	1.87	69	1006.6	105.6	728.6

In more details, we have revisited the PRESENT design strategy and pushed it to its limits, while providing special care to the known weak point of PRESENT: the linear hulls. The diffusion layer of PRESENT being composed of only a bit permutation, most of the security of PRESENT relies on its Sbox. This Sbox presents excellent cryptographic properties, but is quite costly. Indeed, it is trivial to see that the PRESENT Sbox needs to have a branching number of 3, or very good differential paths would exist otherwise (with only a single active Sbox per round). We managed to remove this constraint by carefully crafting the bit permutation in conjunction with the Difference Distribution Table (DDT)/Linear Approximation Table (LAT) of the Sbox. We remark that, to the best of the

authors knowledge, this is the first time that the linear layer and the Sbox are fully intricate in a SPN cipher.

In terms of performances, removing this Sbox constraint allowed us to choose a much cheaper Sbox, which is actually what composes most of the overall area cost in PRESENT. GIFT is not only much smaller, but also much faster than PRESENT. As can be seen in Table 2, GIFT is by far the cipher that uses the least total number of operation per bit up to now. In terms of security, we are able to provide strong security bounds for simple differential and linear attacks. We can even show that GIFT is very resistant against linear hulls, and the clustering effect is greatly reduced when compared to PRESENT, thus correcting its main weak point. We have conducted a thorough security analysis of our candidate with state-of-the-art cryptanalysis techniques.

Table 2. Total number of operations and theoretical performance of GIFT and various lightweight block ciphers. N denotes a NOR gate, A denotes a AND gate, X denotes a XOR gate.

Cipher	nb. of rds	gate cost (per bit per round)			nb. of op.	nb. of op.	round-based
		int. cipher	key sch.	total	w/o key sch.	w/ key sch.	impl. area
GIFT -64-128	28	1 N 2 X		1 N 2 X	3×28 $= 84$	3×28 $=$ **84**	$1 + 2.67 \times 2$ $=$ **6.34**
SKINNY -64-128	36	1 N 2.25 X	0.625 X	1 N 2.875 X	3.25×36 $= 117$	3.875×36 $=$ **139.5**	$1 + 2.67 \times 2.875$ $=$ **8.68**
SIMON -64/128	44	0.5 A 1.5 X	1.5 X	0.5 A 3.0 X	2×44 $= 88$	3.5×44 $=$ **154**	$0.67 + 2.67 \times 3$ $=$ **8.68**
PRESENT -128	31	1 A 3.75 X	0.125 A 0.344 X	1.125 A 4.094 X	4.75×31 $= 147.2$	5.22×31 $=$ **161.8**	$1.5 + 2.67 \times 4.094$ $=$ **12.43**
GIFT -128-128	40	1 N 2 X		1 N 2 X	3.0×40 $= 120$	3.0×40 $=$ **120**	$1 + 2.67 \times 2$ $=$ **6.34**
SKINNY -128-128	40	1 N 2.25 X		1 N 2.25 X	3.25×40 $= 130$	3.25×40 $=$ **130**	$1 + 2.67 \times 2.25$ $=$ **7.01**
SIMON -128/128	68	0.5 A 1.5 X	1 X	0.5 A 2.5 X	2×68 $= 136$	3×68 $=$ **204**	$0.67 + 2.67 \times 2.5$ $=$ **7.34**
AES -128	10	4.25 A 16 X	1.06 A 3.5 X	5.31 A 19.5 X	20.25×10 $= 202.5$	24.81×10 $=$ **248.1**	$7.06 + 2.67 \times 19.5$ $=$ **59.12**

We end up with a very natural and clean cipher, with a simple round function and key schedule (composed of only a bit permutation, thus essentially free in hardware). The cipher can be seen in three different representations (classical 1D, bitslice 2D, and 3D), each offering simple yet different perspective on the cipher's security and opportunities for implementation improvements. GIFT comes in two versions, both with a 128-bit key: one 64-bit block version GIFT-64 and one 128-bit block version GIFT-128. The only difference between these two versions is the bit permutation to accommodate twice more state bits for GIFT-128.

In our hardware implementations of GIFT the storage composes about 75% of the total area, and the (very cheap) Sbox about 20%. Since any weaker choice of

the Sbox would lead to a very insecure design, we argue that GIFT is probably very close to reaching the area limit of lightweight encryption.

Outline. We first specify GIFT in Section 2, and we provide the design rationale in Section 3. A thorough security analysis is performed in Section 4, while performances and implementation strategies are given in Section 5 and Section 6 for hardware and software respectively.

2 Specifications

In this work, we propose two versions of GIFT, GIFT-64-128 is a 28-round SPN cipher and GIFT-128-128 is a 40-round SPN cipher, both versions have a key length of 128-bit. For short, we call them GIFT-64 and GIFT-128 respectively.

GIFT can be perceived in three different representations. In this paper, we adopt the classical 1D representation, describing the bits in a row like PRESENT. It can also be described in bitslice 2D, a rectangular array like RECTANGLE [48] (see Appendix A), and even in 3D cuboid like 3D [34] (see Appendix B).

Round function. Each round of GIFT consists of 3 steps: SubCells, PermBits, and AddRoundKey, which is conceptually similar to wrapping a gift:

1. Put the content into a box (SubCells);
2. Wrap the ribbon around the box (PermBits);
3. Tie a knot to secure the content (AddRoundKey).

Figure 1 illustrates 2 rounds of GIFT-64.

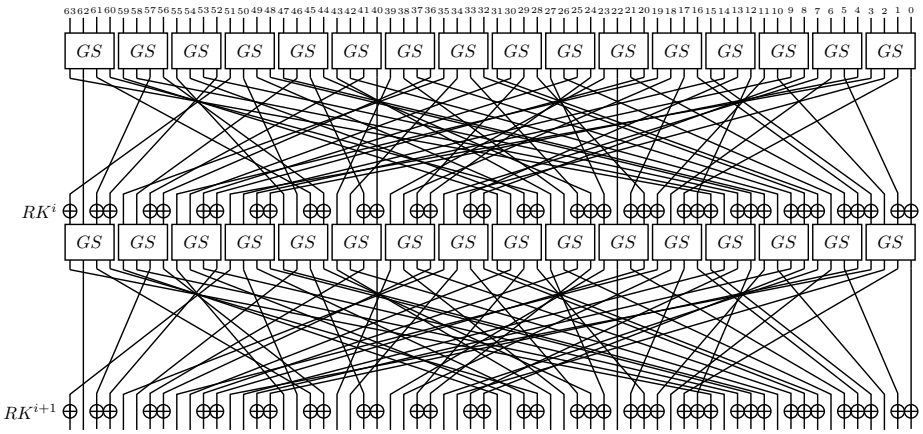

Fig. 1. 2 Rounds of GIFT-64.

Initialization. The cipher receives an n-bit plaintext $b_{n-1}b_{n-2}...b_0$ as the cipher state S, where $n = 64, 128$ and b_0 being the least significant bit. The cipher state can also be expressed as s many 4-bit nibbles $S = w_{s-1}||w_{s-2}||...||w_0$, where $s = 16, 32$. The cipher also receives a 128-bit key $K = k_7||k_6||...||k_0$ as the key state, where k_i is a 16-bit word.

SubCells. Both versions of GIFT use the same invertible 4-bit Sbox, GS. The Sbox is applied to every nibble of the cipher state.

$$w_i \leftarrow GS(w_i), \ \forall i \in \{0, ..., s-1\}.$$

The action of this Sbox in hexadecimal notation is given in Table 3.

Table 3. Specifications of GIFT Sbox GS.

x	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
$GS(x)$	1	a	4	c	6	f	3	9	2	d	b	7	5	0	8	e

PermBits. The bit permutation used in GIFT-64 and GIFT-128 are given in Table 4 and 5 respectively. It maps bits from bit position i of the cipher state to bit position $P(i)$.

$$b_{P(i)} \leftarrow b_i, \ \forall i \in \{0, ..., n-1\}.$$

The permutations can also be expressed as:

$$P_{64}(i) = 4\left\lfloor \frac{i}{16} \right\rfloor + 16\left(\left(3\left\lfloor \frac{i \bmod 16}{4} \right\rfloor + (i \bmod 4)\right) \bmod 4\right) + (i \bmod 4),$$
$$P_{128}(i) = 4\left\lfloor \frac{i}{16} \right\rfloor + 32\left(\left(3\left\lfloor \frac{i \bmod 16}{4} \right\rfloor + (i \bmod 4)\right) \bmod 4\right) + (i \bmod 4).$$

Table 4. Specifications of GIFT-64 Bit Permutation.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$P_{64}(i)$	0	17	34	51	48	1	18	35	32	49	2	19	16	33	50	3
i	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
$P_{64}(i)$	4	21	38	55	52	5	22	39	36	53	6	23	20	37	54	7
i	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
$P_{64}(i)$	8	25	42	59	56	9	26	43	40	57	10	27	24	41	58	11
i	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
$P_{64}(i)$	12	29	46	63	60	13	30	47	44	61	14	31	28	45	62	15

AddRoundKey. This step consists of adding the round key and round constants. An $n/2$-bit round key RK is extracted from the key state, it is further partitioned into 2 s-bit words $RK = U||V = u_{s-1}...u_0||v_{s-1}...v_0$, where $s = 16, 32$ for GIFT-64 and GIFT-128 respectively.
For GIFT-64, U and V are XORed to $\{b_{4i+1}\}$ and $\{b_{4i}\}$ of the cipher state respectively.

$$b_{4i+1} \leftarrow b_{4i+1} \oplus u_i, \ b_{4i} \leftarrow b_{4i} \oplus v_i, \ \forall i \in \{0, ..., 15\}.$$

For GIFT-128, U and V are XORed to $\{b_{4i+2}\}$ and $\{b_{4i+1}\}$ of the cipher state respectively.

$$b_{4i+2} \leftarrow b_{4i+2} \oplus u_i, \ b_{4i+1} \leftarrow b_{4i+1} \oplus v_i, \ \forall i \in \{0, ..., 31\}.$$

Table 5. Specifications of `GIFT-128` Bit Permutation.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$P_{128}(i)$	0	33	66	99	96	1	34	67	64	97	2	35	32	65	98	3
i	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
$P_{128}(i)$	4	37	70	103	100	5	38	71	68	101	6	39	36	69	102	7
i	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
$P_{128}(i)$	8	41	74	107	104	9	42	75	72	105	10	43	40	73	106	11
i	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
$P_{128}(i)$	12	45	78	111	108	13	46	79	76	109	14	47	44	77	110	15
i	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
$P_{128}(i)$	16	49	82	115	112	17	50	83	80	113	18	51	48	81	114	19
i	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
$P_{128}(i)$	20	53	86	119	116	21	54	87	84	117	22	55	52	85	118	23
i	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
$P_{128}(i)$	24	57	90	123	120	25	58	91	88	121	26	59	56	89	122	27
i	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127
$P_{128}(i)$	28	61	94	127	124	29	62	95	92	125	30	63	60	93	126	31

For both versions of `GIFT`, a single bit "1" and a 6-bit round constant $C = c_5c_4c_3c_2c_1c_0$ are XORed into the cipher state at bit position $n-1$, 23, 19, 15, 11, 7 and 3 respectively.

$$b_{n-1} \leftarrow b_{n-1} \oplus 1,$$
$$b_{23} \leftarrow b_{23} \oplus c_5,\ b_{19} \leftarrow b_{19} \oplus c_4,\ b_{15} \leftarrow b_{15} \oplus c_3,$$
$$b_{11} \leftarrow b_{11} \oplus c_2,\ b_7 \leftarrow b_7 \oplus c_1,\ b_3 \leftarrow b_3 \oplus c_0.$$

Key schedule and round constants. The key schedule and round constants are the same for both versions of `GIFT`, the only difference is the round key extraction. A round key is *first* extracted from the key state before the key state update.
For `GIFT-64`, two 16-bit words of the key state are extracted as the round key $RK = U||V$.

$$U \leftarrow k_1,\ V \leftarrow k_0.$$

For `GIFT-128`, four 16-bit words of the key state are extracted as the round key $RK = U||V$.

$$U \leftarrow k_5||k_4,\ V \leftarrow k_1||k_0.$$

The key state is then updated as follows,

$$k_7||k_6||...||k_1||k_0 \leftarrow k_1 \ggg 2||k_0 \ggg 12||...||k_3||k_2,$$

where $\ggg i$ is an i bits right rotation within a 16-bit word.

The round constants are generated using the same 6-bit affine LFSR as SKINNY, whose state is denoted as $(c_5, c_4, c_3, c_2, c_1, c_0)$. Its update function is defined as:

$$(c_5, c_4, c_3, c_2, c_1, c_0) \leftarrow (c_4, c_3, c_2, c_1, c_0, c_5 \oplus c_4 \oplus 1).$$

The six bits are initialized to zero, and updated *before* being used in a given round. The values of the constants for each round are given in the table below, encoded to byte values for each round, with c_0 being the least significant bit.

Rounds	**Constants**
1 - 16	01,03,07,0F,1F,3E,3D,3B,37,2F,1E,3C,39,33,27,0E
17 - 32	1D,3A,35,2B,16,2C,18,30,21,02,05,0B,17,2E,1C,38
33 - 48	31,23,06,0D,1B,36,2D,1A,34,29,12,24,08,11,22,04

Remark: GIFT aims at single-key security, so we do not claim any related-key security (even though no attack is known in this model as of today). In case one wants to protect against related-key attacks as well, we advice to double the number of rounds.

3 Design Rationale

First, let us propose a subclassification for SPN ciphers.

Definition 1. *Substitution-bitPermutation network (SbPN) is a subclassification of Substitution-Permutation network, where the permutation layer (p-layer) only comprises of bit permutation. An m/n-SbPN cipher is an n-bit cipher in which substitution layer (s-layer) comprises of m-bit (Super-)Sboxes.*

For SPN ciphers like `AES` and `SKINNY`, we can shift the XOR components from the p-layer to the s-layer to form Super-Sboxes, leaving the p-layer with only bit permutation. For example, `PRESENT` is a 4/64-SbPN cipher, `SKINNY-64` is a 16/64-SbPN cipher, and `SKINNY-128` and `AES` are 32/128-SbPN ciphers.

Having that said, `GIFT-64` is a 4/64-SbPN cipher while `GIFT-128` is (probably the first of its kind) a 4/128-SbPN cipher.

3.1 The Designing of `GIFT`

Before we discuss the design rationale of `GIFT`, we would like to share some background story about `GIFT`, its design approach, and its comparison with another `PRESENT`-like ciphers.

The origin of `GIFT`. It all started with a casual remark "What if the Sboxes in `PRESENT` are replaced with some smaller Sboxes, say the `PICCOLO` Sbox? It will be extremely lightweight since the core cipher only has some Sboxes and nothing else...". We quickly tested it but only to realise that the differential bounds became very low because the Sbox does not have differential branching number of 3. That is when we started analyzing the differential characteristics and studying the interaction between the linear layer and the Sbox. Surprisingly, we found that by carefully crafting the linear layer based on the properties of the Sbox, we were able to achieve the same differential bound as `PRESENT` without the constraint of differential branching number of 3. In addition, this result can also be applied to the improve linear cryptanalysis resistance which was lacking in `PRESENT`. Eventually, a small present—`GIFT` was created.

Design approach. It is natural to ask how `GIFT` is different from the other lightweight primitives, especially the recent `SKINNY` family of block ciphers that was proposed at CRYPTO2016. One of the main differences is the design approach. `SKINNY` was designed with a high-security-reduce-area approach, that is to have a strong security property, then try to remove/reduce various components as much as possible. While `GIFT` adopts a small-area-increase-security approach, starting from a small area goal, we try to improve its security as much as possible.

Other `PRESENT`-like ciphers. Besides `PRESENT`, one may also compare `GIFT-64` with `RECTANGLE` since both are 4/64-SbPN ciphers and an improvement on the design of `PRESENT`. `RECTANGLE` was designed to be software friendly and to achieve a better resistance against the linear cryptanalysis as compared to `PRESENT`. However, although its bit permutation (ShiftRow) was designed to be software friendly, little analysis was done on the how differential and linear

characteristics propagate through the cipher. Whereas for GIFT, we study the interplay of the Sbox and the bit permutation to achieve better differential and linear bounds. In addition, the ShiftRow of RECTANGLE achieves full diffusion in 4 rounds at best. Whereas GIFT-64 achieves full diffusion in 3 rounds like PRESENT, which can be proven to be the optimal for 4/64-SbPN ciphers.

3.2 Designing of GIFT Bit Permutation

To better understand the design rationale of the linear layer, we first look at the permutation layer of PRESENT to analyze the issue when the Sbox is replaced with another Sbox that does not have branching number of 3. Next, we show how we can solve this issue by carefully designing the bit permutation.

Linear layer of PRESENT. The bit permutation of PRESENT is given in Table 6.

Table 6. Bit Permutation of PRESENT.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$P(i)$	0	16	32	48	1	17	33	49	2	18	34	50	3	19	35	51
i	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
$P(i)$	4	20	36	52	5	21	37	53	6	22	38	54	7	23	39	55
i	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
$P(i)$	8	24	40	56	9	25	41	57	10	26	42	58	11	27	43	59
i	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
$P(i)$	12	28	44	60	13	29	45	61	14	30	46	62	15	31	47	63

It is known that the bit permutation can be partitioned into 4 independent bit permutations, mapping the output of 4 Sboxes to the input of 4 Sboxes in the next round.

For convenience, we number the Sboxes in i^{th} round as $Sb_0^i, Sb_1^i, ..., Sb_{s-1}^i$, where $s = n/4$. These Sboxes can be grouped in 2 different ways - the Quotient and Remainder groups, Qx and Rx, defined as

- $Qx = \{Sb_{4x}, Sb_{4x+1}, Sb_{4x+2}, Sb_{4x+3}\}$,
- $Rx = \{Sb_x, Sb_{q+x}, Sb_{2q+x}, Sb_{3q+x}\}$, where $q = \frac{s}{4}, 0 \leq x \leq q-1$.

In PRESENT, $n = 64$ and output bits of $Qx^i = \{Sb_{4x}^i, Sb_{4x+1}^i, Sb_{4x+2}^i, Sb_{4x+3}^i\}$ map to input bits of $Rx^{i+1} = \{Sb_x^{i+1}, Sb_{4+x}^{i+1}, Sb_{8+x}^{i+1}, Sb_{12+x}^{i+1}\}$, this group mapping is defined in Table 7, where the entry (l, m) at row rw and column cl denotes that the l^{th} output bit of the Sbox corresponding to the row rw at i^{th} round will map to the m^{th} input bit of the Sbox corresponding to the column cl at $(i+1)^{\text{th}}$ round. For example, suppose $x = 2$, row and column start at 0, then the entry $(3, 2)$ at row 2 and column 3 means that the 3^{rd} output bit of Sb_{10}^i maps to 2^{nd} input bit of Sb_{14}^{i+1}, thus $P(43) = 58$ (see Table 6).

PRESENT bit permutation can be realised in hardware with wires only (no logic gates required). Further, full diffusion is achieved in 3 rounds; from 1 bit

Table 7. PRESENT group mapping from Qx^i to Rx^{i+1}.

Qx^i \ Rx^{i+1}	Sb_x^{i+1}	Sb_{4+x}^{i+1}	Sb_{8+x}^{i+1}	Sb_{12+x}^{i+1}
Sb_{4x}^i	$(0,0)$	$(1,0)$	$(2,0)$	$(3,0)$
Sb_{4x+1}^i	$(0,1)$	$(1,1)$	$(2,1)$	$(3,1)$
Sb_{4x+2}^i	$(0,2)$	$(1,2)$	$(2,2)$	$(3,2)$
Sb_{4x+3}^i	$(0,3)$	$(1,3)$	$(2,3)$	$(3,3)$

to 4, then 4 to 16 and then 16 to 64. But, if there exists Hamming weight 1 to Hamming weight 1 differential transition, or $1-1$ bit differential transition, then there exists consecutive single active bit transitions.

We define $1-1$ bit DDT as a sub-table of the DDT containing Hamming weight 1 differences. Consider some Sbox with the following $1-1$ bit DDT (see Table 8). $\Delta\mathbf{x}$ and $\Delta\mathbf{y}$ denote the differential in the input and output of Sbox respectively. It is evident that this Sbox has differential branch number 2.

Table 8. $1-1$ bit DDT Example 1

$\Delta\mathbf{x}$ \ $\Delta\mathbf{y}$	1000	0100	0010	0001
1000	2	0	0	0
0100	0	0	0	0
0010	0	0	0	0
0001	0	0	0	0

Table 9. $1-1$ bit DDT Example 2

$\Delta\mathbf{x}$ \ $\Delta\mathbf{y}$	1000	0100	0010	0001
1000	0	2	2	0
0100	0	0	0	0
0010	0	0	0	0
0001	0	2	2	0

It is trivial to see that there exists a single active bit path which results in a differential characteristic with single active Sboxes in each round. Let the input differences be at 3^{rd} bit of $Sb_{15}^{(i)}$. According to $1-1$ bit DDT (Table 8), there exists a transition from 1000 to 1000. From the group mapping (Table 7), 3^{rd} output bit of $Sb_{15}^{(i)}$ maps to 3^{rd} input bit of $Sb_{15}^{(i+1)}$. And then the differential continues from 3^{rd} output bit of $Sb_{15}^{(i+1)}$ to 3^{rd} input bit of $Sb_{15}^{(i+2)}$ and so on. Not only that, if there exists any $1-1$ bit transition (not necessarily $1000 \rightarrow 1000$), one can verify that there always exists some differential characteristic with single active Sbox per round for at least 4 consecutive rounds.

To overcome this problem, we propose a new construction paradigm, "Bad Output must go to Good Input" or BOGI in short. We explain this in the context of the differential of an Sbox, but the analysis is same for linear case also.

Bad Output must go to Good Input (BOGI). The existence of the single active bit path is because the bit permutation allows $1-1$ bit transition from some Sbox in i^{th} round to propagate to some Sbox in $(i+1)^{\text{th}}$ round that again would produce $1-1$ bit transition. To overcome such problem, it must be ensured

that such path does not exist. In $1-1$ bit DDT, let us define $\Delta\mathbf{x} = \mathtt{x}_3\mathtt{x}_2\mathtt{x}_1\mathtt{x}_0$ be a good input if the corresponding row has all zero entries, else a bad input. Similarly, we define $\Delta\mathbf{y} = \mathtt{y}_3\mathtt{y}_2\mathtt{y}_1\mathtt{y}_0$ be a good output if the corresponding column has all zero entries, else a bad output. In Table 8, 1000 is both bad input and bad output, rest are good.

Consider another $1-1$ bit DDT in Table 9. Let GI, GO, BI, BO denote the set of good inputs, good outputs, bad inputs and bad outputs respectively. Then, in Table 9, $GI = \{0100, 0010\}$, $GO = \{1000, 0001\}$, $BI = \{1000, 0001\}$ and $BO = \{0100, 0010\}$. Or, if we represent these binary strings by integers considering the position of the "1" (rightmost position is 0) in these strings, we may rewrite $GI = \{2, 1\}$, $GO = \{3, 0\}$, $BI = \{3, 0\}$ and $BO = \{2, 1\}$.

An output belonging to BO (bad ouput) could potentially come from a single bit transition through some Sbox in this round. Thus we want to map this active output bit to some GI (good input) in the next round, which guaranteed that it will not propagate to another $1-1$ bit transition. As a result, it avoids single active bit path in 2 consecutive rounds.

BOGI: Let $|BO| \leq |GI|$ and $\pi_1 : BO \to GI$ be an injective map. To ensure that π_1 is an injective map, it is required that $|BO| \leq |GI|$ (the cardinality of the set BO must be less than or equal to the cardinality of the set GI). Let $\pi_2 : GO \to \pi_1(BO)^C$ (the complement of $\pi_1(BO)$) be another injective map. The map π_1 ensures that "Bad Output must go to Good Input". A combined map $\pi : BO \cup GO \to BI \cup GI$ is defined as $\pi(e) = \pi_1(e)$ if and only if $e \in BO$, otherwise $\pi(e) = \pi_2(e)$. For example, consider the Table 9. The injective maps $\pi_1 : \{2, 1\} \to \{2, 1\}$ and $\pi_2 : \{3, 0\} \to \{3, 0\}$ both have 2 choices which altogether make 4 choices for the combined map π. An example BOGI mapping would be $\pi(0) = 0, \pi(1) = 1, \pi(2) = 2, \pi(3) = 3$, which happens to be an identity mapping. Any choice of π may be used to define the bit permutation. We call these πs *differential BOGI permutations* as derived from $1-1$ bit DDT.

Remark: Similar analysis is done for linear case also. Analogous to $1-1$ bit DDT, analysis is done on the basis of $1-1$ bit LAT and BOGI permutations are found for linear case too. We call them *linear BOGI permutations*. We can now choose any common permutation from the set of both differential and linear BOGI permutations.

BOGI bit permutation for `GIFT`. Let $\pi : \{0, 1, 2, 3\} \to \{0, 1, 2, 3\}$ be a common permutation from the set of both differential and linear BOGI permutations. Table 10 shows the group mapping.

Note that we made some left rotations to the rows of the bit mapping, this is because we need the inputs to each Sbox in $(i+1)^{\text{th}}$ round to be coming from 4 different bit positions.

In `GIFT`, we chose an Sbox that has a common BOGI permutation that is an identity mapping, that is $\pi(i) = i$. Figure 2 illustrates the group mapping from $Q0$ to $R0$ in `GIFT-64`. The same BOGI permutation is applied to all the q group mappings to form the final n-bit bit permutation for both version of `GIFT`.

Table 10. BOGI Bit Permutation mapping from Qx^i to Rx^{i+1}.

Qx^i \ Rx^{i+1}	Sb_x^{i+1}	Sb_{q+x}^{i+1}	Sb_{2q+x}^{i+1}	Sb_{3q+x}^{i+1}
Sb_{4x}^i	$(0, \pi(0))$	$(1, \pi(1))$	$(2, \pi(2))$	$(3, \pi(3))$
Sb_{4x+1}^i	$(1, \pi(1))$	$(2, \pi(2))$	$(3, \pi(3))$	$(0, \pi(0))$
Sb_{4x+2}^i	$(2, \pi(2))$	$(3, \pi(3))$	$(0, \pi(0))$	$(1, \pi(1))$
Sb_{4x+3}^i	$(3, \pi(3))$	$(0, \pi(0))$	$(1, \pi(1))$	$(2, \pi(2))$

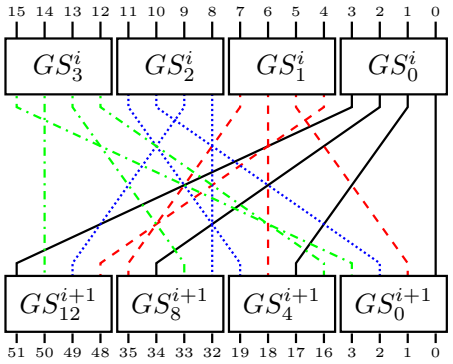

Fig. 2. Group mapping from $Q0$ to $R0$ in `GIFT-64`.

Some results about our bit permutation. Let $Q0, Q1, \cdots, Q(q-1)$ be q different Quotient groups and $R0, R1, \cdots, R(q-1)$ be q different Remainder groups. Then, for $0 \leq x \leq q-1$,

1. The input bits of an Sbox in Rx come from 4 distinct Sboxes in Qx.
2. The output bits of an Sbox in Qx go to 4 distinct Sboxes in Rx.
3. The input bits of 4 Sboxes from the same Qx come from 16 different Sboxes.
4. The output bits of 4 Sboxes from the same Rx go to 16 different Sboxes.

Lemma 1. *When the number of Sboxes in a round is* 16 *or* 32*, the proposed bit permutation achieves an optimal full diffusion which is achievable by a bit permutation.*

Proof. Considering 4 to 4 bits Sbox and a bit permutation in diffusion layer, an input bit to an Sbox will influence 4 output bits which then again influence inputs of 4 Sboxes in the next round influencing altogether 16 bits. By using similar argument, after r rounds, a single bit will influence at most 4^r number of bits. For $4^r \geq 64$, r must be greater than or equal to 3. And for $4^r \geq 128$, $r \geq 4$.

If the number of Sboxes in a round is 16, by using the proposed bit permutation, the number of active Sboxes go from 1 to 16 in two rounds - by using arguments (2) The output bits of an Sbox in Qx group go to 4 distinct Sboxes of the Rx group and (4) The output bits of 4 Sboxes from the same Remainder group Rx

go to 16 different Sboxes. Therefore, a single bit influences 16 Sboxes in two rounds and thus 64 bits in three rounds which is optimal.

It could again be checked that if the number of Sboxes in a round is 32, the proposed bit permutation needs four rounds to achieve the full diffusion, i.e., a single bit influences 128 bits in 4 rounds which is optimal. □

Lemma 2. *In the proposed bit permutation, there does not exist any single active bit transition for two consecutive rounds in both differential and linear characteristics.*

Proof. We prove it by showing with differential only. However, the similar argument holds for linear also. Let the first transition has input $\Delta\mathbf{x}^{(1)}$ and output $\Delta\mathbf{y}^{(1)}$ where both $\Delta\mathbf{x}^{(1)}$ and $\Delta\mathbf{y}^{(1)}$ have hamming weight one. There can be two cases -

1. $\Delta\mathbf{x}^{(1)} \in BI$. Then $\Delta\mathbf{y}^{(1)} \in BO$. Because of BOGI, $BO \rightarrow GI$, and therefore, $\Delta\mathbf{x}^{(2)} \in GI$. Thus, the hamming weight of $\Delta\mathbf{y}^{(2)}$ will be greater than or equal to 2.
2. $\Delta\mathbf{x}^{(1)} \in GI$. Then the hamming weight of $\Delta\mathbf{y}^{(1)}$ will be greater than or equal to 2, contrary to our assumption. □

Definition 2. *The **differential** (resp. **linear**) **score of an Sbox** is* $|GI|+|GO|$ *observed from* $1-1$ *bit DDT (resp. LAT).*

Lemma 3. *There exists differential (resp. linear) BOGI permutation for an Sbox if and only if the differential (resp. linear) score of an Sbox is at least* 4*.*

Proof. As per BOGI, $\pi_1 : BO \rightarrow GI$. Since, π_1 is one-one, $|BO| \leq |GI|$. Furthermore, $|GO|+|BO| = 4$. Combining both, we get, $|GO|+|BO| = 4 \leq |GO|+|GI|$.

Conversely, let $|GO|+|GI| \geq 4$. Since $|GO|+|BO| = 4$, hence $|GO|+|GI| \geq |GO|+|BO|$ which then implies $|GI| \geq |BO|$. □

It is essential that our Sbox has at least score 4 for both differential and linear, and has some common BOGI permutation. These are 2 of the main criteria for the selection of `GIFT` Sbox.

Remark: BOGI permutation is a group mapping that is independent of the number of groups. Thus, this permutation design is scalable to any bit permutation size that is multiple of 16. This allows us to potentially design larger state size like 256-bit that is useful for designing hash functions.

3.3 Selection of `GIFT` Sbox

We first recall some Sbox properties and introduce a metric to estimate the hardware implementation cost of Sboxes.

Properties of Sbox. Let $S : \mathbb{F}_2^4 \to \mathbb{F}_2^4$ denote a 4-bit Sbox. For the differential property, let $\Delta_I, \Delta_O \in \mathbb{F}_2^4$ be the input and output differences, we define

$$D_S(\Delta_I, \Delta_O) = \sharp\{x \in \mathbb{F}_2^4 | S(x) \oplus S(x \oplus \Delta_I) = \Delta_O\},$$

and the maximum all nonzero transitions as

$$D_{max}(S) = \max_{\Delta_I, \Delta_O \neq 0} D_S(\Delta_I, \Delta_O).$$

For the linear property, let $\alpha, \beta \in \mathbb{F}_2^4$ be the input and output masking, we define

$$L_S(\alpha, \beta) = |\sharp\{x \in \mathbb{F}_2^4 | x \bullet \alpha = S(x) \bullet \beta\} - 8|,$$

and the maximum over all nonzero masking as

$$L_{max}(S) = \max_{\alpha, \beta \neq 0} L_S(\alpha, \beta).$$

Definition 3 ([38]). *Let M_i and M_o be two invertible matrices and $c_i, c_o \in \mathbb{F}_2^4$. The Sbox S' defined by $S'(x) = M_o S(M_i(x \oplus c_i)) \oplus c_o$ belongs to the affine equivalence (AE) set of S.*

It is known that both D_{max} and L_{max} are preserved under the AE class.

Definition 4 ([38]). *Let P_i and P_o be two bit permutation matrices and $c_i, c_o \in \mathbb{F}_2^4$. The Sbox S' defined by $S'(x) = P_o S(P_i(x \oplus c_i)) \oplus c_o$ belongs to the permutation-xor equivalence (PE) set of S.*

One is to note that the $1-1$ bit differential and linear transition is preserved only under the PE class. That is to say that the score of an Sbox is preserved under the PE class but not the AE class.

Heuristic Sbox implementation. We use a simplified metric to estimate the implementation cost of Sboxes. We denote $\{\texttt{NOT}, \texttt{NAND}, \texttt{NOR}\}$ as N-operations[1] and $\{\texttt{XOR}, \texttt{XNOR}\}$ as X-operations, and estimate the cost of an N-operation to be 1 unit and X-operations to be 2 units. We consider the following 4 types of instruction for the construction of the Sboxes: $a \leftarrow \texttt{NOT}(a)$; $a \leftarrow a\ \texttt{X}\ b$; $a \leftarrow a\ \texttt{X}\ (b\ \texttt{N}\ c)$; $a \leftarrow a\ \texttt{X}\ ((b\ \texttt{N}\ c)\ \texttt{N}\ d)$, where a, b, c, d are distinct bits of an Sbox input. These so-called *invertible instructions* [25] allow us to implement the inverse Sbox by simply reversing the sequence of the instructions. In addition, the implementation cost of the inverse Sbox would be the same as the direct Sbox since the same set of instructions is used.

Under this metric, we found that PRESENT Sbox requires $4\texttt{N} + 9\texttt{X}$ operations, a cost of 22 units. While RECTANGLE Sbox requires $4\texttt{N} + 7\texttt{X}$ operations, a cost of 18 units. Hence, one of the criteria for our Sbox is to have implementation cost lesser than 18 units[2].

[1] We do not need to consider AND and OR because when we use these invertible instructions, it is equivalent to some other instructions that have been taken into consideration. For instance, $a\ \texttt{XOR}\ (b\ \texttt{AND}\ c) \equiv a\ \texttt{XNOR}\ (b\ \texttt{NAND}\ c)$.

[2] This "unit" metric is to facilitate the Sbox search, the Sboxes are later synthesized to obtain their GE in Section 5.

Search for GIFT Sbox. Our primary design criteria for the GIFT Sbox are:

1. Implementation cost of at most 17 units.
2. With a score of at least 4 in both differential and linear. I.e. For both differential and linear, $|GO| + |GI| \geq 4$.
3. There exists a common BOGI permutation for both differential and linear.

From the list of 302 AE Sboxes presented in [14], we generate the PE Sboxes and check its implementation cost. Our heuristic search shows that there is no optimal Sboxes [31] ($D_{max} = 4$ and $L_{max} = 4$) that satisfies all 3 criteria, hence we extended our search to non-optimal Sboxes. For Sboxes with $D_{max} = 6$ and $L_{max} = 4$, we found some Sboxes with implementation cost of 16 units. For a cost of 15 units, the best possible Sboxes (in terms of D_{max} and L_{max}) that satisfies the criteria have $D_{max} = 12$ and $L_{max} = 6$. And Sboxes with cost of at most 14 units have either $D_{max} = 16$ or $L_{max} = 8$. To maximise the resistance against differential and linear attacks while satisfying the Sbox criteria, we consider Sboxes with $D_{max} = 6$, $L_{max} = 4$ and implementation cost of 16 units.

In order to reduce the occurrence of sub-optimal differential transition, we impose two additional criteria:

4. $\sharp\{(\Delta_I, \Delta_O) \in \mathbb{F}_2^4 \times \mathbb{F}_2^4 | D_S(\Delta_I, \Delta_O) > 4\} \leq 2$.
5. For $D_S(\Delta_I, \Delta_O) > 4$, $wt(\Delta_I) + wt(\Delta_O) \geq 4$, where $wt(\cdot)$ is the Hamming weight.

Criteria (5) ensures that when sub-optimal differential transition occurs, there is a total of at least 4 active Sboxes in the previous and next round.

Finally, we pick an Sbox with a common BOGI permutation for differential and linear that is an identity, i.e. $\pi(i) = i$.

Properties of GIFT Sbox. Our GIFT Sbox GS can be implemented with 4N+6X operations (smaller than the Sboxes in PRESENT and RECTANGLE), has a maximum differential probability of $2^{-1.415}$ and linear bias of 2^{-2}, algebraic degree 3 and no fixed point. For the sub-optimal differential transitions with probability $2^{-1.415}$, there are only 2 such transitions and the sum of Hamming weight of input and output differences is 4. The implementation, differential distribution table (DDT) and linear approximation table (LAT) of GS are provided in Appendix C.

3.4 Designing of GIFT Key Schedule

Key state update. One of our main goals when designing the key schedule is to minimize the hardware area, and thus we chose bit permutation which is just wire shuffle and has no hardware area at all. For it to be also software friendly, we consider the entire key state rotation to be in blocks of 16-bit, and bit rotations within some 16-bit blocks. Since it is redundant to apply bit rotations within key state blocks that have not been introduced to the cipher state, we update the key state blocks only after it has been extracted as a round key.

To introduce the entire key material into the cipher state as fast as possible, the key state blocks that are extracted as the round key are chosen such that all the key material are introduced into the cipher state in the least possible number of rounds.

Adding round keys. To optimize the hardware performances of GIFT, we XOR the round key to only half of the cipher state. This saves a significant amount of hardware area in a round-based implementation. For it to be software friendly too, we XOR the round key at the same i-th bit positions of each nibble. This makes the bitslice implementation more efficient. In addition, since all nibbles contains some key material, the entire state will be dependent on the key after a SubCells operation.

The choice of the positions for adding the round key and 16-bit rotations were chosen to optimize the related-key differential bounds. However, we would like to reiterate that more rounds is advised to resist related-key attacks.

Round constants. For the round constants, but instead of using a typical decimal counter, we use a 6-bit affine LFSR (like in SKINNY [5]). It requires only a single XNOR gate per update which is probably has smallest possible hardware area for a counter. Each of the 6 bits is xored to a different nibble to break the symmetry. In addition, we add a "1" at the MSB to further increase the effect.

4 Security Analysis

In this section, we provide the various cryptanalysis that we had conducted on GIFT.

4.1 Differential and Linear Cryptanalysis

Differential cryptanalysis [9] (DC) and linear cryptanalysis [32] (LC) are among the most powerful techniques available for block ciphers. Analyzing the resistance of a cipher against differential and linear cryptanalysis of a block cipher is perhaps the most common and fundamental security analysis. One of the ways to gauge the resistance of a cipher is to find the lower bound for the number of *active* Sboxes involved in a differential or linear characteristic.

In our work, we use Mixed Integer Linear Programming (MILP) to compute the lower bounds for the number of active Sboxes in both DC and LC for various numbers of rounds, the results are summaries in Table 11. The MILP solution provides us the actual differential or linear characteristics, which allows us to compute the actual differential probability and correlation contribution from the DDT (Table 18) and LAT (Table 19) of GS. In other words, the bounds for number of active Sboxes in Table 11 are strict lower bounds, while the probabilities are heuristic.

Table 11. Lower bounds for number of active Sboxes.

Cipher	DC/LC	Rounds								
		1	2	3	4	5	6	7	8	9
GIFT-64	DC	1	2	3	5	7	10	13	16	18
	LC	1	2	3	5	7	9	12	15	18
PRESENT	DC	1	2	4	6	10	12	14	16	18
	LC	1	2	3	4	5	6	7	8	9
RECTANGLE	DC	1	2	3	4	6	8	11	13	14
	LC	1	2	3	4	6	8	10	12	14
GIFT-128	DC	1	2	3	5	7	10	13	17	19
	LC	1	2	3	5	7	9	12	14	18

Recall that one of our main goals is to match the differential bounds of PRESENT, that is having an average of 2 active Sboxes per round, but with a lighter Sbox and without the constraint of differential branching number of 3. In addition, we aim for the same ratio for the linear bound which was not accomplished by PRESENT. These targets were achieved at 9-round of GIFT. Hence, our DC and LC analysis and discussion focus on 9-round.

Differential cryptanalysis. Generally, for an adversary to mount at DC on an n-bit block cipher using DC, there must be some differential propagation with differential probability larger than 2^{1-n}. An r-round differential is the compiling effect of multiple r-round differential characteristics under the same input and output differences. Thus the differential probability can be computed by simply

taking the summation of the probabilities of all the differential characteristics under the same input and output differences.

To compute a 9-round differential probability of GIFT, we first find a differential characteristic with the least number of active Sboxes. Next, by fixing the input and output differences, we repeatedly search for the next best possible differential characteristic and we sum up the probabilities. The search terminates when the subsequent differential characteristic has insignificant contribution in improving the differential probability further.

For GIFT-64, it has a 9-round differential probability of $2^{-44.415}$, taking the average per round and propagate forward, we expect that the differential probability will be lower than 2^{-63} when we have 14-round[3]. Therefore, we believe 28-round GIFT-64 is enough to resist against DC.

Using the same methodology, we found that PRESENT with 9-round differential probability of $2^{-40.702}$ is expected to require 14-round. While RECTANGLE with 9-round differential probability of $2^{-38.704}$ is expected to require 15-round. It is to note that our estimation matches the belief of the RECTANGLE designers that it is impossible to construct an efficient 15-round differential distinguisher for RECTANGLE [48].

For GIFT-128, it has a 9-round differential probability of $2^{-46.99}$, which suggested that 26-round is sufficient to achieve a differential probability lower than 2^{-127}. Therefore, we believe 40-round GIFT-128 is enough to resist against DC.

For both of the GIFT version, it is interesting to note that in most of the cases, an optimal differential characteristic is the only one with the least number of active Sboxes, subsequent differential characteristics under the same input and output differences have significantly more active Sboxes than the initial characteristic. Thus the differential probability is close to the probability of the optimal differential characteristic. Unlike PRESENT, which due to its symmetry structure, tend to have several optimal differential characteristics for some fixed input and output differences.

Linear cryptanalysis. Given a linear characteristic with a bias ϵ, the square of the correlation contribution (so-called *correlation potential* [19]) is defined as $4\epsilon^2$. For an adversary to mount LC on an n-bit block cipher, she would require the correlation potential to be larger than 2^{-n}. To compute the r-round linear hull effect, which is the compile effect of multiple r-round linear characteristics under the same input and output masking, we use the following theorem.

Theorem 1 ([48]). *The average correlation potential (linear hull effect) between an input and an output selection pattern is the sum of the correlation potentials of all linear trails between the input and output selection patterns.*

[3]Mathematically, we need 13-round to achieve a differential probability lower than 2^{-63}. However, since there is no whitening key at the beginning, the differential probability actually starts from the second round. Hence we added an additional round to the estimation.

Similar to differential, we first find an optimal linear characteristic, fix the input and output masking to find subsequent best possible linear characteristics and take the summation of the correlation potentials. The search is terminated when subsequent linear characteristic has insignificant contribution to the linear hull effect.

For GIFT-64, it has a 9-round linear hull effect of $2^{-49.997}$, which expected to require 13-round[4] to achieve correlation potential lower than 2^{-64}. Therefore, we believe 28-round GIFT-64 is enough to resist against LC.

Using the same methodology, we found that PRESENT with 9-round linear hull effect of $2^{-27.186}$ is expected to require 22-round. While RECTANGLE with 9-round linear hull effect of $2^{-36.573}$ is expected to require 16-round.

For GIFT-128, it has a 9-round linear hull effect of $2^{-45.99}$, which means that we would need around 27 rounds to achieve correlation potential lower than 2^{-128}. Therefore, we believe 40-round GIFT-128 is enough to resist against LC.

Related-key differential cryptanalysis. For GIFT-64, since it takes 4 rounds for all the key material to be introduced into the cipher state, it is trivial to see that it is possible to have no active Sboxes from 1-round to 4-round. Thus we start our computation on the related-key differential bounds from 5-round onwards. From 5-round to 12-round, the probability of these differential characteristics are $2^{-1.415}, 2^{-5}, 2^{-6.415}, 2^{-10}, 2^{-16}, 2^{-22}, 2^{-27}, 2^{-33}$ respectively. Even if we suppose that the probability of 12-round characteristic is lower bounded by 2^{-33}, it is doubtful that 28 rounds are secure against related-key differential cryptanalysis. Therefore, as we mentioned in Section 2, we strongly recommend to increase the number of rounds to achieve the security against the related-key attacks.

For GIFT-128, we start our computation from 3-round onwards. From 3-round to 9-round, the probabilities are $2^{-1.415}, 2^{-5}, 2^{-7}, 2^{-11}, 2^{-20}, 2^{-25}, 2^{-31}$ respectively. Similar to GIFT-64, it is doubtful that 40 rounds are secure against related-key differential cryptanalysis.

We would like to reiterate that our results are found through MILP which minimises the number of active Sboxes, we do not claim optimality for the probabilities.

4.2 Details of Integral Attacks

This section shows the security against integral attacks.

Integral Distinguishers Using Division Property. We first search for integral distinguishers by using the (bit-based) division property, [42, 44] because the division property can find the longest integral distinguisher on similar block cipher PRESENT [45, 47].

[4] Similar to the differential, added an additional round to the estimation.

We first evaluate the propagation of the division property for GIFT Sbox. The algebraic normal form of GIFT Sbox is described as

$$\begin{aligned} y_0 &= 1 + x_0 + x_1 + x_0x_1 + x_2 + x_3, \\ y_1 &= x_0 + x_0x_1 + x_2 + x_0x_2 + x_3, \\ y_2 &= x_1 + x_2 + x_0x_3 + x_1x_3 + x_1x_2x_3, \\ y_3 &= x_0 + x_1x_3 + x_0x_2x_3, \end{aligned}$$

and the propagation of the division property is summarized as Table 12.

Table 12. The possible propagation of the division property for GIFT Sbox.

u \ v	0x0	0x1	0x2	0x4	0x8	0x3	0x5	0x9	0x6	0xA	0xC	0x7	0xB	0xD	0xE	0xF
0x0	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
0x1		x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
0x2		x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
0x4		x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
0x8		x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
0x3		x	x			x	x	x	x	x	x	x	x	x	x	x
0x5			x		x	x	x	x	x	x	x	x	x	x	x	x
0x9				x	x	x	x	x	x	x	x	x	x	x	x	x
0x6				x		x	x	x	x	x	x	x	x	x	x	x
0xA				x	x	x	x	x	x	x	x	x	x	x	x	x
0xC				x	x	x	x	x	x	x	x	x	x	x	x	x
0x7							x					x	x	x	x	x
0xB									x			x	x	x	x	x
0xD					x	x	x	x		x	x	x	x	x	x	x
0xE				x			x	x	x	x	x	x	x	x	x	x
0xF																x

Here, let u and v be the input and output division property, respectively. The propagation from u to v labeled x is possible. Otherwise, the propagation is impossible.

Taking into account the bit-permutation of GIFT, we evaluated the propagation of the division property on reduced-round GIFT. To search for the longest integral distinguisher, we choose only one bit in plaintext as constant, and the others are active. Then, the number of rounds that we can find integral distinguishers is 9 rounds for GIFT-64, and the following is an example.

$$(A^{60}, ACAA) \xrightarrow{9R} ((UUBB)^{16})$$

Here, only 2nd bit in plaintext is constant, and $(4 \times i)$th and $(4 \times i + 1)$th bits in 9-round ciphertexts are balanced. Note that the round key is not XORed with plaintext in the first round. Therefore, we can trivially extend integral distinguishers by one round, and GIFT-64 has 10-round integral distinguishers, respectively.

14-Round Attack on `GIFT-64-128`. We append four rounds to the 10-round integral distinguisher as the key recovery and attack 14-round `GIFT-64-128`. Let $s^r_{i,j}$ be the input of the $(r+1)$th round function. Then, $s^{10}_{i,0}$ and $s^{10}_{i,1}$ are balanced for any $i \in \{0, 1, \ldots, 16\}$. Moreover, the attack is executed by using the partial-sum technique [21].

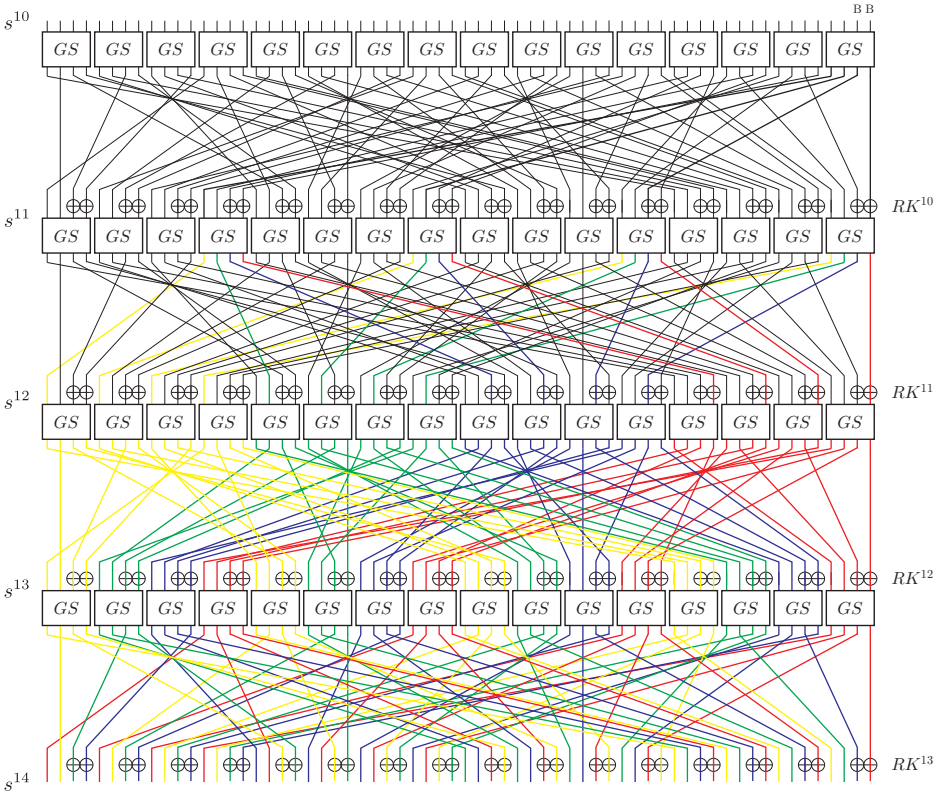

Fig. 3. 4-round key recovery.

We first evaluate whether or not $s^{10}_{0,0}$ and $s^{10}_{0,1}$ are balanced from ciphertexts. To evaluate the balancedness, we need to evaluate the value of

$$(s^{12}_{15,3}, \ldots, s^{12}_{12,3}, s^{12}_{11,2}, \ldots, s^{12}_{8,2}, s^{12}_{7,1}, \ldots, s^{12}_{4,1}, s^{12}_{3,0}, \ldots, s^{12}_{0,0})$$

Please see Fig 3. The value $(s^{12}_{3,0}, \ldots, s^{12}_{0,0})$, which is labeled in red color, are computed from 16 bits in ciphertexts by guessing 16 bits in (RK^{12}, RK^{13}). Therefore, guess 2^{16} round keys involved to red lines for 2^{63} ciphertexts, reduce the memory size from 2^{63} to 2^{52}, and the time complexity is $2^{16+63} = 2^{79}$. Next, additionally guess 2^{16} round keys involved to blue lines for 2^{52} memory, reduce

the memory size from 2^{52} to 2^{40}, and the time complexity is $2^{16\times2+52} = 2^{84}$. Similarly, additionally guess 2^{16} round keys involved to green lines for 2^{40} memory, reduce the memory size from 2^{40} to 2^{28}, and the time complexity is $2^{16\times3+40=88}$. Additionally guess 2^{16} round keys involved to yellow lines for 2^{28} memory, reduce the memory size from 2^{28} to 2^{16}, and the time complexity is $2^{16\times4+28=92}$. Finally, additionally guess 2^{10} round keys in (RK^{10}, RK^{11}) and evaluate whether or not $s^{10}_{0,0}$ and $s^{10}_{0,1}$ are balanced. The time complexity is $2^{16\times4+10+16} = 2^{90}$. In total, the time complexity in the partial-sum technique is about 2^{92}. We repeat this procedure 16 times as changing the bit position where we evaluate the balancedness, and the total time complexity is 2^{96}. As a result, the number of candidates of the secret key is reduced to $2^{128-32=96}$, and we exhaustively guess these keys. Therefore, the total time complexity is about 2^{97} and the data complexity is 2^{63}.

Remarks on `GIFT-128-128`. We also evaluated the longest integral distinguisher for `GIFT-128-128` by using the (bit-based) division property. As a result, we can find 11-round integral distinguisher. The number of rounds is improved by two rounds than that for `GIFT-64-128`. However, the number of bits in round key that is XORed every round increases from 32 bits to 64 bits. Therefore, we expect that `GIFT-128-128` is also secure against integral attacks.

4.3 Impossible Differential Attacks

Impossible differential cryptanalysis [8,27] exploits a pair of difference Δ_1 and Δ_2 in which the state difference Δ_1 never reaches the state difference Δ_2 after some particular rounds. Such Δ_1, Δ_2 are called impossible differentials. In general, several rounds are added before and after the impossible differentials. Given two pairs of plaintext and ciphertext with difference $\Delta P, \Delta C$, the attacker guesses subkey values for the appended rounds, and apply the partial encryption/decryption to the impossible differentials. Subkeys leading to impossible differentials are detected to be wrong.

Given that the `GIFT-64-128` achieves full diffusion only after 3 rounds, there does not exist any 6-round truncated impossible differentials. We then implemented impossible differential search tool based on MILP [18,40], to take into account the differential distribution through the Sbox. We exhaustively tested input and output differences satisfying the following conditions.

- The input difference activates only one of the first four Sboxes.
- The output difference activates only one Sbox.

For the first condition, there are $4 \times 15 = 60$ such input differences. For the second condition, there are $16 \times 15 = 240$ such output differences. Hence, we tested $60 \times 240 = 14,400$ pairs of input and output differences.

The search results show that 11,904 choices out of 14,400 choices are actually impossible. Hence, with a high probability, a pair of input and output differences with 1 active nibble is impossible after 6 rounds. We further extend this search procedure to 7 rounds, and obtained that there does not exist any impossible differential from the 14,400 pairs for 7 rounds.

The number of rounds of impossible differentials is much smaller than the integral attack, thus we omit the detailed analysis of the key recovery part. Full rounds are quite sufficient to resist the impossible differential attack against GIFT.

4.4 Meet-in-the-Middle Attacks

This section shows the security against meet-in-the-middle attacks. The meet-in-the-middle (MITM) attack discussed here is a rather classical one, which separates the encryption algorithm into two independent functions [13, 16]. After the discovery of several techniques against hash functions such as splice-and-cut [1] or initial-structure [39] techniques, MITM attacks can be applied to ciphers with some complex structures. Here, we explain a key recovery attack against 15-round GIFT-64-128 by using the MITM attack.

Chunk Separation. We first observe that GIFT-64-128, in every round, XORs 32 bits out of 128 bits of the master key to the state. In round i mod 1, subkey bits are derived from (k_0, k_1). Although bit positions are rotated for different i, bits from other subkeys are never used in round i mod 1. Similarly, subkey bits in round i mod 2, i mod 3, and i mod 4 are derived from (k_2, k_3), (k_4, k_5) and (k_6, k_7), respectively.

Given this property, 6-round attack can be mounted easily. Namely the first three rounds (rounds 1 to 3) are independent from 64 key bits of k_6, k_7 and the subsequent three rounds (rounds 4 to 6) are independent from 64 key bits of k_4, k_5. Hence, for each guess of k_0, k_1, k_2, k_3 the two parts can be computed independently.

We then extend the number of attacked rounds by using several techniques. As a result of our analysis, we choose that 8 bits of (k_6, k_7) and 8 bits of k_2, k_3 as sources of independent computations. Those bits are called neutral its. We then separate 15 rounds as shown in Fig. 4. Hereafter, we use the notation k_6^F

Round	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Subkey U	k_1	k_3^B	k_5	k_7^F	k_1	k_3^B	k_5	k_7^F	k_1	k_3^B	k_5	k_7^F	k_1	k_3^B	k_5
V	k_0	k_2^B	k_4	k_6^F	k_0	k_2^B	k_4	k_6^F	k_0	k_2^B	k_4	k_6^F	k_0	k_2^B	k_4
Remarks	$\longleftarrow$			IS			$\longrightarrow$			PM			$\longleftarrow$		

Fig. 4. Chunk separation for 15-round MitM attack

and k_7^F to stress that neutral bits in the forward direction is included in k_6 and k_7. Similarly, we use the notations k_2^B, k_3^B.

As shown Fig. 4, the forward computation starts from round 4 while the backward computation starts from round 6. The initial structure (IS) allows us to process those three rounds independently by carefully choosing the position of neutral bits so that the propagation cannot overlap.

From round 7 to round 9, k_2 and k_3 never appear, thus independent computation can be performed easily. Similarly from round 3 to round 1, k_6 and k_7 never appear, thus by fixing the internal state during IS, backward computation can be done up to plaintext independently of the value of k_6 and k_7. Then the attacker uses the splice-and-cut technique, i.e. to make an adoptive chosen plaintext query to obtain the corresponding ciphertext. Then the independent computation can continue until the input of round 13. Finally, we process the partial computation for the middle 3 rounds (round 10 to round 12) and partially match (PM) the results from two directions to efficiently filter unmatched candidates of k_2, k_3 and k_6, k_7.

Initial Structure. We explain that the impact of neutral bits of k_6, k_7 from round 4 to round 6 and the impact of neutral bits of k_2, k_3 from round 6 to round 4 never interact each other, hence the forward computation and backward computation can start from round 4 and from round 6, respectively. The analysis is illustrated in Fig. 5. We first introduce notations S_i^I, S_i^{SC} and S_i^{PB} to denote

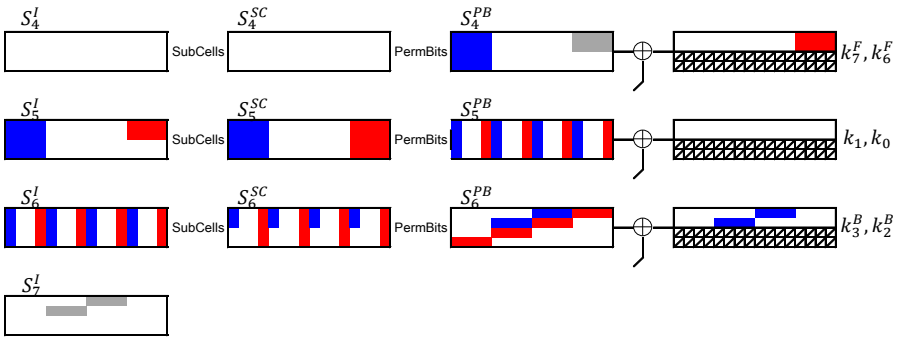

Fig. 5. 3-round initial structure for 15-round MITM attack.

the initial state, the state after SubCells, and the state after PermBits in round i, respectively. In Fig. 5, 64-bit states are denoted in 2-dimensional array which will be detailed in Appendix A. In short, each 4×16 represents 1 bit. The cell in row i and column j where $i \in \{0, 1, 2, 3\}, j \in \{0, 1, \cdots, 15\}$ corresponds to bit position $4j + i$. Thus, S-box is applied to each column.

We choose 4 bits of k_6 in bit positions $0, 1, 2, 3$ and 4 bits of k_7 in bit position $0, 1, 2, 3$ in round 4 as neutral bits in the forward computation. We also choose 4 bits of k_2 in bit positions $4, 5, 6, 7$ and 4 bits of k_3 in bit position $8, 9, 10, 11$ in round 6 as neutral bits in the backward computation. Those bits are XORed to the state by AddRoundKey in round 4 and round 6.

Suppose that 112 subkey bits not chosen as neutral bits are fixed. We then fix the 48 state bits in S_4^{PB} in bit positions 0 to 47, which are not affected by the

backward computation. Because the value of bit positions $0, 1, 4, 5, 8, 9, 12, 13$ of state S_4^{PB} are fixed (gray in Fig. 5), for any 2^8 choices of neutral bits in k_6^F, k_7^F, we can compute the corresponding value of S_5^I. After the SubCells, the first 16 bits of the state will get affected, and those will be eventually moved to bit positions $0, 4, 8, 12, 17, 21, 25, 29, 34, 38, 42, 46, 51, 55, 59, 63$ of S_6^{PB}.

The fixed 96 bits in S_4^{PB} fix 64 bits of the state from S_5^I to S_7^I. We also fix 32 bits of S_6^{PB} in bit positions $3, 7, 11, 15, 16, 20, 24, 28, 33, 37, 41, 45, 50, 54, 58, 62$. This allows to compute bit positions k_2^B, k_3^B. Those 8 bits eventually affect bit positions 48 to 63 of S_5^I. In the end, computations in those three rounds never overlap, which indicates that they can be independently computed.

Partial Matching. After the IS, the forward computation can continue up to round 9 and SubCells and PermBits operations in round 10 until k_2^B, k_3^B appear. The results of 2^8 choices of neutral bits of k_6^F, k_7^F are stored in a list L_F with 2^8 entries. Similarly, the backward computation can be processed until the beginning of round 13 via chosen plaintext query. We then search for the matched pair over the three rounds (rounds 10 to 12) in the middle. The analysis is illustrated in Fig. 6.

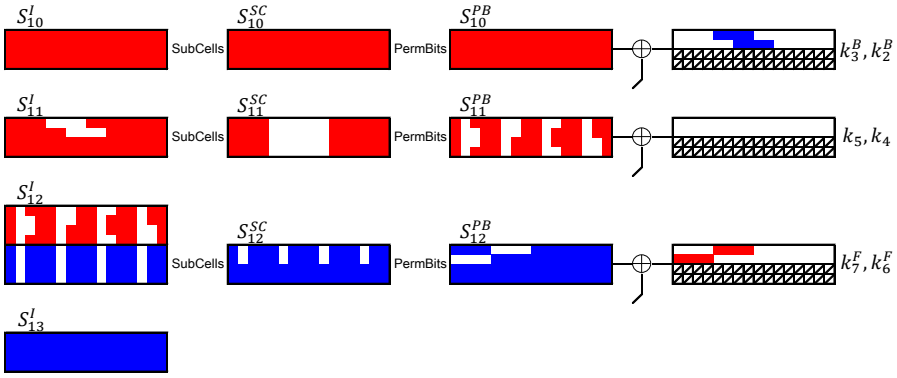

Fig. 6. 3-round partial matching for 15-round MITM attack.

k_2^B and k_3^B in round 10 are rotated from the values in round 6. k_2^B is rotated by 12 bits to right and k_3^B is rotated by 2 bits to right, which moves the neutral bit positions as shown in Fig. 6. Regarding k_6^F and k_7^F in round 12, those are updated by rotation twice compared to the values in round 4. Hence, those are rotated by 24 bits and 4 bits to right respectively.

Partial computations are rather straightforward. In forward direction, 8 unknown bits of subkeys in round 10 hide 24 bits of S_{12}^I, but the attacker can still compute the 40 bits of S_{12}^I. Similarly the attacker can compute the 48 bits

of S_{12}^I in the backward direction. Those overlap in 40 bits, thus can match 40-bit value at S_{12}^I.

Complexity Analysis. For each of 2^{112} values of subkeys not chosen as neutral bits, the attacker compute the forward chunk and backward chunks for 2^8 choices of neutral bits. After the 32-bit match, $2^8 \cdot 2^8 \cdot 2^{-40} = 2^{-24}$ candidates will remain. After 2^{112} iterations, $2^{112} \cdot 2^{-24} = 2^{88}$ key candidates will be obtained, which is verified with exhaustive search. Therefore, the time complexity is $2^{112} \cdot 2^8 + 2^{88} \approx 2^{120}$ and the memory complexity is 2^8. During the backward computation, we make adaptive chosen-plaintext queries. Thus the number of queries is $2^{112} \cdot 2^8 = 2^{120}$ which requires the knowledge of the full codebook. Thus the data complexity is 2^{64}. During the attack, the attacker can refer to this codebook instead of making adoptive chosen plaintext queries.

Remarks on `GIFT-128-128`. In `GIFT-128-128`, 4 key registers are used to update the state in each round, which makes the number of rounds for independent computations shorter. Although the number of rounds for full diffusion is longer than `GIFT-64-128`, this does not extend the length of IS and PM in a simple manner. Due to the large number of rounds in `GIFT-128-128` compared to `GIFT-64-128`, we believe that `GIFT-128-128` has sufficient security margin against the MITM attack.

4.5 Invariant Subspace Attacks

Since the round constant is XORed only in the MSB of several Sboxes, invariant subspace attacks [22,29,30] can be a potential threat. The attack utilizes a linear subspace A and a constant u which is invariant for the round transformation. Its generalized version utilizes the property that the subspace $A \oplus u$ is mapped to another subspace $A' \oplus v$ after the round transformation. Then if round keys and constants belong to the subspace $A \cap A' \oplus u \oplus v$, the state value always stays in the subspace $A \cap A'$ thus the cipher can be distinguished only with a single query.

We searched for the subspace transition through the `GIFT` Sbox. There does not exist any subspace transition with A's dimension 2 and 3. For dimension 1, there are five transitions shown in below:

$$\{0,5\} \oplus \mathtt{a} \xrightarrow{S} \{0,5\} \oplus \mathtt{b}, \quad k \in \{0,5\} \oplus 1,$$
$$\{0,\mathtt{a}\} \oplus 0 \xrightarrow{S} \{0,\mathtt{a}\} \oplus 1, \quad k \in \{0,\mathtt{a}\} \oplus 1,$$
$$\{0,\mathtt{c}\} \oplus 2 \xrightarrow{S} \{0,\mathtt{c}\} \oplus 4, \quad k \in \{0,\mathtt{c}\} \oplus 6,$$
$$\{0,\mathtt{d}\} \oplus 5 \xrightarrow{S} \{0,\mathtt{d}\} \oplus 2, \quad k \in \{0,\mathtt{d}\} \oplus 7,$$
$$\{0,\mathtt{f}\} \oplus 0 \xrightarrow{S} \{0,\mathtt{f}\} \oplus 1, \quad k \in \{0,\mathtt{f}\} \oplus 1.$$

In any case, XORing the constant to MSB, i.e. XORing `8` to some nibble, breaks the invariant subspace, thus `GIFT` resists the invariant subspace attacks.

4.6 Nonlinear Invariant Attacks

Nonlinear invariant attacks [43] are weak-key attacks that can be applied when the round constant is XORed only to some particular bits of nibbles. The core idea is to find a nonlinear approximation of the round transformation with probability one. For the SPN structure, the attacks are mounted when 1) Sbox has the quadratic nonlinear invariant and 2) the linear layer is represented by the multiplication with an orthogonal binary matrix.

The diffusion of `GIFT` (bit permutation) is orthogonal. However, it is not represented by the multiplication with an orthogonal binary matrix. Moreover, we searched for the quadratic nonlinear invariant for `GIFT` Sbox, but there is no such invariant. Therefore, `GIFT` is secure against the nonlinear invariant attacks.

4.7 Algebraic Attacks

We show that algebraic attacks do not threaten `GIFT`. The Sbox GS has algebraic degree 3, and from Table 11 we see that for 9-round differential characteristic of `GIFT`, there are at least 18 active Sboxes. One can easily check that we have $3 \cdot 18 \cdot \lfloor \frac{r}{9} \rfloor > n$, where r is the number of rounds for `GIFT-n`. Moreover,GS is described by 21 quadratic equations in the 8 input/output variables over binary field. The entire system for a fixed-key `GIFT` permutation therefore consists of $16 \cdot r \cdot 21$ quadratic equations in $16 \cdot r \cdot 8$ variables. For example, in the case of `GIFT-64`, there are 9408 quadratic equations in 3584 variables. In comparison, the entire system for a fixed-key `AES` permutation consists of 6400 equations in 2560 variables. While the applicability of algebraic attacks on `AES` remains unclear, those numbers tend to indicate that `GIFT` offers a high level of protection.

5 Hardware Implementation

GIFT is surprisingly efficient and on ASIC platforms across various degrees of serialization. This is mainly due to the extremely lightweight round function that performs key addition on only half of the state and uses a bit permutation as the only diffusion mechanism.

5.1 Round based implementation

GIFT includes various design strategies in order to minimize gate count. GIFT employs key addition to only half of the state and so saves silicon area in the process. SKINNY uses the same mechanism, but it additionally uses an equal amount of XOR gates to add the tweak to the state, and so the number of XOR gates required to construct the roundkey addition layer is equal to that of any cipher employing full state addition. Additionally, like PRESENT, the GIFT diffusion layer consists of a bit permutation instead of a lightweight MDS or AMDS matrix which decreases the area further. However the PRESENT Sbox occupies around 22.5 GE when synthesized with the standard cell library of the STM 90nm logic process. The GIFT Sbox occupies only 16.5 GE using the same library. Although, this is not the most compact choice for Sbox (the 4 bit SKINNY Sbox occupies only 12 GE), overall the area of GIFT is smaller than both SKINNY and PRESENT. Furthermore the keyschedule function used in GIFT is also a bit permutation, and so this module is constructed by a simple wire shuffle and takes no area at all. We can list the following advantages for GIFT when compared to other block ciphers available in literature:

SKINNY No logic required for diffusion layer, no logic required for keyschedule, and only half the amount of gates required for roundkey addition.

PRESENT, RECTANGLE No logic required for keyschedule, half the amount of gates required for roundkey addition, plus GIFT employs a smaller Sbox.

SIMON The SIMON round function also employs roundkey addition of only half of the state. The SIMON non-linear layer is also smaller than GIFT. However SIMON has a heavier key-schedule function due to which its total area exceeds that of GIFT for both the 64 and 128 bit versions.

MIDORI, LED, PICCOLO No logic required for diffusion layer.

In Table 13, we compare the hardware performances of GIFT with other lightweight ciphers. In Figure 7 we list the individual area requirements of the respective components in GIFT. We used scan flip-flops to design the storage elements, which on average saves 1 GE over the combination of a D flip-flop and 2 to 1 multiplexer. For all the designs in the table, the following design flow was adhered to. The ciphers were first implemented in VHDL and a functional simulation was done using the *Mentorgraphics Modelsim* software. Thereafter the design was synthesized using the Standard cell library of the STM 90nm CMOS logic process (CORE90GPHVT v 2.1.a) with the Synopsys Design Compiler, with the compiler being specifically instructed to optimize the circuit for area. A timing simulation was done on the synthesized netlist with 1000 test vectors. The switching activity of each gate of the circuit was collected while running

post-synthesis simulation. The average power was obtained using Synopsys Power Compiler, using the back annotated switching activity.

Table 13. Comparison of performance metrics for round based implementations synthesized with STM 90nm Standard cell library (* Piccolo implemented in dynamic key mode)

	Area (GE)	Delay (ns)	Cycles	TP_{MAX} (MBit/s)	Power (μW) (@10MHz)	Energy (pJ)
GIFT-64-128	1345	1.83	29	1249.0	74.8	216.9
SKINNY-64-128	1477	1.84	37	966.2	80.3	297.0
PRESENT 64/128	1560	1.63	33	1227.0	71.1	234.6
SIMON 64/128	1458	1.83	45	794.8	72.7	327.3
MIDORI 64	1542	2.06	17	1941.7	60.6	103.0
PICCOLO 64/128*	1868	2.32	32	889.9	79.4	254.1
RECTANGLE 64/128	1637	1.61	27	1472.2	76.2	206.0
LED 64/128	1831	5.25	50	243.8	131.3	656.5
GIFT-128-128	1997	1.85	41	1729.7	116.6	478.1
SKINNY-128-128	2104	1.85	41	1729.7	132.5	543.3
SIMON 128/128	2064	1.87	69	1006.6	105.6	728.6
MIDORI 128	2522	2.25	21	2844.4	89.2	187.3
AES 128	7215	3.83	11	3038.2	730.3	803.3

We see that GIFT has the smallest area compared to the other ciphers. From the pie chart, we see that the storage area (which is a fixed cost) took up most of the area percentage, the cipher component (which is the variable) only make up a small percentage to the overall area.

5.2 Serial implementation

The serial implementation of GIFT-64-128 uses a mixed datapath of size 4 bits on the stateside and 16 bits on the keyside. The architecture has been explained in Figure 8. GIFT-128-128 uses a similar architecture: a mixture of 4 bit datapath in the stateside and a 32 bit datapath on the keyside is employed. We also implemented bit serial versions of GIFT as per the techniques outlined in [26]. In Table 14, we list the performance comparisons of GIFT with other block ciphers. While the bit serial implementation of Simon is probably the most compact due to the nature of the design, but the performance of GIFT is comparable/better with other ciphers with similar level of serialization.

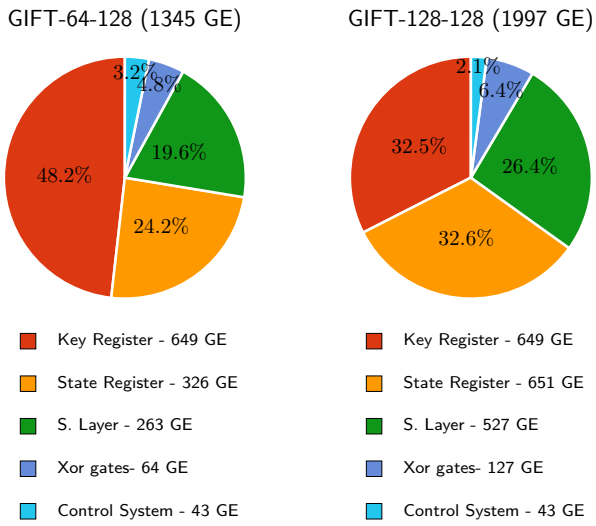

Fig. 7. Componentwise area requirements for `GIFT-64-128` and `GIFT-128-128`

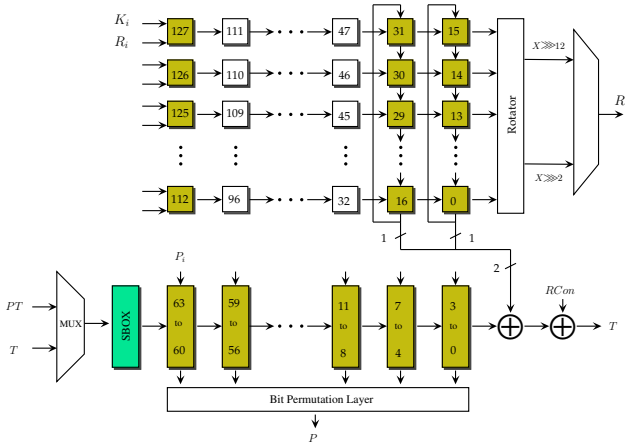

Fig. 8. Serial Implementation for `GIFT-64-128` (The boxes in green denote scan flip-flops/registers)

Table 14. Comparison of performance metrics for serial implementations synthesized with STM 90nm Standard cell library (* `AES` implementation figures from [3])

	Degree of Serialization	Area (GE)	Delay (ns)	Cycles	TP_{MAX} (MBit/s)	Power (μW) (@10MHz)	Energy (nJ)
`GIFT-64-128`	4/16	1113	2.14	522	57.3	39.0	2.04
`GIFT-64-128`	1	930	2.67	2816	8.5	35.9	10.11
`SKINNY`-64-128	4	1265	1.73	756	48.9	59.2	4.48
`SKINNY`-64-128	1	887	0.98	3152	20.7	42.6	13.42
`PRESENT` 64/128	4	1158	1.94	576	57.3	58.0	3.34
`SIMON` 64/128	1	794	1.10	1536	37.9	44.7	6.87
`LED` 64/128	4	1225	2.54	1904	13.2	49.8	9.48
`GIFT-128-128`	4/32	1455	2.25	714	79.7	61.7	4.40
`GIFT-128-128`	1	1213	2.46	6528	8.0	40.3	26.30
`SKINNY`-128-128	8	1638	1.95	840	78.1	79.1	6.64
`SKINNY`-128-128	1	1110	0.81	6976	22.7	53.8	37.53
`SIMON` 128/128	1	1077	1.17	4480	25.1	60.5	27.10
`AES` 128*	8	2060	5.79	246	88.6	129.7	3.19

6 Software Implementation

In this section, we describe our software implementation of `GIFT-64` and `GIFT-128`. Due to its inherent bitslice structure, it seems natural to consider that the most efficient software implementations of `GIFT` will be bitslice implementations.

Thus, whether you are manipulating 64-bit words, 128-bit XMM words, 256-bit YMM words or even 512-bit ZMM words, the first task will be to pack the data into bitslice form. Since the Sbox in both versions of `GIFT` is a 4-bit Sbox, we will need to maintain at least four words to be filled with plaintext data. We prefer to maintain eight such registers, as this will allow us to use the powerful `pshufb` instruction to implement the bit permutation. If one word can contain x blocks of `GIFT` state, our implementation will then encrypt $8x$ blocks at each iteration. For example, implementing `GIFT-128` on 256-bit YMM words will require 16 blocks to be ciphered in parallel. We explain here our software implementation using `GIFT-64` on 128-bit XMM words, but the very same strategy (up to a few details) applies to `GIFT-128` and to bigger words as it directly scales.

Packing/Unpacking. We first load the 16 64-bit blocks $B^0, B^1, \ldots, B^{15}$ into the eight 128-bit registers $R_1, R_2, \ldots, R_8$:

$$\begin{aligned} R_1 =& b_{63}^1\ b_{62}^1\ \ldots\ b_0^1 \parallel b_{63}^0\ b_{62}^0\ \ldots\ b_0^0 \\ R_2 =& b_{63}^3\ b_{62}^3\ \ldots\ b_0^3 \parallel b_{63}^2\ b_{62}^2\ \ldots\ b_0^2 \\ & \ldots \\ R_8 =& b_{63}^{15}\ b_{62}^{15}\ \ldots\ b_0^{15} \parallel b_{63}^{14}\ b_{62}^{14}\ \ldots\ b_0^{14} \end{aligned}$$

The packing and unpacking can be done very simply using the SWAPMOVE process [33]:

$$\begin{aligned} &\text{SWAPMOVE}(A, B, M, N): \\ &\quad T = ((A \gg N) \oplus B)\ \&\ M \\ &\quad B = B \oplus temp \\ &\quad A = A \oplus (T \ll N) \end{aligned}$$

This process will swap the bits in B masked by M, with the bits in A masked by $(M \ll N)$ using only 6 logical operations. Thus, one can pack all the first, second, third and fourth bits of the Sboxes into R_1/R_5, R_2/R_6, R_3/R_7 and R_4/R_8 respectively using:

$\text{SWAPMOVE}(R_1, R_2, \texttt{0xaaa...aa}, 1)$ $\quad$ $\text{SWAPMOVE}(R_5, R_6, \texttt{0xaaa...aa}, 1)$
$\text{SWAPMOVE}(R_3, R_4, \texttt{0xaaa...aa}, 1)$ $\quad$ $\text{SWAPMOVE}(R_7, R_8, \texttt{0xaaa...aa}, 1)$
$\text{SWAPMOVE}(R_1, R_3, \texttt{0xccc...cc}, 2)$ $\quad$ $\text{SWAPMOVE}(R_5, R_7, \texttt{0xccc...cc}, 2)$
$\text{SWAPMOVE}(R_2, R_4, \texttt{0xccc...cc}, 2)$ $\quad$ $\text{SWAPMOVE}(R_6, R_8, \texttt{0xccc...cc}, 2)$

Then, we can regroup these nibbles into bytes:

$\text{SWAPMOVE}(R_1, R_5, \texttt{0xf0...f0}, 4)$ $\quad$ $\text{SWAPMOVE}(R_2, R_6, \texttt{0xf0...f0}, 4)$
$\text{SWAPMOVE}(R_3, R_7, \texttt{0xf0...f0}, 4)$ $\quad$ $\text{SWAPMOVE}(R_4, R_8, \texttt{0xf0...f0}, 4)$

At this point, the bits are grouped into packs of bytes as wanted, but bits of the same plaintext block are spread into different registers, which would slow down the implementation. We regroup them together using a few unpacking instructions like `punpckhbw` and `punpcklbw`.

Inverting the packing is trivial: one only needs to apply the very same SWAPMOVE calls, but in a reverse order (the last byte reordering can be inverted using the `packuswb` packing instruction).

Round function. Once the data in bitslice mode, computing the round function is easy. We provide in Appendix C.1 a software-optimized bitslice implementation of the `GIFT` Sbox, which requires only 6 XORs, 3 ANDs, 1 OR and 1 NOT instruction.

Applying the bit permutation is also trivial now that the data is packed into bytes. Indeed, a crucial property of the `GIFT` bit permutation is that a bit in slice i is always sent to the same slice i during this permutation. Thus, applying the bit permutation layer means simply permuting the ordering of the bytes inside the registers independently. The entire bit permutation can therefore be applied with just one `pshufb` instruction per register.

Subkey addition is performed on all first and second bits of the state. Since several plaintext blocks are usually ciphered under the same key, it seems a good strategy to first precompute all the round subkeys, store them and reuse them when needed with just one memory access. The key schedule can also be performed very efficiently by simply using a `pshufb` instruction to complete the key permutation.

Benchmarks. We have produced this bitslice implementation for AVX2 registers and we give in Table 15 the benchmarking results on a computer with a Intel Haswell processor (i5-4460U). We have benchmarked the bitslice implementations of `SIMON` and `SKINNY` (available online) on the same computer for fairness.

Comments. Bitslice implementations can be used for any parallel mode (as it is the case for most modern operating modes), but can also be used for serial modes when several users are communicating in parallel. In this setting, the implementation would be exactly the same, as our key preparation does not assume that the keys have to be the same for all blocks. In the scenario of a serial mode for a single user, then a classical table-based or VPERM implementation will probably be the most efficient option [6].

For low-end micro-controllers, it is very likely that `GIFT` will perform very well on this platform. `RECTANGLE` is very good on micro-controllers and `GIFT` shares the same general strategy on this regard. The key schedule being even simpler, we believe that it will actually perform even better than `RECTANGLE`.

Table 15. Bitslice software implementations of GIFT and other lightweight block ciphers. Performances are given in cycles per byte, with messages composed of 2000 64-bit blocks to obtain the results.

Cipher	Speed (c/B)	Ref.	Cipher	Speed (c/B)	Ref.
GIFT-64-128	2.10	**new**	GIFT-128-128	2.57	**new**
SKINNY-64-128	2.88	[28]	SKINNY-128-128	4.70	[28]
SIMON-64-128	1.74	[46]	SIMON-128-128	2.55	[46]

Acknowledgements

The authors would like to thank the anonymous referees for their helpful comments. This work is partly supported by the Singapore National Research Foundation Fellowship 2012 (NRF-NRFF2012-06).

References

1. Aoki, K., Sasaki, Y.: Preimage attacks on one-block MD4, 63-step MD5 and more. In: Selected Areas in Cryptography 2008. Volume 5381 of LNCS., Springer (2008) 103–119
2. Banik, S., Bogdanov, A., Isobe, T., Shibutani, K., Hiwatari, H., Akishita, T., Regazzoni, F.: Midori: A Block Cipher for Low Energy. In: ASIACRYPT 2015 - Part II. Volume 9453 of Lecture Notes in Computer Science., Springer (2015) 411–436
3. Banik, S., Bogdanov, A., Regazzoni, F.: Atomic-AES v 2.0. Cryptology ePrint Archive, Report 2016/1005 (2016)
4. Beaulieu, R., Shors, D., Smith, J., Treatman-Clark, S., Weeks, B., Wingers, L.: The SIMON and SPECK Families of Lightweight Block Ciphers. Cryptology ePrint Archive, Report 2013/404 (2013)
5. Beierle, C., Jean, J., Kölbl, S., Leander, G., Moradi, A., Peyrin, T., Sasaki, Y., Sasdrich, P., Sim, S.M.: The SKINNY Family of Block Ciphers and Its Low-Latency Variant MANTIS. In Robshaw, M., Katz, J., eds.: CRYPTO 2016, Proceedings, Part II. Volume 9815 of LNCS., Springer (2016) 123–153
6. Benadjila, R., Guo, J., Lomné, V., Peyrin, T.: Implementing lightweight block ciphers on x86 architectures. In Lange, T., Lauter, K., Lisonek, P., eds.: SAC 2013. Volume 8282 of LNCS., Springer, Heidelberg (August 2014) 324–351
7. Biham, E., Anderson, R.J., Knudsen, L.R.: Serpent: A new block cipher proposal. In Vaudenay, S., ed.: FSE'98. Volume 1372 of LNCS., Springer, Heidelberg (March 1998) 222–238
8. Biham, E., Biryukov, A., Shamir, A.: Cryptanalysis of Skipjack reduced to 31 rounds using impossible differentials. J. Cryptology **18**(4) (2005) 291–311
9. Biham, E., Shamir, A.: Differential cryptanalysis of DES-like cryptosystems. In Menezes, A.J., Vanstone, S.A., eds.: CRYPTO'90. Volume 537 of LNCS., Springer, Heidelberg (August 1991) 2–21
10. Blondeau, C., Nyberg, K.: Links between truncated differential and multidimensional linear properties of block ciphers and underlying attack complexities. In Nguyen,

P.Q., Oswald, E., eds.: EUROCRYPT 2014. Volume 8441 of LNCS., Springer, Heidelberg (May 2014) 165–182
11. Bogdanov, A., Knežević, M., Leander, G., Toz, D., Varici, K., Verbauwhede, I.: Spongent: A lightweight hash function. [37] 312–325
12. Bogdanov, A., Knudsen, L.R., Leander, G., Paar, C., Poschmann, A., Robshaw, M.J.B., Seurin, Y., Vikkelsoe, C.: PRESENT: An ultra-lightweight block cipher. In Paillier, P., Verbauwhede, I., eds.: CHES 2007. Volume 4727 of LNCS., Springer, Heidelberg (September 2007) 450–466
13. Bogdanov, A., Rechberger, C.: A 3-subset meet-in-the-middle attack: Cryptanalysis of the lightweight block cipher KTANTAN. In: SAC 2010. Volume 6544 of LNCS., Springer (2010) 229–240
14. Cannière, C.D.: Analysis and Design of Symmetric Encryption Algorithms. PhD thesis, Katholieke Universiteit Leuven (2007) Bart Preneel (promotor).
15. Cannière, C.D., Dunkelman, O., Knežević, M.: KATAN and KTANTAN - a family of small and efficient hardware-oriented block ciphers. In Clavier, C., Gaj, K., eds.: CHES 2009. Volume 5747 of LNCS., Springer, Heidelberg (September 2009) 272–288
16. Chaum, D., Evertse, J.: Crytanalysis of DES with a reduced number of rounds: Sequences of linear factors in block ciphers. In: CRYPTO 1985. Volume 218 of LNCS., Springer (1985) 192–211
17. Cho, J.Y.: Linear cryptanalysis of reduced-round PRESENT. In Pieprzyk, J., ed.: CT-RSA 2010. Volume 5985 of LNCS., Springer, Heidelberg (March 2010) 302–317
18. Cui, T., Jia, K., Fu, K., Chen, S., Wang, M.: New automatic search tool for impossible differentials and zero-correlation linear approximations. Cryptology ePrint Archive, Report 2016/689 (2016) http://eprint.iacr.org/2016/689.
19. Daemen, J., Rijmen, V.: The Design of Rijndael. Springer-Verlag New York, Inc., Secaucus, NJ, USA (2002)
20. Daemen, J., Rijmen, V.: The Design of Rijndael: AES - The Advanced Encryption Standard. Springer (2002)
21. Ferguson, N., Kelsey, J., Lucks, S., Schneier, B., Stay, M., Wagner, D., Whiting, D.: Improved cryptanalysis of Rijndael. In Schneier, B., ed.: FSE 2000. Volume 1978 of LNCS., Springer (2000) 213–230
22. Guo, J., Jean, J., Nikolic, I., Qiao, K., Sasaki, Y., Sim, S.: Invariant subspace attack against midori64 and the resistance criteria for s-box designs. IACR Transactions on Symmetric Cryptology **2016**(1) (2016) 33–56
23. Guo, J., Peyrin, T., Poschmann, A.: The PHOTON family of lightweight hash functions. In Rogaway, P., ed.: CRYPTO 2011. Volume 6841 of LNCS., Springer, Heidelberg (August 2011) 222–239
24. Guo, J., Peyrin, T., Poschmann, A., Robshaw, M.J.B.: The LED block cipher. [37] 326–341
25. Jean, J., Peyrin, T., Sim, S.M.: Optimizing implementations of lightweight building blocks. Cryptology ePrint Archive, Report 2017/101 (2017)
26. Jean, J. and Moradi, A. and Peyrin T. and Sasdrich, P.: Bit-Sliding: A Generic Technique for Bit-Serial Implementations of SPN-based Primitives. to appear in Cryptographic Hardware and Embedded Systems - CHES 2017 - Taipei, Taiwan, September 25-28, 2017
27. Knudsen, L.: Deal - a 128-bit block cipher. NIST AES Proposal (1998)
28. Kölbl, S.: AVX implementation of the Skinny block cipher. https://github.com/kste/skinny_avx (2016)

29. Leander, G., Abdelraheem, M.A., AlKhzaimi, H., Zenner, E.: A cryptanalysis of PRINTcipher: The invariant subspace attack. In Rogaway, P., ed.: CRYPTO 2011. Volume 6841 of LNCS., Springer (2011) 206–221
30. Leander, G., Minaud, B., Rønjom, S.: A generic approach to invariant subspace attacks: Cryptanalysis of Robin, iSCREAM and Zorro. In Oswald, E., Fischlin, M., eds.: EUROCRYPT 2015, Proceedings, Part I. Volume 9056 of LNCS., Springer (2015) 254–283
31. Leander, G., Poschmann., A.: On the classification of 4 bit S-boxes. In Carlet, C., Sunar, B., eds.: WAIFI 2007. Volume 4547 of LNCS., Springer (2007) 159–176
32. Matsui, M.: Linear cryptoanalysis method for DES cipher. In Helleseth, T., ed.: EUROCRYPT'93. Volume 765 of LNCS., Springer, Heidelberg (May 1994) 386–397
33. May, L., Penna, L., Clark, A.J.: An Implementation of Bitsliced DES on the Pentium MMX^{TM} Processor. In Dawson, E., Clark, A.J., Boyd, C., eds.: Information Security and Privacy, 5th Australasian Conference, ACISP 2000, Brisbane, Australia, July 10-12, 2000, Proceedings. Volume 1841 of Lecture Notes in Computer Science., Springer (2000) 112–122
34. Nakahara, J.: 3D: A three-dimensional block cipher. In Franklin, M.K., Hui, L.C.K., Wong, D.S., eds.: CANS 2008. Volume 5339 of LNCS., Springer (2008) 252–267
35. National Institute of Standards and Technology: Fips 180-2: Secure hash standard. http://csrc.nist.gov
36. National Institute of Standards and Technology: Lightweight cryptography. https://www.nist.gov/programs-projects/lightweight-cryptography (2016)
37. Preneel, B., Takagi, T., eds.: CHES 2011. In Preneel, B., Takagi, T., eds.: CHES 2011. Volume 6917 of LNCS., Springer, Heidelberg (September / October 2011)
38. Saarinen, M.J.O.: Cryptographic analysis of all 4×4-bit S-boxes. In Miri, A., Vaudenay, S., eds.: SAC 2011. Volume 7118 of LNCS., Springer, Heidelberg (August 2012) 118–133
39. Sasaki, Y., Aoki, K.: Finding preimages in full MD5 faster than exhaustive search. In: EUROCRYPT 2009. Volume 5479 of LNCS., Springer (2009) 134–152
40. Sasaki, Y., Todo, Y.: New impossible differential search tool from design and cryptanalysis aspects - revealing structural properties of several ciphers. In Coron, J., Nielsen, J.B., eds.: EUROCRYPT 2017, Part III. Volume 10212 of LNCS. (2017) 185–215
41. Shibutani, K., Isobe, T., Hiwatari, H., Mitsuda, A., Akishita, T., Shirai, T.: Piccolo: An ultra-lightweight blockcipher. [37] 342–357
42. Todo, Y.: Structural evaluation by generalized integral property. In: EUROCRYPT 2015, Part I. Volume 9056 of LNCS., Springer (2015) 287–314
43. Todo, Y., Leander, G., Sasaki, Y.: Nonlinear invariant attack - practical attack on full SCREAM, iSCREAM, and Midori64. In Cheon, J.H., Takagi, T., eds.: ASIACRYPT 2016 Part II. Volume 10032 of LNCS. (2016) 3–33
44. Todo, Y., Morii, M.: Bit-based division property and application to Simon family. In Peyrin, T., ed.: FSE 2016. Volume 9783 of LNCS., Springer (2016) 357–377
45. Todo, Y., Morii, M.: Compact representation for division property. In Foresti, S., Persiano, G., eds.: CANS 2016. Volume 10052 of LNCS. (2016) 19–35
46. Wingers, L.: Software for SUPERCOP Benchmarking of SIMON and SPECK. https://github.com/lrwinge/simon_speck_supercop (2015)
47. Xiang, Z., Zhang, W., Bao, Z., Lin, D.: Applying MILP method to searching integral distinguishers based on division property for 6 lightweight block ciphers. In Cheon, J.H., Takagi, T., eds.: ASIACRYPT 2016 Part I. Volume 10031 of LNCS. (2016) 648–678

48. Zhang, W., Bao, Z., Lin, D., Rijmen, V., Yang, B., Verbauwhede, I.: Rectangle: a bit-slice lightweight block cipher suitable for multiple platforms. Science China Information Sciences **58**(12) (2015) 1–15

A GIFT in 2-Dimensional Array

This section provides an alternative description of GIFT that resembles the RECTANGLE description.

A.1 Initialization

The plaintext is arranged into 4 rows of 16/32 bits in a top-down, right to left manner. The cipher state is described in a two-dimensional array, as illustrated below.

$$\begin{bmatrix} b_{n-4} & ... & b_8 & b_4 & b_0 \\ b_{n-3} & ... & b_9 & b_5 & b_1 \\ b_{n-2} & ... & b_{10} & b_6 & b_2 \\ b_{n-1} & ... & b_{11} & b_7 & b_3 \end{bmatrix} \Rightarrow \begin{bmatrix} s_{0,\frac{n}{4}-1} & ... & s_{0,2} & s_{0,1} & s_{0,0} \\ s_{1,\frac{n}{4}-1} & ... & s_{1,2} & s_{1,1} & s_{1,0} \\ s_{2,\frac{n}{4}-1} & ... & s_{2,2} & s_{2,1} & s_{2,0} \\ s_{3,\frac{n}{4}-1} & ... & s_{3,2} & s_{3,1} & s_{3,0} \end{bmatrix}$$

The key, on the other hand, is arranged in a right to left, top-down manner.

$$\begin{bmatrix} k_{31} & ... & k_2 & k_1 & k_0 \\ k_{63} & ... & k_{34} & k_{33} & k_{32} \\ k_{95} & ... & k_{66} & k_{65} & k_{64} \\ k_{127} & ... & k_{98} & k_{97} & k_{96} \end{bmatrix} \Rightarrow \begin{bmatrix} t_{0,31} & ... & t_{0,2} & t_{0,1} & t_{0,0} \\ t_{1,31} & ... & t_{1,2} & t_{1,1} & t_{1,0} \\ t_{2,31} & ... & t_{2,2} & t_{2,1} & t_{2,0} \\ t_{3,31} & ... & t_{3,2} & t_{3,1} & t_{3,0} \end{bmatrix}$$

A.2 The Round Function

SubCells. Both versions of GIFT use the same invertible 4-bit Sbox, GS. The Sbox is applied in parallel to every columns of the state.

$$(s_{3,j}\|s_{2,j}\|s_{1,j}\|s_{0,j}) \leftarrow GS(s_{3,j}\|s_{2,j}\|s_{1,j}\|s_{0,j}),\ \forall j \in \{0, ..., \frac{n}{4} - 1\}.$$

PermBits. Four different bit permutations are applied to the rows of the cipher state independently. It maps bits from bit position (i, j) to bit position $(i, P_i(j))$. Refer to Table 16 and 17 for each row of the bit permutation for GIFT-64 and GIFT-128 respectively.

$$s_{i,P_i(j)} \leftarrow s_{i,j},\ \forall i \in \{0, ..., 3\},\ \forall j \in \{0, ..., \frac{n}{4} - 1\}.$$

AddRoundKey. A round key of length $n/2$ is extracted from the key state and XORed to 2 rows of the cipher state. For GIFT-64, the key state row 0 is extracted, the first 16 bits are XORed to the cipher state row 0 while the next 16 bits are XORed to the cipher state row 1.

$$\begin{aligned} s_{0,j} &\leftarrow s_{0,j} \oplus t_{0,j} \\ s_{1,j} &\leftarrow s_{1,j} \oplus t_{0,j+16},\ \forall j \in \{0, ..., 15\}. \end{aligned}$$

Table 16. Specifications of `GIFT-64` bit permutation for $s_{i,j}$.

j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$P_0(j)$	0	12	8	4	1	13	9	5	2	14	10	6	3	15	11	7
$P_1(j)$	4	0	12	8	5	1	13	9	6	2	14	10	7	3	15	11
$P_2(j)$	8	4	0	12	9	5	1	13	10	6	2	14	11	7	3	15
$P_3(j)$	12	8	4	0	13	9	5	1	14	10	6	2	15	11	7	3

Table 17. Specifications of `GIFT-128` bit permutation for $s_{i,j}$.

j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
$P_0(j)$	0	24	16	8	1	25	17	9	2	26	18	10	3	27	19	11	4	28	20	12	5	29	21	13	6	30	22	14	7	31	23	15
$P_1(j)$	8	0	24	16	9	1	25	17	10	2	26	18	11	3	27	19	12	4	28	20	13	5	29	21	14	6	30	22	15	7	31	23
$P_2(j)$	16	8	0	24	17	9	1	25	18	10	2	26	19	11	3	27	20	12	4	28	21	13	5	29	22	14	6	30	23	15	7	31
$P_3(j)$	24	16	8	0	25	17	9	1	26	18	10	2	27	19	11	3	28	20	12	4	29	21	13	5	30	22	14	6	31	23	15	7

For `GIFT-128`, the key state row 0 and 2 are extracted and XORed to the cipher state row 1 and 2 respectively.

$$s_{1,j} \leftarrow s_{1,j} \oplus t_{0,j}$$
$$s_{2,j} \leftarrow s_{2,j} \oplus t_{1,j},\ \forall j \in \{0, ..., 31\}.$$

In addition to the round key, for both version of `GIFT`, a single bit "1" is XORed to the last bit of the cipher state row 3 and a 6-bit round constant $(c_5\|c_4\|c_3\|c_2\|c_1\|c_0)$ is XORed to first 6 bits of the cipher state row 3.

$$s_{3,\frac{n}{4}-1} \leftarrow s_{3,\frac{n}{4}-1} \oplus 1,$$
$$s_{3,j} \leftarrow s_{3,j} \oplus c_j,\ \forall j \in \{0, ..., 5\}.$$

A.3 The Key Schedule and Round Constants

The key state update and round constants are the same for both versions of `GIFT`, the round key is *first* extracted from the key state before the key state update.

The key state update first rotates 2 blocks of 16 bits of the key state row 0 independently as follows,

$$(t_{0,11}\|t_{0,10}\|...\|t_{0,13}\|t_{0,12}) \xleftarrow{\ggg 12} (t_{0,15}\|t_{0,14}\|...\|t_{0,1}\|t_{0,0})$$
$$(t_{0,17}\|t_{0,16}\|...\|t_{0,19}\|t_{0,18}) \xleftarrow{\ggg 2} (t_{0,31}\|t_{0,30}\|...\|t_{0,17}\|t_{0,16}).$$

Next, the key state rows are rotated upwards to form the final key state.

The entire key state update is depicted in the following:

$$\begin{bmatrix} t_{0,31} \ldots t_{0,16}\ t_{0,15} \ldots t_{0,0} \\ t_{1,31} \ldots t_{1,16}\ t_{1,15} \ldots t_{1,0} \\ t_{2,31} \ldots t_{2,16}\ t_{2,15} \ldots t_{2,0} \\ t_{3,31} \ldots t_{3,16}\ t_{3,15} \ldots t_{3,0} \end{bmatrix} \leftarrow \begin{bmatrix} t_{1,31} \ldots t_{1,16}\ t_{1,15} \ldots\ t_{1,0} \\ t_{2,31} \ldots t_{2,16}\ t_{2,15} \ldots\ t_{2,0} \\ t_{3,31} \ldots t_{3,16}\ t_{3,15} \ldots\ t_{3,0} \\ t_{0,17} \ldots t_{0,18}\ t_{0,11} \ldots t_{0,12} \end{bmatrix}$$

The round constants are generated using a 6-bit affine LFSR, whose state is denoted as $(c_5, c_4, c_3, c_2, c_1, c_0)$. Its update function is defined as:

$$(c_5, c_4, c_3, c_2, c_1, c_0) \leftarrow (c_4, c_3, c_2, c_1, c_0, c_5 \oplus c_4 \oplus 1).$$

The six bits are initialized to zero, and updated *before* being used in a given round.

B GIFT in 3-Dimensional Cuboid

This section is not a formal description of GIFT but just an alternative way to visualise the GIFT structure, especially the bit permutation.

B.1 GIFT-64 Structure

For each nibble, bit 0 is placed in the slice 0, bit 1 in slice 1, bit 2 in slice 2 and bit 3 in slice 3. As shown in Figure 9.

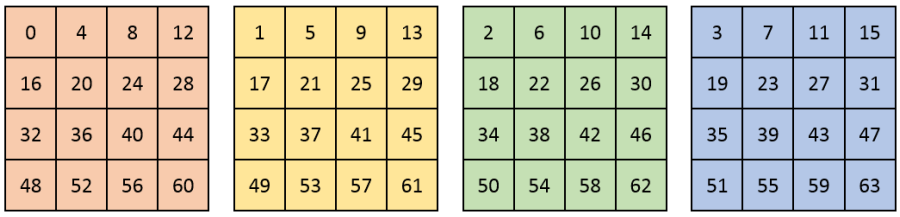

Fig. 9. Position of the bits. Slice 0,1,2,3 are in red, yellow, green, blue respectively.

These slices can be placed together to form a cuboid. As shown in Figure 10.

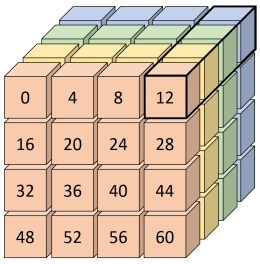

Fig. 10. Cubic representation of the main state of GIFT-64. The black cuboid is where an Sbox is implemented.

Subcells. 16 Sboxes are implemented in parallel in the bitslice manner, as seen in Figure 10.

PermBits. The bit permutation can be implemented as follows:

1. Take the transpose of each individual slice (see Figure 11).
2. Apply row swap to each slice,
   - Slice 0: swap row 1 with 3
   - Slice 1: swap row 0 with 1, and swap row 2 with 3

- Slice 2: swap row 0 with 2
- Slice 3: swap row 0 with 3, and swap row 1 with 2

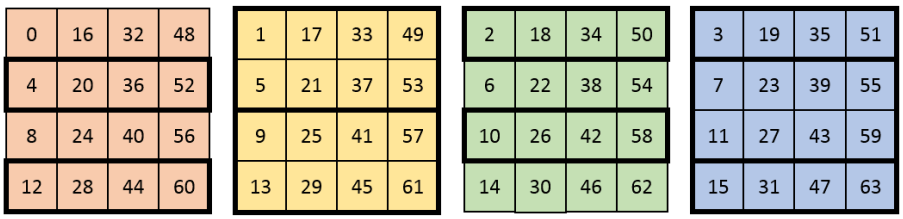

Fig. 11. The slices after transpose. The black boxes are the rows that will be swapped.

The final bit positions after the PermBits can be seen in Figure 12.

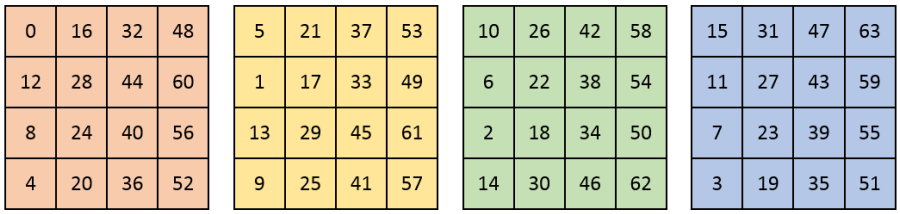

Fig. 12. The final bit positions after PermBits.

B.2 `GIFT-128` Structure

Two slices are needed to pack bit i from each nibble, forming 2 cuboids as seen in Figure 13. Slice 0 and 4 contains all the bit 0, slice 1 and 5 contains all the bit 1, slice 2 and 6 contains all the bit 2, and slice 3 and 7 contains all the bit 3.

SubCells. 32 Sboxes are implemented in parallel in the bitslice manner.

PermBits. The bit permutation can be implemented as follows:

1. Take the transpose of each individual slice (see Figure 14).
2. Shuffle each pair of slices containing same the bit i (see Figure 15).
3. Apply swap between each pair of slices (see Figure 16),
   - Slice 0 and 4: swap the 2 bottom halves
   - Slice 1 and 5: swap the top and bottom halves of the slices independently
   - Slice 2 and 6: swap the 2 top halves
   - Slice 3 and 7: cross swap the top and bottom halves

The final bit positions after the PermBits can be seen in Figure 17.

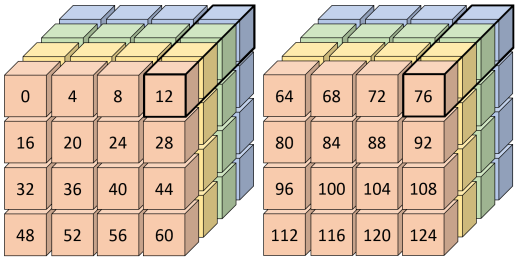

Fig. 13. Cubic representation of the main state of `GIFT-128`. The black cuboid is where an Sbox is implemented.

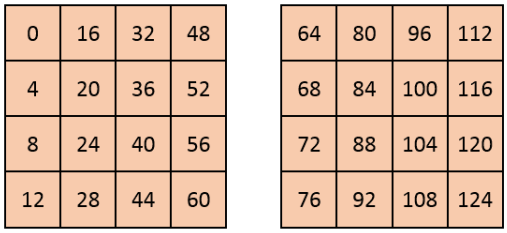

Fig. 14. Slice 0 and 4 after transpose.

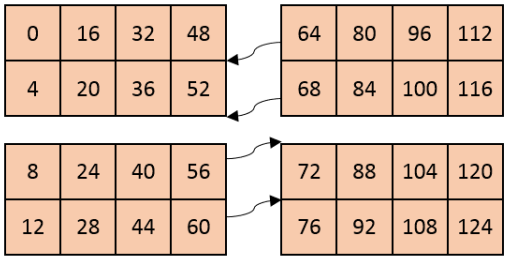

Fig. 15. Shuffling Slice 0 and 4.

B.3 Key State Structure

The key state structure is different from the cipher state structure, the first 16 bits are loaded into key slice 0, next 16 bits are loaded into key slice 1 and so on for all the 8 slices. The key slices are placed together to form a cuboid as seen in Figure 18.

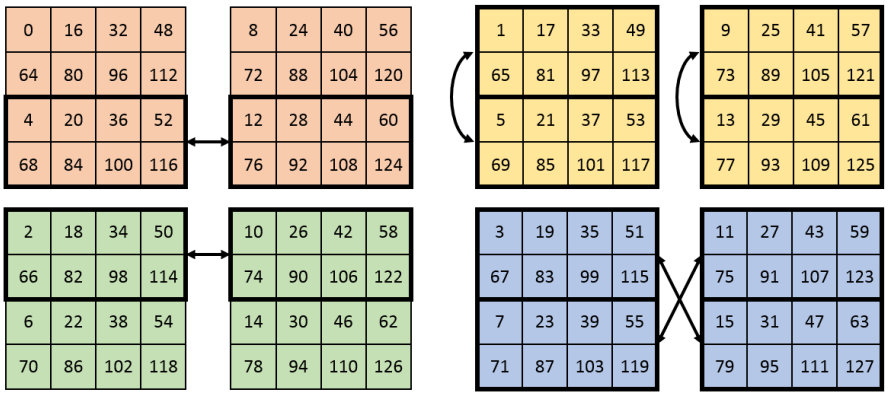

Fig. 16. Swapping the slices.

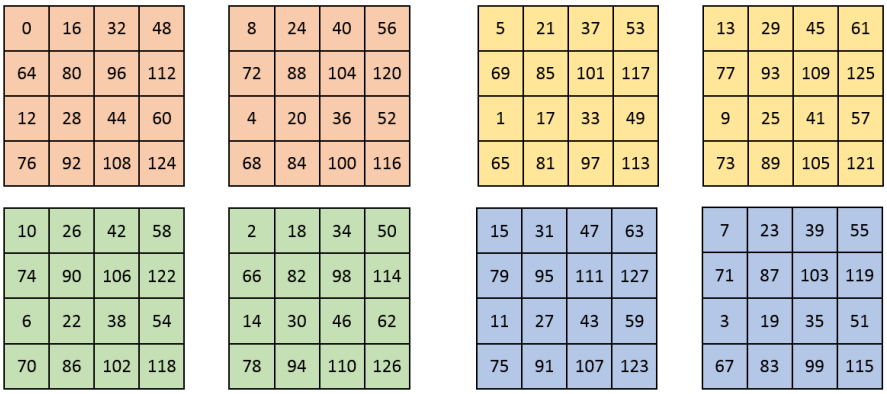

Fig. 17. Final slice arrangement.

For `GIFT-64`, key slice 0 is XORed to state slice 0 and key slice is XORed to state slice 1 in a bitwise manner. For `GIFT-128`, key slice 0 and 1 are XORed to state slice 1 and 5, while key slice 4 and 5 are XORed to state slice 2 and 6.

The key state is then updated as follows:

1. Slice 0: shift 12 bits towards LSB
2. Slice 1: shift 2 bits towards LSB
3. Move slice 0 and 1 to the back of slice 7

Figure 19 shows the bit positions of key slice 0 and 1 after the bit rotations, the other 6 slices remain unchanged.

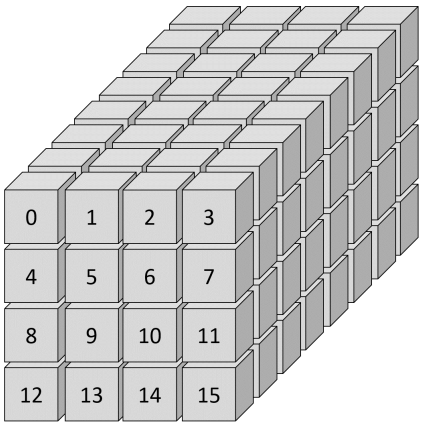

Fig. 18. Cubic representation of the key slices.

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

18	19	20	21
22	23	24	25
26	27	28	29
30	31	16	17

Fig. 19. Key slice 0 and 1 after rotation.

C Details of GIFT Sbox

C.1 GIFT Sbox Implementation

```
/* Input: (MSB) x[3], x[2], x[1], x[0] (LSB) */
x[1] = x[1] XNOR (x[0] NAND x[2]);
x[0] = x[0] XNOR (x[1] NAND x[3]);
x[2] = x[2] XNOR (x[0] NOR x[1]);
x[3] = x[3] XNOR x[2];
x[1] = x[1] XNOR x[3];
x[2] = x[2] XNOR (x[0] NAND x[1]);
/* Output: (MSB) x[0], x[2], x[1], x[3] (LSB) */
```

Fig. 20. Area-optimized hardware implementation of the GIFT Sbox.

```
/* Input: (MSB) x[3], x[2], x[1], x[0] (LSB) */
x[1] = x[1] XOR (x[0] AND x[2]);
t    = x[0] XOR (x[1] AND x[3]);
x[2] = x[2] XOR (t OR x[1]);
x[0] = x[3] XOR x[2];
x[1] = x[1] XOR x[0];
x[0] = NOT x[0];
x[2] = x[2] XOR (t AND x[1]);
x[3] = t;
/* Output: (MSB) x[3], x[2], x[1], x[0] (LSB) */
```

Fig. 21. Software-optimized implementation of the GIFT Sbox.

C.2 GIFT Sbox DDT and LAT

Table 18. Differential Distribution Table (DDT) of GIFT Sbox.

Δ_I \ Δ_O	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0	16	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	2	2	0	2	2	2	2	2	0	0	2
2	0	0	0	0	0	4	4	0	0	2	2	0	0	2	2	0
3	0	0	0	0	0	2	2	0	2	0	0	2	2	2	2	2
4	0	0	0	2	0	4	0	6	0	2	0	0	0	2	0	0
5	0	0	2	0	0	2	0	0	2	0	0	0	2	2	2	4
6	0	0	4	6	0	0	0	2	0	0	2	0	0	0	2	0
7	0	0	2	0	0	2	0	0	2	2	2	4	2	0	0	0
8	0	0	0	4	0	0	0	4	0	0	0	4	0	0	0	4
9	0	2	0	2	0	0	2	2	2	0	2	0	2	2	0	0
a	0	4	0	0	0	0	4	0	0	2	2	0	0	2	2	0
b	0	2	0	2	0	0	2	2	2	2	0	0	2	0	2	0
c	0	0	4	0	4	0	0	0	2	0	2	0	2	0	2	0
d	0	2	2	0	4	0	0	0	0	0	2	2	0	2	0	2
e	0	4	0	0	4	0	0	0	2	2	0	0	2	2	0	0
f	0	2	2	0	4	0	0	0	0	2	0	2	0	0	2	2

Table 19. Linear approximation table (LAT) of GIFT Sbox. Each entry represents $\sharp\{x \in \mathbb{F}_2^4 | x \bullet \alpha = S(x) \bullet \beta\} - 8$.

α \ β	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0	8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	2	-2	-2	2	4	0	0	-4	-2	-2	-2	-2
2	0	0	0	-4	0	4	0	0	2	2	2	-2	2	-2	2	2
3	0	0	0	-4	-2	-2	2	-2	2	-2	-2	-2	0	4	0	0
4	0	0	0	0	0	0	-4	-4	0	0	0	0	0	0	4	-4
5	0	0	0	0	2	-2	2	-2	0	4	4	0	2	2	-2	-2
6	0	0	0	-4	4	0	0	0	-2	-2	-2	2	2	-2	-2	-2
7	0	0	0	4	2	2	2	-2	2	-2	-2	-2	4	0	0	0
8	0	0	0	0	0	-4	0	-4	0	0	0	0	0	-4	0	4
9	0	0	0	0	-2	-2	2	2	4	0	0	4	2	-2	2	-2
a	0	0	-4	0	0	0	4	0	-2	-2	2	-2	-2	-2	2	-2
b	0	0	4	0	2	-2	2	2	-2	2	-2	-2	0	0	4	0
c	0	4	4	0	0	0	0	0	0	-4	4	0	0	0	0	0
d	0	-4	4	0	-2	2	2	-2	0	0	0	0	-2	-2	-2	-2
e	0	-4	0	0	4	0	0	0	2	-2	2	2	-2	2	2	2
f	0	-4	0	0	-2	-2	-2	2	-2	-2	2	-2	4	0	0	0