The Littlun S-box and the Fly block cipher

Pierre Karpman❦ Benjamin Grégoire✧

Abstract

We present the construction and implementation of an 8-bit S-box with a differential and linear branch number of 3. We show an application by designing Fly, a simple block cipher based on bitsliced evaluations of the S-box and bit rotations that targets the same platforms as Pride, and which can be seen as a variant of Present with 8-bit S-boxes. The round function of Fly achieves the same performance as the one of Pride on 8-bit microcontrollers (in terms of number of instructions per round and code size) while having 1.5 times more equivalent active S-boxes on average. The S-box also has an efficient implementation with SIMD instructions, a low implementation cost in hardware and it can be masked efficiently thanks to its sparing use of non-linear gates and to the fact that it has a natural expression in terms of a single 4-bit S-box.

Keywords. Block cipher design, Lai-Massey S-box, bitsliced implementation, SPN.

1 Introduction

Since the late 1990's and the end of the AES competition, the academic community and the industry have been provided with excellent block ciphers. In most cases where a cipher is needed, AES [32] can readily be used and there is currently little need for a replacement. Consequently, the symmetric cryptographic community shifted focus to *e.g.* the wider picture of *authenticated encryption* through the Caesar competition, or to more specific applications of block ciphers. In the latter case, an important topic is the design of "lightweight" block ciphers intended to be implemented on low-cost, resource-constraint devices. An early successful example following this trend is the block cipher Present [10], which can be implemented in small hardware circuits. Most lightweight algorithms similarly target one (or a few) platform(s) on which they are expected to perform particularly well; good performance in other cases are however not usually expected and lightweight ciphers are in general not very versatile. Typical platforms of interest include hardware circuits and 8-bit to 32-bit microcontrollers.

In this work, we design a conceptually simple block cipher targeting efficient *light* implementations on 8-bit microcontrollers[1]. The chief academic proposal to date for this scenario is the Pride block cipher[2], that was presented at CRYPTO 2014 [1]. Our block cipher is built around Littlun-1, a compact 8-bit S-box with branch number 3. This allows to define a round function similar to a scaled-up variant of Present, composing the S-box application with a simple bit permutation[3]. This offers a trade-off between hardware and light software implementations: Littlun-1 is more expensive in hardware than (two applications of) the S-box of Present, but the bit permutation is simple to implement with 8-bit rotations. Owing to Golding, we name our block cipher "Fly".

Excluding on-the-fly key expansion, the round function of Fly costs 4 instructions less to implement than Pride's on AVR. Using the good branch number of Littlun-1, we can show that with a similar number of rounds, Fly is more resistant than Pride to statistical (differential and linear) attacks. This is all the more relevant as the security margin of Pride seems to be quite thin [39]. Taking the key-schedule into account, one round of Fly costs 8 more instructions than one round of Pride. However, unlike Pride, we do not use an FX construction for the key-schedule and thus the generic

❦Centrum Wiskunde & Informatica, The Netherlands; École polytechnique, France; Nanyang Technological University, Singapore, pierre.karpman@gmail.com. Part of this work was done when the author was at Inria.

✧Inria, France, benjamin.gregoire@inria.fr.

[1] By *light* implementations, we mean in particular that the size of the code is small, typically of the order of 200 bytes. If more ressources are available, the best current block cipher is probably the AES (see *e.g.* [7]).

[2] Notable "non-academic" ciphers for the same scenario are the "NSA ciphers" Simon and Speck [6].

[3] As a bit permutation obviously does not increase the number of active bits, an important part of the diffusion in such a cipher is played by the S-box. The typical measure of the quality of the diffusion of an S-box is its "branch number" which plays a role similar to the minimum distance of the linear codes used in AES-like designs.

security of FLY does not decrease with the amount of data available to the adversary (Dinur also showed how the FX construction can lead to more efficient time-memory-data trade-offs [23]).

As implementations on resource-constraint devices are more likely to be vulnerable to side-channel attacks, one should also consider the additional cost of protection against, say, differential power analysis when evaluating schemes that target such platforms. In that respect, the small number of gates necessary to implement the LITTLUN-1 S-box, as well as its simple expression in terms of light 4-bit S-boxes allows one to produce masked implementations of FLY with limited overhead.

Related work. The block cipher literature is so numerous that most new proposal will bear some similarity with past designs. In that respect, apart from PRESENT, FLY is quite similar to RECTANGLE [38], which also combines a SERPENT-like bitsliced application of an S-box [8] with a rotation-implemented bit permutation. However, the S-box in RECTANGLE is on 4 bits, it does not have a branch number of 3 and the rotations are on 16-bit words. The construction of the LITTLUN S-box uses the Lai-Massey structure from the IDEA block cipher [29]; this structure was already used to build the second S-box of the WHIRLPOOL hash function [4] and the S-box of the block cipher FOX [27].

2 Preliminaries

We start by defining the main notions that will be used in evaluating the cryptographic properties of our construction. Although we will mostly consider S-boxes as defined over binary strings, we may see an n-bit S-box as a mapping $\mathbf{F}_2^n \to \mathbf{F}_2^n$ whenever convenient.

Definition 2.1 (Differential uniformity of an S-box)**.** Let $\mathcal{S}$ be an n-bit S-box. We define its *difference distribution table* (or DDT) as the function $\delta_{\mathcal{S}}$ defined extensively by:

$$\delta_{\mathcal{S}}(a,b) := \#\{x \in \mathbf{F}_2^n \,|\, \mathcal{S}(x) + \mathcal{S}(x+a) = b\}.$$

The *differential uniformity* Δ of $\mathcal{S}$ is defined as:

$$\max_{(a,b)\neq(0,0)} \delta_{\mathcal{S}}(a,b).$$

Put another way, an n-bit S-box with differential uniformity Δ has a maximal differential probability of $\Delta/2^n$ over its inputs.

Definition 2.2 (Linearity of an S-box)**.** Let $\mathcal{S}$ be an n-bit S-box. We define its *linear approximation table* (or LAT) as the function $\mathcal{L}_{\mathcal{S}}$ defined extensively by:

$$\mathcal{L}_{\mathcal{S}}(a,b) := \sum_{x\in\mathbf{F}_2^n} (-1)^{\langle b,\mathcal{S}(x)\rangle + \langle a,x\rangle}.$$

The *linearity* ℓ of $\mathcal{S}$ is defined as:

$$\max_{(a,b)\neq(0,0)} |\mathcal{L}_{\mathcal{S}}(a,b)|.$$

Roughly speaking, the linearity measures the maximum (absolute) difference between how many times a (non-trivial) linear approximation takes the value 1 and how many times it takes the value 0. It is therefore twice the difference between 2^{n-1} (for an n-bit S-box) and how many times either value is taken. In particular, if we define the *bias* b of a probability p as $|p - 1/2|$, it means that the bias of any linear approximation of an n-bit S-box of linearity ℓ is upper-bounded by $(\ell/2)/2^n$.

Definition 2.3 (Branch number of an S-box)**.** The *differential branch number* of an S-box $\mathcal{S}$ is:

$$\min_{\{(a,b)\neq(0,0)|\,\delta_{\mathcal{S}}(a,b)\neq 0\}} \mathrm{wt}(a) + \mathrm{wt}(b),$$

where $\mathrm{wt}(x)$ is the Hamming weight of x.
The *linear branch number* of an S-box $\mathcal{S}$ is:

$$\min_{\{(a,b)\neq(0,0)|\,\mathcal{L}_{\mathcal{S}}(a,b)\neq 0\}} \mathrm{wt}(a) + \mathrm{wt}(b).$$

Definition 2.4 (Algebraic normal form)**.** Let $f : \mathbf{F}_2^n \to \mathbf{F}_2$ be an n-bit Boolean function, its *algebraic normal form* (or ANF) is defined as the polynomial $g \in \mathbf{F}_2[x_0, x_1, \ldots x_{n-1}]/ < x_i^2 - x_i >_{i<n}$ such that for all $x \in \mathbf{F}_2^n$, $f(x) = g(x[0], \ldots, x[n-1])$. Similarly, the ANF of an n-bit S-box $\mathcal{S}$ is the sequence of the ANFs of its n constituent Boolean functions projected on the canonical basis of $\mathbf{F}_2^n$.

3 The Littlun S-box construction

3.1 The Lai-Massey structure

Our S-box uses the *Lai-Massey structure*, which was proposed in 1991 for the design of the block cipher IDEA [29]. The structure is similar in its objective to a Feistel or Misty structure (see *e.g.* [14] for definitions) as it allows to construct n-bit functions out of smaller components. It is in particular well-suited to build efficient 8-bit S-boxes from 4-bit S-boxes all the while amplifying the good cryptographic properties of the 4-bit S-boxes. It was already used as such for the design of the second S-box of the Whirlpool hash function [4] (an early version of Whirlpool used a randomly-generated S-box), using five 4-bit S-boxes and for the design of the S-box of the Fox block cipher [27] which uses a 3-round iterated structure. In our construction, we use only one round of the more classical variant of the structure, with only 3 S-boxes, and the linearity of the underlying 4-bit S-box.

The choice of the Lai-Massey structure was mainly motivated by our objective of building an S-box with a branch number of 3[4]. Indeed, it is easy to see that the S-box will have this property for the differential branch number by construction as soon as the 4-bit S-boxes have differential branch number 3, and such S-boxes are well-known (see *e.g.* Serpent [8]). So much cannot be said however for the linear branch number, as no (differential and linear optimal) 4-bit S-box exists with this property[5]. In fact, we are not aware of previous examples of 8-bit S-boxes with this feature either.

Other good properties of the structure are that it yields S-boxes with a circuit depth of two S-boxes and it allows for efficient vector implementations using SIMD instructions. On the downside, it requires the 4-bit S-boxes to be permutations if we want the 8-bit S-box to be one. Canteaut, Duval and Leurent recently showed how the absence of such a restriction for Feistel structures could be used to build compact S-boxes with particularly low differential probability [14]. We should note however that for the applications we have in mind (see Sec. 5), the linearity of the S-box is as important as the differential probability, and the S-box of Canteaut *et al.* is average in that respect (and in particular not better than ours).

3.2 An instantiation: Littlun-1

We now define Littlun-1, a concrete instantiation of the Lai-Massey structure which achieves a differential and linear branch number of 3. Although we have seen that we could guarantee this in the differential case by using a 4-bit S-box of differential branch number 3, this is actually not necessary and we use instead a very compact member of the *class 13* of Ullrich *et al.* [36]. This S-box uses only 4 non-linear and 4 XOR gates, which is minimal for an optimal S-box of this size. This leads to an 8-bit S-box using 12 non-linear and 24 XOR gates. We give the table of the 4-bit S-box "Littlun-S4" in Fig. B.1 and of the complete 8-bit S-box in Fig. B.2, in App. B.1, and conclude this section by a summary of the cryptographic properties of Littlun-1.

Proposition 3.1 (Statistical properties). *The differential uniformity of* Littlun-1 *and of its inverse is 16 and its linearity is 64.*

In essence, Prop. 3.1 means that the probability (taken over all the inputs) of any non-trivial differential relation going through the S-box is upper-bounded by 2^{-4} and the bias of any non-trivial linear approximation is upper-bounded by 2^{-3}.

Proposition 3.2 (Diffusion properties). *The differential and linear branch number of* Littlun-1 *and of its inverse is 3.*

Proposition 3.3 (Algebraic properties). *The maximal degree of (the ANF of)* Littlun-1 *is 5 in 4 of the 8 output bits, 4 in two other and 3 in the remaining two. The maximal degree of its inverse is 5 in 6 of the 8 output bits and 4 in the other 2.*

[4]This will in turn be useful to design a good lightweight round function.
[5]As demonstrated by an exhaustive search we performed on the optimal classes described *e.g.* in [31].

4 Implementation of Littlun-1

4.1 Hardware implementation

We give a circuit (using OR, AND and XOR gates) implementing Littlun-S4 in Fig. B.3 in App. B.2. A hardware implementation of the entire S-box can easily be deduced by plugging this circuit into the one of a Lai-Massey construction. As previously mentioned, Littlun-1 can be implemented with 12 non-linear (OR and AND) gates and 24 XOR gates. With a typical cell library such as the Virtual Silicon standard cell library, OR and AND gates cost 1.33 gate equivalent (GE), and XOR gates 2.67 GE. Thus synthesising the S-box with this library would cost 80 GE.

4.2 Bitsliced software implementation

One of our main objective w.r.t. implementation was to obtain an S-box with an efficient *bitsliced* implementation in software. This is closely related to the simplicity of the circuit of the S-box, though not exactly equivalent. We purposefully chose a 4-bit S-box from the *class 13* of Ullrich *et al.* [36] because of its very efficient bitsliced implementation that requires only 9 instructions on a wide variety of platforms. Such an implementation is given in Fig. 4.1. From this, it is easy to obtain an efficient bitsliced implementation for the whole S-box, as shown in Fig. 4.2. This implementation typically requires 43 instructions and 13 registers.

```
t  = b;    b |= a;    b ^= c; // (B): c ^ (a | b)
c &= t;    c ^= d;            // (C): d ^ (c & b)
d &= b;    d ^= a;            // (D): a ^ (d & B)
a |= c;    a ^= t;            // (A): b ^ (a | C)
```

Figure 4.1: Snippet for a bitsliced C implementation of Littlun-S4 with input and output in registers a, b, c, d (the word holding the most significant bit is taken to be a), using one extra register t.

```
t = a ^ e;
u = b ^ f;
v = c ^ g;
w = d ^ h;
S4(t,u,v,w); // uses one more extra register x
a ^= t;    e ^= t;
b ^= u;    f ^= u;
c ^= v;    g ^= v;
d ^= w;    h ^= w;
S4(a,b,c,d); // reuses t as extra
S4(e,f,g,h); // reuses u as extra
```

Figure 4.2: Snippet for a bitsliced C implementation of Littlun-1, using the code of Fig. 4.1 as subroutine. The input and output registers are a, b, c, d, e, f, g, h (with the most significant bit in word a), the 5 extra registers are t, u, v, w, x.

4.3 Masking

The low number of non-linear gates needed to implement Littlun-1 makes it a suitable choice for applications where counter-measures against side-channel attacks are required. Indeed, it directly implies a lower cost when using Boolean masking schemes (both hardware and software), which represent the primitive to be masked as a circuit [25, 15]. In particular, Littlun-1 is competitive with the S-boxes proposed by Grosso *et al.* [24]: it has the same gate count as the S-box used for Robin and only one more non-linear (and one less XOR) gate than the one used for Fantomas. All 3 S-boxes are comparable in terms of cryptographic properties.

One could alternatively consider that the chief non-linear component to take into account in that context is actually Littlun-S4, the 4-bit S-box underlying Littlun-1, rather than the full S-box. Indeed, any cryptosystem using Littlun-1 (in combination with an arbitrary linear layer) can be re-written as using only Littlun-S4 for its non-linear part. In that respect, the number of non-linear gates to consider for masking would only be 4[6].

[6]One could object that additional factors need to be taken into account, such as for instance the total number of application of the S-box in an execution of the cipher. Yet if we jump a little ahead and consider the block cipher Fly of

The ability to express LITTLUN-1 only in terms of a 4-bit S-box is also convenient when considering threshold implementations (although these chiefly apply to hardware implementations, which are not the focus of this article). For instance, it allows one to benefit from the recent progresses in such protected implementations of small S-boxes [9].

We further discuss the cost of masking a concrete block cipher instance using LITTLUN-1 in Sec. 5.3.

4.4 Inverse S-box

The inverse LITTLUN-1^{-1} of LITTLUN-1 is slightly costlier to implement, because of a more expensive inverse for LITTLUN-S4. As a circuit, the latter requires 5 XOR gates, 4 non-linear (OR and AND) gates and one NOT gate (costing 0.67 GE). The total hardware cost of LITTLUN-1^{-1} is thus 90 GE.

Software bitsliced implementations are also more expensive. We give a snippet for the inverse of LITTLUN-S4 in Fig. A.1 in App. A.

5 An application: the FLY block cipher

In this section, we present the FLY block cipher as an application of the LITTLUN-1 S-box. It is a 64-bit block cipher with 128-bit keys. Thanks to the branch number of the S-box, it is easy to design a round function with good resistance to statistical attacks by combining its bitsliced application with a simple bit permutation. This results in a cipher with a structure similar to PRESENT [10] with a tradeoff: the S-box is bigger (and thus more expensive to implement, in particular in hardware) but the permutation is simpler (and thus cheaper to implement in software). This cipher was designed to be used in the same cases as PRIDE, and its chief implementation target is 8-bit microcontrollers.

5.1 Specifications

We first give the specification of the round function $\mathcal{R}_{\text{FLY}}$ of FLY. It takes a 64-bit block and 64-bit round key as input. Let $x := (x_0||x_1||x_2||x_3||x_4||x_5||x_6||x_7)$, $rk := (rk_0||rk_1||rk_2||rk_3||rk_4||rk_5||rk_6||rk_7)$ be such an input, with x_i, rk_i 8-bit words[7].

Let us first define $f_i(t) := \iota_i(t_0)||\kappa(t_1)||t_2||t_3||t_4||t_5||t_6||t_7$, with $\kappa(x) := x \oplus \texttt{0xFF}$; $\iota_i(x) := x \oplus \mathcal{L}(i)$, $\mathcal{L}(i)$ being a round constant produced as the i^{th} iteration of the LFSR shown in Fig 5.1, initialized with zero. Algebraically speaking, this LFSR implements the mapping $x \mapsto \alpha(x + \alpha^7)$ in $\mathbf{F}_2[\alpha]/\alpha^8 + \alpha^7 + \alpha^3 + \alpha^2 + 1$. Note however that the mapping of elements of this field to the state of the LFSR (or equivalently 8-bit machine words) uses the inverse ordering of the usual convention, *i.e.* the highest coefficient is stored in the LSB. This (as well as the addition of α^7 prior to the multiplication by α) is done to ease software implementations of this round constant generation. An example of such an implementation is given in Fig. C.1.

```
flip = r[0] ^ 1;     r[n] = r[n + 1], n = 6...0;     r[7] = 0;
r[0] = r[0] ^ flip; r[4] = r[4] ^ flip;
r[5] = r[5] ^ flip; r[7] = r[7] ^ flip;
```

Figure 5.1: The LFSR which i^{th} iteration (starting from a zero state) defines the i^{th} round constant. This pseudo-code assumes an 8-bit state r, which entry $r[0]$ maps to the LSB in a machine representation.

We write ARK_i the addition of the i^{th} round key: $\text{ARK}_i(rk_i, x) := f_i(x \oplus rk_i)$; $\text{BLS}(x)$ a bitsliced application of LITTLUN-1 (such as *e.g.* the one shown in Fig 4.2), with x_0 holding the most significant bits of the input to the S-boxes; and ROT the "SHIFTROW" word-wise rotation (with $\circlearrowleft$ denoting bitwise rotation to the left)

$$\text{ROT}(x) := (x_0||x_1 \circlearrowleft 1||x_2 \circlearrowleft 2||x_3 \circlearrowleft 3||x_4 \circlearrowleft 4||x_5 \circlearrowleft 5||x_6 \circlearrowleft 6||x_7 \circlearrowleft 7)$$

which can alternatively be defined at the bit level as the permutation $\pi(i) := (i + 8 \times (i \mod 8)) \mod 64$, applied to a suitable binary representation of $x := b_0 \dots b_{63}$. Then we simply have $\mathcal{R}_{\text{FLY}}(\cdot,\cdot) := \text{ROT} \circ \text{BLS} \circ \text{ARK}$. We give a (slightly simplified) graphical representation of the SPN structure of this round function at the bit level in Fig. 5.2.

Sec. 5, we can see that in terms of the 4-bit S-box, FLY needs $20 \times 8 \times 3 = 480$ calls to LITTLUN-S4, which is comparable to the $31 \times 16 = 496$ of PRESENT (discounting the key-schedule).

[7]The big endian convention is used to convert from x and rk to the x_is and rk_is.

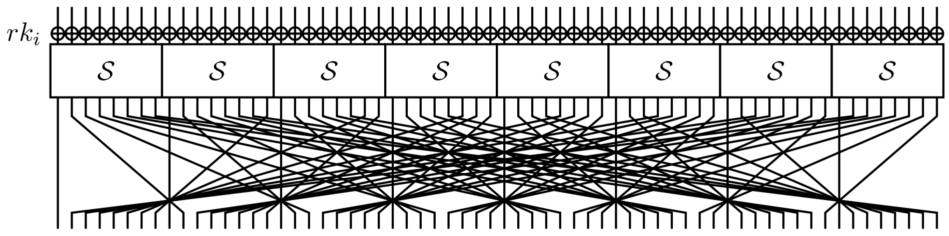

Figure 5.2: The round function of Fly. Bits are numbered left to right from 0 to 63 (w.r.t. the bit permutation). The addition of the round constant is omitted.

We propose two key-schedules, KS1 and KS2, depending on whether resistance to related-key attacks is required (in the case of KS2) or not. In order to distinguish between the two block ciphers, we write Fly for the (default) case where KS1 is used and Fly_{RK} when KS2 is used. We describe KS1 first, which in fact performs only an elementary scheduling: let $k := k_0||k_1$ be the 128-bit master key, and k_0 (resp. k_1) its first (resp. second) half; then the sequence (rk) of round keys of KS1 is the simple alternation of k_0 and $k_0 \oplus k_1$ defined as $rk_i := k_0 \oplus i \times k_1$[8]. A round constant is also added in Ark through the function f_i to prevent self-similarity attacks.

For Fly to be resistant to related-key attacks, we use the same approach as Noekeon [22] to define KS2 as follows. Let us denote by $\text{Fly}(0,\cdot)/12$ twelve applications of the round function of Fly with the all-zero 128-bit key and define $k' := k'_0||k'_1 = \text{Fly}(0,k_0)/12||\text{Fly}(0,k_1)/12$. Then KS2 is defined through Fly_{RK} as $\text{Fly}_{RK}(k,\cdot) := \text{Fly}(k',\cdot)$.

The round function of Fly is applied 20 times, the same as Pride. The entire cipher can thus finally be defined as $\text{Fly}(k,\cdot) := \text{Ark}(rk_{20},\cdot) \circ \mathcal{R}_{\text{Fly}}(rk_{19},\cdot) \circ \ldots \circ \mathcal{R}_{\text{Fly}}(rk_1,\cdot) \circ \mathcal{R}_{\text{Fly}}(rk_0,\cdot)$.

Design rationale

The core of Fly is the Littlun-1 S-box, which was designed to have a branch number of 3. This allows to achieve a good diffusion when combining the S-box application with a simple bit permutation. The latter was chosen so that all eight bits at the output of an S-box go to one different S-box each (similarly, all input bits come from a different S-box). Unlike in Present, this permutation also has cycles of different lengths (discounting fixed points), namely 2 (on 8 values), 4 (on 16) and 8 (on 32). This might reduce the impact of (linear and differential) characteristic clustering. The round constants break the self-similarity and self-symmetry of the round function (through ι) and the self-symmetry of the S-box (through κ)[9].

The two components of the round function can be efficiently implemented on an 8-bit architecture through a bitsliced application of the S-box and word rotations respectively (cf. Sec 5.3).

The two key-schedules were designed according to different possible scenarios. Most applications do not require resistance to related-key attacks and a simple alternating key-schedule is enough in that case. We chose not to use an FX construction as in Pride as we did not consider the slight gain in efficiency it offers to be worth the generic security loss it implies. In the spirit of Noekeon, we propose a second key-schedule to offer resistance to related-key attacks that consists in "scrambling" the master key with a permutation of good differential uniformity before it is used as in the first key-schedule.

5.2 Preliminary security analysis

We now analyse the security of Fly against various types of attacks. Considering the similarity of the design with Present and the published analysis on this cipher, the most efficient attacks on Fly are likely to be (variants of) classical statistical (differential and linear) attacks, which we analyse first (in the single-key setting). We then give an overview of the resistance against other attack techniques.

We can use the branch number of Littlun-1 together with the properties of the bit permutation Rot to easily derive a lower bound on the number of (differentially and linearly) active S-boxes. Indeed,

[8]By writing $i \times k_1$, we mean the all-zero key for even values of i and k_1 otherwise.

[9]This symmetry is actually already broken by ι and (most of the times) by the round keys, but using an extra constant allows for a simple and clean argument at a negligible cost.

as the branch number is 3, we are guaranteed to have at least 6 active S-boxes every 4 rounds of any non-trivial differential or linear characteristic. This is a consequence of the following proposition:

Proposition 5.1. *There is no 3-round characteristic on* FLY *activating 1, then n, then 1 S-box, for any value of n.*

Proof. In a round following one round with a single active S-box, all n active S-boxes are active in a single bit of their input, and consequently each of their output activates at least 2 S-boxes. □

The block size of FLY being 64 bits, we want any differential characteristic to have a probability $p \approx 2^{-64}$ when averaged over the key and message space. Similarly, we want any linear characteristic to have an average bias $b \approx 2^{-32}$. From the Prop. 3.1 of LITTLUN-1, by multiplying the differential probabilities and applying the piling-up lemma respectively, this means that we want a differential (respectively linear) characteristic to have *at least* 16 active S-boxes. This happens at the latest after 12 rounds, for which at least 18 S-boxes are guaranteed to be active. Even by discounting the additional 2 S-boxes and assuming that a distinguisher can be found for this amount of rounds, this gives a very comfortable margin of 8 rounds, which we estimate to be much beyond the ability of an attacker to convert the distinguisher into, say, key-recovery (in particular, this is twice the number of rounds needed for full diffusion). This also leaves some margin to ensure that even in the case where FLY would exhibit a strong differential or linear hull it would be unlikely for an attacker to be able to mount a meaningful attack. For instance, after 16 rounds, an attacker would need about 2^{32} "optimal" contributing characteristics to obtain a distinguisher with non-trivial probability, and would still be facing 4 rounds to mount an attack.

Thus, we conjecture that FLY with 20 rounds offers good resistance to statistical attacks.

Brief comparison with PRESENT. The best attacks to date on PRESENT are based on multidimensional linear cryptanalysis [16, 11]. These attacks exploit the presence of linear characteristics that constantly activate only one S-box per round (*i.e.* "single-bit characteristics"). As there are no such characteristics in FLY, we believe that these attacks would be less effective on the latter. Similarly, some other good attacks on PRESENT exploit the fact that half of the bits of some groups of S-boxes remain in the same group [18], and there is no such property for FLY.

Algebraic attacks. We would like to estimate how many rounds of FLY are necessary for the degree of the cipher to reach the maximum of 63, as a lower degree could be exploited in "algebraic" attacks. Computing the exact degree of an iterated function is a difficult problem in general, but we should at least compute the upper bound of Boura, Canteaut and De Cannière to estimate how quickly the degree increases [13]. In our case, this bound states that $\deg(G \circ F) \leq n - (n - \deg(G))/(n_0 - 2)$, where n is the block size and n_0 the size of the S-box. Combining this bound with the fact that the degree of the S-box is 5 (and thus that $\deg(\text{BLS} \circ F) \leq 5 \cdot \deg(F)$), we can see that 5 rounds of FLY are necessary to reach a full degree. If we assume this bound to be an equality, any (algebraic) distinguisher on more than about twice this number of rounds is unlikely to exist. Even when (necessarily) relaxing this latter (strong) assumption, 20 rounds seem to be well enough to make FLY resistant to algebraic attacks.

Meet-in-the-middle attacks. We analysed how many rounds are necessary to ensure that every bit of the (intermediate) ciphertext depends on every bit of the key, as a basic way to estimate the resistance of FLY to meet-in-the-middle (MitM) attacks, which typically exploit the opposite effect. We did this by performing random trials with 2^{20} pairs of random keys and random plaintexts and found that this happens after at most 5 rounds. Any MitM attack on more than about twice this number of rounds is unlikely to exist, and we therefore conjecture that FLY is resistant to such attacks[10].

Invariant subspace attacks. Invariant subspace attacks exploit the propensity of a round function to map inputs from a certain (non-trivial) affine coset to another, when in addition a trivial key-schedule and sparse round constants are used [30]. As the two latter points appear in FLY, we analysed its round function in order to see if it could also meet the first, critical condition. We ran the automated search tool provided by the authors of [30][11] for about 2^{36} iterations without finding any invariant subspace;

[10] Which key-schedule (KS1 or KS2) is used is irrelevant, as KS2 is equivalent to using KS1 with a different "effective" master key, which a MitM attacker can recover in the exact same way as the "true" master key produced by KS1.

[11] Available at http://invariant-space.gforge.inria.fr/.

this shows that with good probability, no such subspace of dimension greater than $64 - 36 = 28$ exists.

Integral attacks, impossible differentials, zero correlation. We did not analyse in detail the security of Fly against integral attacks, nor against impossible differentials and zero correlation attacks. Indeed, none of these techniques seem to be able to attack a significant number of rounds of bit-oriented ciphers such as Present (see *e.g.* [37, 21, 12], where the obtained distinguishers reach significantly less rounds than the best statistical ones) or Fly and we do not consider them to be a threat for our cipher.

Related-key attacks. We now study the resistance of Fly against (XOR-induced, differential) related-key attacks. Fly equipped with the simple key-alternating key-schedule KS1 offers (nearly) no resistance to related-key attacks. With KS2, however, an attacker is unable to control the differences between two distinct effective master keys $k_0'||k_1'$ and $\widetilde{k_0'||k_1'}$ with a probability much better than $2^{-2 \cdot 64}$, as each difference pair $(k_0, \widetilde{k_0})$ and $(k_1, \widetilde{k_1})$ goes through a permutation with maximum expected differential probability not significantly above 2^{-64}. Furthermore, unlike single-key differential attacks, which introduce differences on the plaintext, we do not expect an attacker to easily be able to force a change of (related) keys if their effective master keys fail to verify a difference relation. Thus, even if a differential on KS2 with probability p higher than 2^{-128} were found, it would only lead to a related-key attack on a weak-key class (of size $\approx p/2^{-128}$) or to an attack requiring a huge amount of keys. Putting everything together, we believe Fly_{RK} to be resistant to XOR-induced related-key attacks.

5.3 Implementation

5.3.1 Microcontrollers implementations

The S-box application can take advantage of the bitsliced expression of Littlun-1 from Sec. 4, which can easily be implemented with instructions available on the cheapest ATtiny chips [2]. It is even possible to save 2 instructions from the 43 quoted in Sec. 4 on higher-end architectures such as the ATmega family [3] by using word-wise 16-bit `movw` instructions, resulting in the implementation given in Fig. C.3 of App. C. A straightforward implementation of the inverse S-box application requires 59 instructions —a significant overhead of 44 %. However, as a lightweight cipher is precisely used in cases where the available resources are limited, we would mostly expect it to be used in a mode of operation that only uses encryption, such as *e.g.* CTR (for encryption only) or CLOC [26] (for authenticated-encryption). Hence we do not believe that a slower inverse is a significant drawback.

Even though the AVR instruction set does not include rotations by an arbitrary constant, the permutation Rot can still be compactly implemented with only 11 instructions, as shown in Fig. C.2 of App. C.

The entire substitution and permutation layers of Fly can therefore be implemented with only 52 instructions on ATmega (54 on ATtiny), which is 4 less than the 56 of Pride [1], while at the same time having at least 1.5 times more *equivalent* active S-boxes every 4 rounds[12].

On-the-fly computation of one round-key of the key-schedule KS1 can be done in 8 instructions. The complete key expansion and round constant addition can be done in 24 instructions as shown in Fig. C.1.

The complete round function of Fly including the key-schedule can thus be implemented in 76 instructions, which is eight more than Pride. Note however that the conjectured security margin of Fly is much bigger, and unlike Pride, its resistance to generic attacks does not decrease with the amount of data available to the adversary[13]. Furthermore, as the key schedules of Fly and Pride are rather similar, one could consider using the round function of Fly with the key schedule of Pride. This would result in a cipher slightly more efficient than both, with the same profile as the latter; we do not expect such a swap to introduce any specific weakness.

[12]There are at least 4 active 4-bit S-boxes of maximum differential probability 2^{-2} and best linear correlation 2^{-1} every two rounds of Pride and there are at least 3 active 8-bit S-boxes of maximum differential probability 2^{-4} and best linear correlation 2^{-2} every two rounds of Fly.

[13]Pride uses an FX construction, where one half of its 128-bit key is only used for pre- and post-whitening. This leads to a simple way to merge on-the-fly round-key generation with the round function, but significantly degrades the security of the cipher to generic attacks from 128 bits to $128 - \log(D)$, with D the amount of data available to an attacker [28], while also leading to more efficient time-memory-data trade-offs [23].

Cipher	Unmasked	Order 2	Order 4	Order 7	Order 11
Fly (8)	1909	10741	27253	66421	145525
Simon64-128 (8)	3400	14056	30344	65336	131704
Simon64-128 (32)	1012	3926	8240	17336	34364
Pride (8)	1374	22922	60550	150592	333368
Speck64-128 (32)	486	48198	132652	337843	757983

Table 5-1: Count of operations needed to encrypt one block with each cipher, masked at various orders.

5.3.2 Masked implementations

We have just considered the good performance of Fly on AVR processors. However, on such platforms, discounting the overhead of protection against side-channel attacks may be misleading. Indeed, these devices are rather prone to leakage, and it might not be entirely reasonable to deploy unprotected cryptosystems on them [35]. Consequently, we consider here the cost of masked implementations of Fly, and compare it to Pride and the "NSA ciphers" Simon and Speck [6] in the same setting. All the masked implementations have been generated automatically by using the compiler of Barthe *et al.* [5]. This compiler takes a `C` implementation of a cipher as an input and generates `C` code for a corresponding masked implementation. This code is then instrumented to count the number of basic instructions (*e.g.* logical and arithmetic) executed in the encryption of one block.

We report the cost (in terms of number of instruction) for the studied ciphers and configurations in Tbl. 5-1. The first column gives the name of the cipher, followed by the basic word-size used in the implementation (for 8-bit microcontrollers, the most relevant value for this number is understandingly 8). The next columns give the number of instructions (over the same word-size as the cipher) taken to encrypt one block with an unmasked implementation, and then masked implementations at various orders (an order-t implementation ensures security in the t-probing model [25]).

From these results, a first important remark we can make is that neither Pride nor Speck seem to be well suited to masked implementations. This is due to their conjoined use of bitwise operations and integer modular addition (80 8-bit additions for Pride (in its key-schedule) and 54 32-bit additions for Speck64-128). Masking bitwise operations can be done relatively efficiently by using a Boolean sharing scheme but it is costly to do so with an additive scheme, while the converse holds for modular additions. In practice, the compiler of Barthe *et al.* uses the algorithm of Coron *et al.* from CHES 2014 to mask modular additions with a Boolean scheme [20][14].

On the contrary, both Fly and Simon are quite efficient to mask with a Boolean scheme. Note that we implemented Simon64-128 in two ways: one using 8-bit words, suitable for 8-bit microcontrollers (and thus directly comparable with the intended use of Fly), and one using 32-bit words (which is more straightforward in a way as all instances of Simon on $2n$-bit blocks can be expressed naturally with n-bit arithmetic).

On 8-bit platforms, our unmasked implementation of Fly is more efficient than Simon64-128. This advantage is maintained up to a small number of shares, but starting from 8 (*i.e.* an implementation secure at order 7), Simon becomes more efficient. This behaviour can be explained by the breakdown of the cost for the two ciphers: although an unmasked Simon64-128 is costlier than Fly overall, our masked implementation of the former uses only 176 *refresh* and *and* operations, while Fly needs 240. As the cost of these operations is quadratic in the number of shares, implementing Simon at high order becomes cheaper than implementing Fly, but the latter starts with a significant initial advantage.

In conclusion both Fly and Simon are well suited to masked implementations on 8-bit microcontrollers. While the latter is the most efficient of the two for 7+-order implementations, Fly is cheaper in the (in our opinion more relevant) case of low-order ones.

[14]If we were to restrict ourselves to first-order masking, it would be possible to use the more efficient algorithm of Coron *et al.* from FSE 2015 [19].

References

[1] Albrecht, M.R., Driessen, B., Kavun, E.B., Leander, G., Paar, C., Yalçin, T.: Block Ciphers - Focus on the Linear Layer (feat. PRIDE). In: Garay, J.A., Gennaro, R. (eds.) CRYPTO 2014. Lecture Notes in Computer Science, vol. 8616, pp. 57–76. Springer (2014)

[2] Atmel: 8-bit AVR Microcontroller with 1K Byte Flash (Rev 1006FS-AVR-06/07)

[3] Atmel: 8-bit AVR Microcontroller with 8KBytes In-System Programmable Flash (Rev 2486AA-AVR-02/2013)

[4] Barreto, P.S.L.M., Rijmen, V.: The Whirlpool Hashing Function (May 2003)

[5] Barthe, G., Belaïd, S., Dupressoir, F., Fouque, P., Grégoire, B.: Compositional Verification of Higher-Order Masking: Application to a Verifying Masking Compiler. IACR Cryptology ePrint Archive 2015, 506 (2015)

[6] Beaulieu, R., Shors, D., Smith, J., Treatman-Clark, S., Weeks, B., Wingers, L.: The SIMON and SPECK Families of Lightweight Block Ciphers. IACR Cryptology ePrint Archive 2013, 404 (2013)

[7] Beaulieu, R., Shors, D., Smith, J., Treatman-Clark, S., Weeks, B., Wingers, L.: SIMON and SPECK: Block Ciphers for the Internet of Things. IACR Cryptology ePrint Archive 2015, 585 (2015), presented at the NIST Lightweight Cryptography Workshop 2015

[8] Biham, E., Anderson, R.J., Knudsen, L.R.: Serpent: A New Block Cipher Proposal. In: Vaudenay, S. (ed.) FSE '98. Lecture Notes in Computer Science, vol. 1372, pp. 222–238. Springer (1998)

[9] Bilgin, B., Nikova, S., Nikov, V., Rijmen, V., Tokareva, N., Vitkup, V.: Threshold implementations of small S-boxes. Cryptography and Communications 7(1), 3–33 (2015)

[10] 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. Lecture Notes in Computer Science, vol. 4727, pp. 450–466. Springer (2007)

[11] Bogdanov, A., Tischhauser, E., Vejre, P.S.: Multivariate Linear Cryptanalysis: The Past and Future of PRESENT. IACR Cryptology ePrint Archive 2016, 667 (2016)

[12] Boura, C., Canteaut, A.: Another View of the Division Property. In: Robshaw, M., Katz, J. (eds.) CRYPTO 2016. Lecture Notes in Computer Science, vol. 9814, pp. 654–682. Springer (2016)

[13] Boura, C., Canteaut, A., De Cannière, C.: Higher-Order Differential Properties of Keccak and *Luffa*. In: Joux, A. (ed.) FSE 2011. Lecture Notes in Computer Science, vol. 6733, pp. 252–269. Springer (2011)

[14] Canteaut, A., Duval, S., Leurent, G.: Construction of Lightweight S-Boxes using Feistel and MISTY structures (Full Version). IACR Cryptology ePrint Archive 2015, 711 (2015)

[15] Carlet, C., Goubin, L., Prouff, E., Quisquater, M., Rivain, M.: Higher-Order Masking Schemes for S-Boxes. In: Canteaut, A. (ed.) FSE 2012. Lecture Notes in Computer Science, vol. 7549, pp. 366–384. Springer (2012)

[16] Cho, J.Y.: Linear Cryptanalysis of Reduced-Round PRESENT. In: Pieprzyk, J. (ed.) CT-RSA 2010. Lecture Notes in Computer Science, vol. 5985, pp. 302–317. Springer (2010)

[17] Cid, C., Rechberger, C. (eds.): Fast Software Encryption — FSE 2014, Lecture Notes in Computer Science, vol. 8540. Springer (2015)

[18] Collard, B., Standaert, F.: A Statistical Saturation Attack against the Block Cipher PRESENT. In: Fischlin, M. (ed.) CT-RSA 2009. Lecture Notes in Computer Science, vol. 5473, pp. 195–210. Springer (2009)

[19] Coron, J., Großschädl, J., Tibouchi, M., Vadnala, P.K.: Conversion from Arithmetic to Boolean Masking with Logarithmic Complexity. In: Leander, G. (ed.) FSE 2015. Lecture Notes in Computer Science, vol. 9054, pp. 130–149. Springer (2015)

[20] Coron, J., Großschädl, J., Vadnala, P.K.: Secure Conversion between Boolean and Arithmetic Masking of Any Order. In: Batina, L., Robshaw, M. (eds.) CHES 2014. Lecture Notes in Computer Science, vol. 8731, pp. 188–205. Springer (2014)

[21] Cui, T., Jia, K., Fu, K., Chen, S., Wang, M.: New Automatic Search Tool for Impossible Differentials and Zero-Correlation Linear Approximations. IACR Cryptology ePrint Archive 2016, 689 (2016)

[22] Daemen, J., Peeters, M., Assche, G.V., Rijmen, V.: The Noekeon Block Cipher. Nessie Proposal (October 2000)

[23] Dinur, I.: Cryptanalytic Time-Memory-Data Tradeoffs for FX-Constructions with Applications to PRINCE and PRIDE. In: Oswald and Fischlin [33], pp. 231–253

[24] Grosso, V., Leurent, G., Standaert, F., Varici, K.: LS-Designs: Bitslice Encryption for Efficient Masked Software Implementations. In: Cid and Rechberger [17], pp. 18–37

[25] Ishai, Y., Sahai, A., Wagner, D.: Private Circuits: Securing Hardware against Probing Attacks. In: Boneh, D. (ed.) CRYPTO 2003. Lecture Notes in Computer Science, vol. 2729, pp. 463–481. Springer (2003)

[26] Iwata, T., Minematsu, K., Guo, J., Morioka, S.: CLOC: Authenticated Encryption for Short Input. In: Cid and Rechberger [17], pp. 149–167

[27] Junod, P., Vaudenay, S.: FOX : A New Family of Block Ciphers. In: Handschuh, H., Hasan, M.A. (eds.) SAC 2004. Lecture Notes in Computer Science, vol. 3357, pp. 114–129. Springer (2004)

[28] Kilian, J., Rogaway, P.: How to Protect DES Against Exhaustive Key Search (an Analysis of DESX). J. Cryptology 14(1), 17–35 (2001)

[29] Lai, X., Massey, J.L.: Markov Ciphers and Differential Cryptanalysis. In: Davies, D.W. (ed.) EUROCRYPT '91. Lecture Notes in Computer Science, vol. 547, pp. 17–38. Springer (1991)

[30] Leander, G., Minaud, B., Rønjom, S.: A Generic Approach to Invariant Subspace Attacks: Cryptanalysis of Robin, iSCREAM and Zorro. In: Oswald and Fischlin [33], pp. 254–283

[31] Leander, G., Poschmann, A.: On the Classification of 4 Bit S-Boxes. In: Carlet, C., Sunar, B. (eds.) WAIFI 2007. Lecture Notes in Computer Science, vol. 4547, pp. 159–176. Springer (2007)

[32] National Institute of Standards and Technology: FIPS 197: Advanced Encryption Standard (AES) (November 2001)

[33] Oswald, E., Fischlin, M. (eds.): Advances in Cryptology — EUROCRYPT 2015, Lecture Notes in Computer Science, vol. 9056. Springer (2015)

[34] Poschmann, A.: Lightweight Cryptography. Ph.D. thesis, Ruhr-Universität Bochum (2009)

[35] Strobel, D., Oswald, D., Richter, B., Schellenberg, F., Paar, C.: Microcontrollers as (In)Security Devices for Pervasive Computing Applications. Proceedings of the IEEE 102(8), 1157–1173 (2014)

[36] Ullrich, M., De Cannière, C., Indesteege, S., Özgül Küçük, Mouha, N., Preneel, B.: Finding Optimal Bitsliced Implementations of 4×4-bit S-boxes. In: SKEW 2011 (2011)

[37] Z'aba, M.R., Raddum, H., Henricksen, M., Dawson, E.: Bit-Pattern Based Integral Attack. In: Nyberg, K. (ed.) FSE 2008. Lecture Notes in Computer Science, vol. 5086, pp. 363–381. Springer (2008)

[38] Zhang, W., Bao, Z., Lin, D., Rijmen, V., Yang, B., Verbauwhede, I.: RECTANGLE: A Bit-slice Ultra-Lightweight Block Cipher Suitable for Multiple Platforms. IACR Cryptology ePrint Archive 2014, 84 (2014)

[39] Zhao, J., Wang, X., Wang, M., Dong, X.: Differential Analysis on Block Cipher PRIDE. IACR Cryptology ePrint Archive 2014, 525 (2014)

A Software implementation of the inverse of Littlun-S4

Implementing the inverse of Littlun-S4 in a straightforward way requires 11 instructions and 5 registers. The complete inverse of Littlun can be implemented with 49 instructions and 13 registers in a straightforward adaptation of Fig. 4.2[15].

```
t   = c;     c &= b;     c ^= d; // (A): d ^ (b & c)
d  |= t;     d ^= a;             // (B): a ^ (c | d)
a  &= c;     a ^= b;     a ^= d; // (C): b ^ B ^ (a & A)
b  = ~b;     b &= d;     b ^= t; // (D): c ^ (~b & B)
```

Figure A.1: Snippet for a bitsliced C implementation of the inverse of Littlun-S4 with inputs in registers a, b, c, d (the word holding the most significant bit is taken to be a), using one extra register t. The output is in c, d, a, b.

B Examples of implementation of Littlun-1

B.1 Tables for Littlun-1 and Littlun-S4

```
uint8_t littlun1_s4[16] =
                  {0x0, 0xa, 0x4, 0xf, 0xc, 0x7, 0x2, 0x8,
                   0xd, 0xe, 0x9, 0xb, 0x5, 0x6, 0x3, 0x1};
```

Figure B.1: The 4-bit S-box Littlun-S4 used in Littlun-1 as a C array

```
uint8_t littlun1[256] =
        {0x00, 0x9b, 0xc2, 0x15, 0x5d, 0x84, 0x4c, 0xd1,
         0x67, 0x38, 0xef, 0xb0, 0x7e, 0x2b, 0xf6, 0xa3,
         0xb9, 0xaa, 0x36, 0x78, 0x2f, 0x6e, 0xe3, 0xf7,
         0x12, 0x5c, 0x9a, 0xd4, 0x89, 0xcd, 0x01, 0x45,
         0x2c, 0x63, 0x44, 0xde, 0x02, 0x96, 0x39, 0x70,
         0xba, 0xe4, 0x18, 0x57, 0xa1, 0xf5, 0x8b, 0xce,
         0x51, 0x87, 0xed, 0xff, 0xb5, 0xa8, 0xca, 0x1b,
         0xdf, 0x90, 0x6c, 0x32, 0x46, 0x03, 0x7d, 0x29,
         0xd5, 0xf2, 0x20, 0x5b, 0xcc, 0x31, 0x04, 0xbd,
         0xa6, 0x41, 0x8e, 0x79, 0xea, 0x9f, 0x68, 0x1c,
         0x48, 0xe6, 0x69, 0x8a, 0x13, 0x77, 0x9e, 0xaf,
         0xf3, 0x05, 0xcb, 0x2d, 0xb4, 0xd0, 0x37, 0x52,
         0xc4, 0x3e, 0x93, 0xac, 0x40, 0xe9, 0x22, 0x56,
         0x7b, 0x8d, 0xf1, 0x06, 0x17, 0x62, 0xbf, 0xda,
         0x1d, 0x7f, 0x07, 0xb1, 0xdb, 0xfa, 0x65, 0x88,
         0x2e, 0xc9, 0xa5, 0x43, 0x58, 0x3c, 0xe0, 0x94,
         0x76, 0x21, 0xab, 0xfd, 0x6a, 0x3f, 0xb7, 0xe2,
         0xdd, 0x4f, 0x53, 0x8c, 0xc0, 0x19, 0x95, 0x08,
         0x83, 0xc5, 0x4e, 0x09, 0x14, 0x50, 0xd8, 0x9c,
         0xf4, 0xee, 0x27, 0x61, 0x3b, 0x7a, 0xa2, 0xb6,
         0xfe, 0xa9, 0x81, 0xc6, 0xe8, 0xbc, 0x1f, 0x5a,
         0x35, 0x72, 0x99, 0x0a, 0xd3, 0x47, 0x24, 0x6d,
         0x0b, 0x4d, 0x75, 0x23, 0x97, 0xd2, 0x60, 0x34,
         0xc8, 0x16, 0xa0, 0xbb, 0xfc, 0xe1, 0x5e, 0x8f,
         0xe7, 0x98, 0x1a, 0x64, 0xae, 0x4b, 0x71, 0x85,
         0x0c, 0xb3, 0x3d, 0xcf, 0x55, 0x28, 0xd9, 0xf0,
         0xb2, 0xdc, 0x5f, 0x30, 0xf9, 0x0d, 0x26, 0xc3,
         0x91, 0xa7, 0x74, 0x1e, 0x82, 0x66, 0x4a, 0xeb,
         0x6f, 0x10, 0xb8, 0xd7, 0x86, 0x73, 0xfb, 0x0e,
         0x59, 0x2a, 0x42, 0xe5, 0x9d, 0xa4, 0x33, 0xc7,
         0x3a, 0x54, 0xec, 0x92, 0xc1, 0x25, 0xad, 0x49,
         0x80, 0x6b, 0xd6, 0xf8, 0x0f, 0xbe, 0x7c, 0x11};
```

Figure B.2: The Littlun-1 S-box as a C array

B.2 Hardware circuit for Littlun-S4

[15]Because the output registers form a non-trivial permutation of the input ones, additional instructions may also be needed in the cases where this cannot be dealt with implicitly.

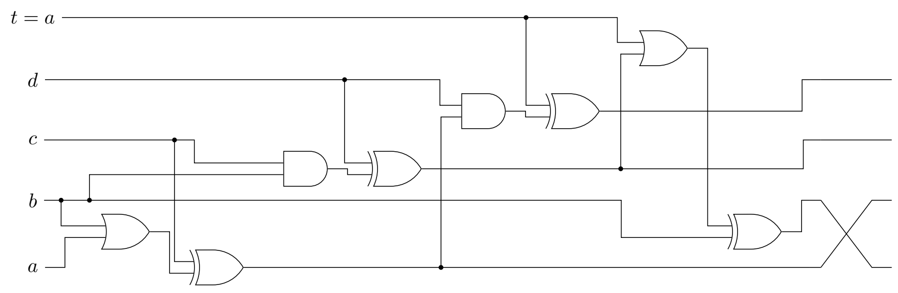

Figure B.3: A circuit implementation of LITTLUN-S4. The symbols , and represent the AND, OR and XOR gates respectively.

C AVR implementation of the FLY round function

We give pseudo AVR assembly code for the S-box layer, the permutation and on-the-fly computation of the key-schedule of FLY. All the state, key and temporary variables fit in the 32 registers of an ATtiny or ATmega.

```
; key input in k0,...,k15
; cipher state in s0,...,s7
; round constant in c0
; temporary register in t0

; add the current round key & round constant to the state
eor s0, k0
eor s1, k1
eor s2, k2
eor s3, k3
eor s4, k4
eor s5, k5
eor s6, k6
eor s7, k7

eor s0, c0
eor s1, 255

; update k0,...,k7 to the next round key
eor k0, k8
eor k1, k9
eor k2, k10
eor k3, k11
eor k4, k12
eor k5, k13
eor k6, k14
eor k7, k15

; update c0 to the next round constant
mov  t0, c0
andi t0, 1
dec  t0
andi t0, 177
lsr  c0
eor  c0, t0
```

Figure C.1: Key addition, and the KS1 key-schedule on ATmega/ATtiny, using 24 instructions.

```
; input/output in s0,...,s7
rol s1
rol s2
rol s2
swap s3
ror s3
swap s4
swap s5
rol s5
ror s6
ror s6
ror s7
```

Figure C.2: The Rot permutation on ATmega/ATtiny, using 11 instructions.

```
; input/output in s0,...,s7 (with MSBs in s0)
; temporary values held in t0,...,t4
; top XOR
movw t0, s0 ; moves t1:s0 <- s1:s0
movw t2, s2
eor t0, s4
eor t1, s5
eor t2, s6
eor t3, s7
; middle S-box
mov t4, t1
or  t1, t0
eor t1, t2
and t2, t4
eor t2, t3
and t3, t1
eor t3, t0
or  t0, t2
eor t0, t4
; bottom XOR
eor s0, t0
eor s1, t1
eor s2, t2
eor s3, t3
eor s4, t0
eor s5, t1
eor s6, t2
eor s7, t3
; bottom S-boxes
mov t0, s1
or  s1, s0
eor s1, s2
and s2, t0
eor s2, s3
and s3, s1
eor s3, s0
or  s0, s2
eor s0, t0

mov t0, s5
or  s5, s4
eor s5, s6
and s6, t0
eor s6, s7
and s7, s5
eor s7, s4
or  s4, s6
eor s4, t0
```

Figure C.3: The Littlun-1 S-box on ATmega, using 41 instructions.

D Hardware implementation of Fly

Fly was not designed to be particularly efficient in hardware, and there are clearly better alternatives in that setting. Thus we did not implement Fly in hardware, but it might be informative to very roughly estimate the cost (in GE) of such an implementation. This can be done by looking at the cost of Present, given the similarity of their structures. A round-based ASIC implementation of Present-128 can be done for 1884 GE [34], of which 27×16 are dedicated to implementing the 16 S-boxes. If we make the assumption that the key-schedule of Fly does not use significantly more area than the one of Present-128, we can estimate that a similar round-based implementation of Fly would cost in the area of $1884 - 27 \times 16 + 80 \times 8 = 2092$ GE, meaning that the overhead is about 11 %.

E Test vectors for Fly

All numbers are given in big endian (*i.e.*, those are arrays of bytes, with the byte of lowest address on the left).

```
k0: 0x0000000000000000 k1: 0x0000000000000000
p : 0x0000000000000000
FLY(k0||k1,p)          : 0x40A942D3FB302724

k0: 0x0001020304050607 k1: 0x08090A0B0C0D0E0F
p : 0xF7E6D5C4B3A29180
FLY(k0||k1,p)          : 0x0D3FE2BF9650AE34

k0: 0x0000000000000000 k1: 0x0000000000000000
p : 0x0000000000000000
FLY(0,k0)/12           : 0x228F5762975E5B43
FLY(0,k1)/12           : 0x228F5762975E5B43
FLY_RK(k0||k1,p)       : 0x7C5B37DC56F4829A

k0: 0x0001020304050607 k1: 0x08090A0B0C0D0E0F
p : 0xF7E6D5C4B3A29180
FLY(0,k0)/12           : 0x68F5FC8290A95219
FLY(0,k1)/12           : 0x58F242AC38C00E6B
FLY_RK(k0||k1,p)       : 0x8EE2EA8B0A63DE6D
```