IACR Transactions on Symmetric Cryptology
ISSN 2519-173X, Vol. 2020, No. S1, pp. 31–59. DOI:10.13154/tosc.v2020.iS1.31-59

Pyjamask: Block Cipher and Authenticated Encryption with Highly Efficient Masked Implementation

Dahmun Goudarzi[1], Jérémy Jean[2], Stefan Kölbl[3], Thomas Peyrin[4], Matthieu Rivain[5], Yu Sasaki[6] and Siang Meng Sim[4]

[1] PQShield, Oxford, United Kingdom
Dahmun.Goudarzi@pqshield.com

[2] Agence nationale de la sécurité des systèmes d'information (ANSSI), Paris, France
Jeremy.Jean@ssi.gouv.fr

[3] Independent[†]
kste@mailbox.org

[4] School of Physical and Mathematical Sciences
Nanyang Technological University, Singapore
Thomas.Peyrin@ntu.edu.sg, crypto.s.m.sim@gmail.com

[5] CryptoExperts, Paris, France
Matthieu.Rivain@cryptoexperts.com

[6] NTT Secure Platform Laboratories, Tokyo, Japan
Sasaki.Yu@lab.ntt.co.jp

Abstract. This paper introduces `Pyjamask`, a new block cipher family and authenticated encryption proposal submitted to the NIST lightweight cryptography standardization process. `Pyjamask` targets side-channel resistance as one of its main goal. More precisely, it strongly minimizes the number of nonlinear gates used in its internal primitive in order to allow efficient masked implementations, especially for high-order masking in software. Compared to other block ciphers, our proposal has thus among the smallest number of binary AND computations per input bit at the time of writing. Even though `Pyjamask` minimizes such an important criterion, it remains rather lightweight and efficient, thanks to a general bitslice construction that enables to computation of all nonlinear gates in parallel. For authenticated encryption, we adopt the provably secure AEAD mode `OCB` which has been extensively studied and has the benefit to offer full parallelization. Of course, other block cipher-based modes can be considered as well if other performance profiles are to be targeted.

The paper first gives the specification of the `Pyjamask` block cipher and the associated AEAD proposal. We also provide a detailed design rationale for the block cipher which is guided by our aim of software efficiency in the presence of high-order masking. The security of the design is analyzed against most commonly known cryptanalysis techniques. We finally describe efficient (masked) implementations in software and provide implementation results with aggressive performances for masking of very high orders (up to 128). We also provide a rough estimation of the hardware performances which remain much better than those of an AES round-based implementation.

[†]Author is now working at Google.

Received: 2019-12-10, Accepted: 2020-02-15, Published: 2020-06-22

Keywords: Block Cipher · Authenticated Encryption · Fast Software Encryption · High-Order Masking · Lightweight Cryptography

1 Introduction

Confidentiality and authenticity of data are the two crucial security properties that one must ensure when communicating over an insecure channel. These properties have historically been realized separately, for example using a secure block cipher and an encryption operating mode to provide confidentiality and using a hash function or again a block cipher in a proper MAC algorithm to provide authenticity. However, this means that it was left to the practitioners how to combine both tools to secure the communication channel, leading to major security breaches [Kra01, AP13]. It became clear that a primitive providing both security notions at the same time would be interesting, not only from a security point of view, but also from an efficiency perspective as this would potentially provide an opportunity for the designers to save some computations (in comparison to two independent computations). An Authenticated Encryption (AE) scheme ensures jointly authenticity and confidentiality of data, and this concept was generalized to Authenticated Encryption with Associated Data (AEAD) which would allow some data to not be encrypted but only authenticated [Rog02]. Many provable AEAD modes and even ad-hoc designs, especially through the CAESAR competition [CAE], have been proposed since then.

Due to the increasing widespread of pervasive computing devices, another hot topic in cryptography during this last decade was lightweight cryptography, whose goal is to provide cryptographic primitives for constrained devices where classical algorithms might be too costly. Many parameters can be considered to define “constrained”, from area, latency, power, energy in hardware, to memory consumption, throughput in software. A lot of research has been conducted in this area, leading to new lightweight operating modes, better lightweight cryptographic bricks, improved implementation strategies, etc. Yet, while side-channels attacks have become a critical threat for virtually all security systems, most of the proposed lightweight designs only focus on pure performances for unprotected implementations. This generally tends to bring poor performances for protected implementations, which might be a necessity as lightweight devices are likely to evolve in very adversarial environments.

A classical protection against side-channels attacks is masking. This countermeasure basically consists in randomly splitting every sensitive intermediate variable occurring in the computation into d shares, which are then independently processed by the masked implementation. If masking is to be used as a countermeasure against side-channels attacks, a designer better minimize the number of non-linear operations, as they highly impact the performance of the masking, especially for high-order masking [GR17]. Several block ciphers already started exploring this direction [GGNS13, GLSV15].

Our Contributions. In this article, we propose `Pyjamask`, a new AEAD scheme, with a special focus on efficient side-channels protected implementation in software. Our candidate is composed of a block cipher minimizing the use of non-linear operations and ensuring efficient bitslice implementations, plugged into the well known and well studied `OCB` parallel operating mode. Our block cipher has actually one of the smallest number of AND gates per bit at the time of writing (except LowMC [ARS$^+$15] or Rasta [DEG$^+$18] which are really not lightweight designs, or `3-WAY` [DGV93] and its variant `BASEKING` [Dae95] which are very weak against related-key attacks). Even though `Pyjamask` minimizes such an important criterion, it remains rather lightweight and efficient, thanks to its general bitslice construction that enables computation of all nonlinear gates in parallel. Its performance in software for protected implementations is verified with actual measurements, making it a very suitable candidate for many practical scenarios. For instance, assuming a fast hardware RNG, our implementation of Pyjamask with masking order 128 runs in less than 0.05 seconds on a Cortex-M4 processor (clocked at 168 MHz). According to our estimations, Pyjamask is also fairly lightweight in hardware and it achieves much smaller area than AES for a typical round-based ASIC implementation.

Outline. We first provide in Section 2 the complete description of our designs and in Section 3 their rationale. We conducted a complete security analysis of our designs and report it in Section 4. Finally, we discuss implementation results in Section 5.

2 Specifications

This section gives the specification of the Pyjamask block cipher family and the Pyjamask AEAD algorithms (based on the OCB mode).

2.1 The Pyjamask Block Cipher Family

The block cipher family Pyjamask contains two algorithms: one with a 96-bit block size called Pyjamask-96, and a second with a 128-bit block size called Pyjamask-128. The parameters of the two instances are summarized in Table 1 and detailed hereafter. Our cipher share some similarities with existing ciphers, such as 3-WAY [DGV93], BASEKING [Dae95] or NOEKEON [DPAR00] (for their general structure), ASCON [DEMS16] (for the different linear layers on each slice) or even LowMC [ARS+15] (for its general AND gate minimization).

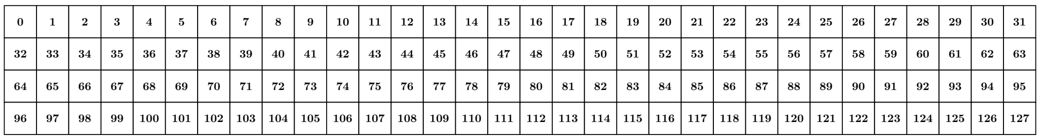
Table 1: Parameters of Pyjamask block ciphers. All the sizes are in bits.

Instance	State size	Rows	Columns	Key size	Rounds
	n	r	n/r	k	
Pyjamask-96	96	3	32	128	14
Pyjamask-128	128	4	32	128	14

The ciphers rely on a Substitution-Mixing structure that transforms the initial plaintext to a ciphertext through several applications of a key-dependent round function. Each round key is derived from the secret key through an iterated key schedule algorithm. In the rest of this section, we first describe the data representation within the cipher. Then, we give a detailed specification of the round function, inverse round function and key schedule. We conclude the section with pseudocode for the encryption, decryption and key schedule algorithms.

2.1.1 Data Representation

The plaintext is initially loaded into the internal states of the ciphers (see Figure 1) which are viewed as a two-dimensional array of bits having r rows and 32 columns ($r = 3$ for Pyjamask-96 and $r = 4$ for Pyjamask-128).



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
32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127

Figure 1: Internal state of Pyjamask-128 with $r = 4$ rows of 32 bits: each cell represents a single bit.

The first (resp. 2nd, 3rd, 4th) group of 4 bytes of the plaintext is loaded into the first (resp. 2nd, 3rd, 4th) row of the state in big endian format. For instance, the 16-byte plaintext

$$[\texttt{0x00},\ \texttt{0x11},\ \texttt{0x22},\ \texttt{0x33},\ \texttt{0x44},\ \ldots\ ,\texttt{0xff}]$$

is loaded into the state as

$$\begin{pmatrix} \texttt{0x00112233} \\ \texttt{0x44556677} \\ \texttt{0x8899aabb} \\ \texttt{0xccddeeff} \end{pmatrix},$$

the first row being `0x00112233` and the last row being `0xccddeeff`. Within one row, the cell of lowest index holds the most significant bit of the row while the cell of greatest index holds the least significant bit of the row. In the above example, the first row is loaded with `0x00112233`, which means that the cell of Index 0 holds the most significant bit of `0x00` (i.e. `0`), and the cell of Index 31 holds the least significant bit of `0x33` (i.e. `1`).

2.1.2 Round Function

The number of rounds applied is 14 for both `Pyjamask-96` and `Pyjamask-128`. The round functions of the two ciphers are similar and only differ due to the extra row present in `Pyjamask-128`. In detail, one round is composed of the following transformations (see also Figure 2):

- AddRoundKey – The first n bits of the key state (defined below) is XORed to the internal state. For `Pyjamask-128`, the full key state is XORed to the internal state. For `Pyjamask-96`, the 3 first rows of the key state are XORed to the internal state.
- SubBytes – The same Sbox is applied to each of the 32 columns of the internal state. For `Pyjamask-96`, the Sbox is S_3 and for `Pyjamask-128`, the Sbox is S_4 (see definitions hereafter).
- MixRows – Each row R_i of the internal state, with $i \in \{0,1,2\}$ for `Pyjamask-96` and $i \in \{0,1,2,3\}$ for `Pyjamask-128` is seen as a column vector of 32 elements in $\mathbb{F}_2$ and is replaced by $\mathbf{M}_i \cdot R_i$. The matrices $\mathbf{M}_i$ are 32×32 constant circulant binary matrices defined below.

After the last round has been applied, a final AddRoundKey operation adds a post-whitening key to the internal state.

Sboxes. The 3-bit Sbox used in `Pyjamask-96` is given by the following lookup table:

$$\mathsf{S}_3 = [1, 3, 6, 5, 2, 4, 7, 0],$$

and the 4-bit Sbox used in `Pyjamask-128` is described by the following lookup table:

$$\mathsf{S}_4 = [\texttt{0x2}, \texttt{0xd}, \texttt{0x3}, \texttt{0x9}, \texttt{0x7}, \texttt{0xb}, \texttt{0xa}, \texttt{0x6}, \texttt{0xe}, \texttt{0x0}, \texttt{0xf}, \texttt{0x4}, \texttt{0x8}, \texttt{0x5}, \texttt{0x1}, \texttt{0xc}].$$

In both cases, the MSB of the inputs and outputs of the Sboxes are located in the top row of the internal state depicted on Figure 2.

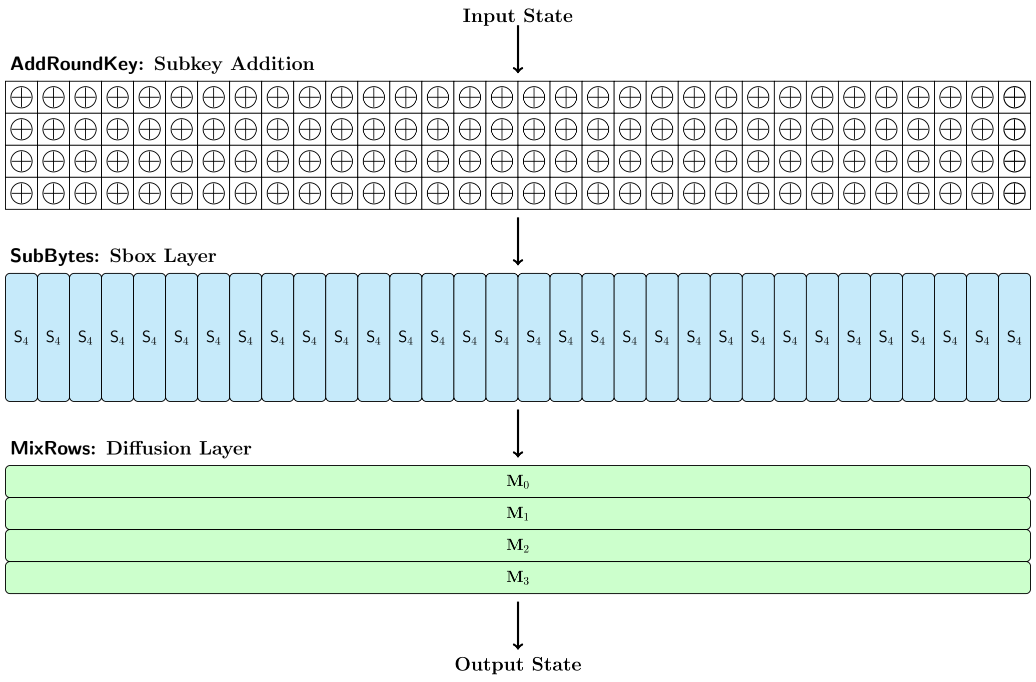


Figure 2: Round function of `Pyjamask-128`.

Matrices. The binary circulant matrices used in the MixRows operation are given below:

$$\begin{aligned}
\mathbf{M}_0 &= \mathsf{cir}\left([1,1,0,1,0,0,0,0,1,0,0,0,0,1,0,0,0,0,1,1,0,0,0,0,1,1,1,0,0,0,1,0]\right),\\
\mathbf{M}_1 &= \mathsf{cir}\left([0,1,0,0,0,0,1,0,0,0,0,0,0,1,1,1,0,1,0,0,0,0,0,1,0,1,1,0,0,0,1,1]\right),\\
\mathbf{M}_2 &= \mathsf{cir}\left([0,0,0,0,0,0,0,0,1,0,1,0,0,1,1,1,1,0,0,1,1,0,1,0,0,1,0,0,1,0,1,1]\right),\\
\mathbf{M}_3 &= \mathsf{cir}\left([0,1,1,0,0,1,0,0,0,0,0,0,1,0,0,1,0,1,0,1,0,0,1,0,1,0,0,0,1,0,0,1]\right).
\end{aligned}$$

Note that $\mathbf{M}_0$, $\mathbf{M}_1$ and $\mathbf{M}_2$ are used in both `Pyjamask-96` and `Pyjamask-128`, but $\mathbf{M}_3$ is only used in `Pyjamask-128`.

2.1.3 Inverse Round Function

As the decryption functionality of some mode of operation requires the inverse block cipher, we also give a description of the inverse round function. It is defined similarly to the forward round function but applies the inverse of the elementary transformations in reversed order. Namely, if performs 14 times the following operations:

- invMixRows – Each row R_i of the internal state, with $i \in \{0,1,2\}$ for `Pyjamask-96` and $i \in \{0,1,2,3\}$ for `Pyjamask-128` is seen as a column vector of 32 elements in $\mathbb{F}_2$ and is replaced by $\mathbf{M}_i^{-1} \cdot R_i$.

- invSubBytes – The inverse Sbox (either S_3^{-1} or S_4^{-1}) is applied to all 32 columns of the internal state.

- invAddRoundKey – The first n bits of the key state is XORed to the internal state.

Again, after the last inverse round, a last subkey is XORed to the internal state.

2.1.4 Key Schedule

The two ciphers `Pyjamask-96` and `Pyjamask-128` share the same key schedule: the only difference is the size of the subkeys extracted from key state that are injected into the internal state during the AddRoundKey operations.

In both ciphers, the secret key consists of 128 bits. It is initially loaded into the 128-bit key state in the same ordering as the internal state (Figure 1). Then, the 128-bit key state undergoes three elementary transformations (see Figure 3):

- MixColumns – Each 4-bit column C_i of the key state is seen as a vector of four element over $\mathbb{F}_2$ and is replaced by $\mathbf{M} \cdot C_i$, where the matrix $\mathbf{M}$ is defined by:

$$\mathbf{M} = \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix} .$$

- MixAndRotateRows – The first row R_0 of the key state is seen as vector of 32 elements over $\mathbb{F}_2$ and is replaced by $\mathbf{M_k} \cdot R_0$, where the matrix $\mathbf{M}_k$ is defined by:

  $$\mathbf{M}_K = \mathsf{cir}\left([1,0,1,0,1,0,0,1,1,1,0,0,1,1,1,0,1,1,0,0,0,0,0,0,1,0,0,0,1,1,1,0]\right).$$

  The second row R_1, third row R_2, and fourth row R_3 are left-rotated by 8, 15, 18 positions. Namely they are replaced by $R_1 \lll 8$, $R_2 \lll 15$, and $R_3 \lll 18$ respectively.

- AddConstant – In the final step, a 32-bit round constant is defined and separated in four bytes which are bitwise added to various parts of the rows of the key state. The last four bits of the constant encode a counter equal to the round number between 0 and 13, and the remaining 28 bits are fixed to a constant represented on Figure 3 using the hexadecimal value `0x243f6a8`:

  $$\mathbf{CONSTANT} = [0,0,1,0,\ 0,1,0,0,\ 0,0,1,1,\ 1,1,1,1,\ 0,1,1,0,\ 1,0,1,0,\ 1,0,0,0].$$

  Then, the most significant byte (MSB) of this constant is XORed to the MSB of the fourth row R_3, the second MSB of this constant is XORed to the MSB of the third row R_2, the third MSB of this constant is XORed to the MSB of the second row R_1, and eventually the LSB of this constant is XORed to the LSB of the first row R_0.

2.1.5 Pseudo-code

We give hereafter some high-level pseudo-code for the encryption, decryption and key schedule algorithms. The Load primitive loads a $4r$-byte input (`plaintext` or `ciphertext`) into an r-row state as described above, with $r \in \{3,4\}$. The Unload primitive consists in the inverse operation. The KeySchedule algorithm takes a 16-byte key (denoted `key`) and produces a table of 15 round keys (denoted `roundkey`$[0:14]$), each round key being made of r rows of the key state. The AddRoundKey, SubBytes and MixRows primitives are the round transformations as defined above. The inverse of the two latter transformations are further denoted InvSubBytes and InvMixRows.

The `Pyjamask-96` and `Pyjamask-128` encryption of `plaintext` under `key` proceeds as follows (Algorithm 1):

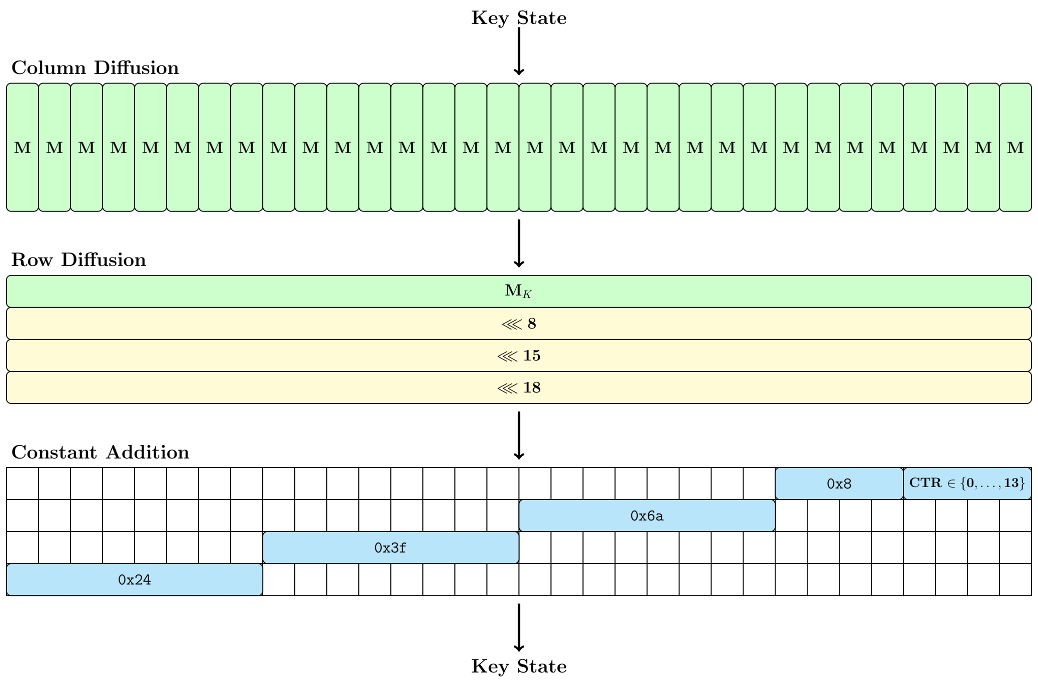


Figure 3: Key schedule of `Pyjamask-96` and `Pyjamask-128`.

Algorithm 1 Encryption

```
1: state ← Load(plaintext)
2: roundkey[0 : 14] ← KeySchedule(key)
3: for $i = 0$ to 13 do
4:     state ← AddRoundKey(state, roundkey[$i$])
5:     state ← SubBytes(state)
6:     state ← MixRows(state)
7: end for
8: state ← AddRoundKey(state, roundkey[14])
9: ciphertext ← Unload(state)
10: return ciphertext
```

Algorithm 2 Decryption

```
1: state ← Load(ciphertext)
2: roundkey[0 : 14] ← KeySchedule(key)
3: state ← AddRoundKey(state, roundkey[14])
4: for $i = 13$ downto 0 do
5:     state ← InvMixRows(state)
6:     state ← InvSubBytes(state)
7:     state ← AddRoundKey(state, roundkey[$i$])
8: end for
9: plaintext ← Unload(state)
10: return plaintext
```

The `Pyjamask-96` and `Pyjamask-128` decryption of `ciphertext` under `key` proceeds as follows (Algorithm 2):

In the following pseudo-code (Algorithm 3), we denote by MixColumns, MixAndRotateRows and AddConstant the key schedule transformations as defined above.

Algorithm 3 Key schedule

```
1: keystate ← Load(key)
2: roundkey[0] ← keystate
3: for $i = 1$ to 14 do
4:     keystate ← MixColumns(keystate)
5:     keystate ← MixAndRotateRows(keystate)
6:     keystate ← AddConstant(keystate, $i$)
7:     roundkey[$i$] ← keystate[$0 : n-1$]
8: end for
9: return roundkey[$0 : 14$]
```

2.2 The Pyjamask AEAD Algorithms

We further specify two authenticated encryption with associated data (AEAD) algorithms in the Pyjamask family. These algorithms are composed of an encryption part and a verification/decryption part.

The encryption part $\mathcal{E}$ takes as input a variable-length plaintext M (with $|M| = m$), a variable-length associated data A (with $|A| = a$), a fixed-length public message number N and a k-bit key K. It outputs a m-bit ciphertext C and a τ-bit tag, denoted tag (with $\tau \in [0, \dots, n]$), i.e. $(C, \texttt{tag}) = \mathcal{E}_K(N, A, M)$. The verification/decryption part $\mathcal{D}$ takes as input a variable-length ciphertext C (with $|C| = m$), a τ-bit authentication tag tag (with $\tau \in [0, \dots, n]$), a variable-length associated data A (with $a = |A|$), a fixed-length public message number N and a k-bit key K. It outputs either an error string $\perp$ to inform that the verification failed, or an m-bit string $M = \mathcal{D}_K(N, A, C, \texttt{tag})$ when the tag is valid.

The two AEAD members of Pyjamask are summarized in Table 2.

Table 2: AEAD members of Pyjamask (all the values are given in bits).

Member Name	Mode	Block Cipher	n	k	$\|N\|$	τ
Pyjamask-128-AEAD †	OCB	Pyjamask-128	128	128	96	128
Pyjamask-96-AEAD	OCB	Pyjamask-96	96	128	64	96

†: Primary member.

OCB Mode. The original OCB [KR14] mode has been designed for 128-bit block ciphers and can hence be used as is for Pyjamask-128. To handle our 96-bit block cipher described above, we have made some slight modifications which are described in Subsection 3.5.

Security Claims. We consider the nonce-respecting authenticated encryption with associated data model for the adversary: nonce values in encryption queries may be chosen by the adversary but they must be distinct. We do not claim security under the case of nonce-misuse or release of unverified plaintext (RUP). He queries for nonce/associated data/message tuples (N, A, M) to the encryption oracle and obtains the corresponding ciphertext/tag (C, T). When interacting with the decryption oracle, he can use any nonce value, even repeating. However, he queries for nonce/associated data/ciphertext/tag tuples (N, A, C, T) to the decryption oracle, but only obtains the corresponding message M if the tag T is valid for that query.

Our security claims are summarized in Table 3. The variables in the table denote the bit security of our designs in terms of calls to the internal primitive, the small constant factors are neglected in these tables. We do not claim security beyond these suggested numbers. A more detailed analysis can be found in the `OCB` [KR14] document.

Table 3: Security claims of `Pyjamask` under the assumption that nonces never repeat. The values are given in bits.

Member Name	**Privacy**	**Authentication**	**Key Recovery**
`Pyjamask-128-AEAD`	64	64	128
`Pyjamask-96-AEAD`	48	48	128

3 Design Rationale

`Pyjamask` aims to provide symmetric (authenticated) encryption enjoying fast software implementations with high levels of security against side-channel attacks. To achieve this goal, `Pyjamask` has been designed to be as lightweight as possible in the presence of *high-order masking* in software, while still enjoying unmasked and/or hardware implementations with satisfying performances.

3.1 Main Design Criteria

When masking is applied to protect a cryptographic implementation against side-channel attacks, each variable in the computation is split into *d shares*. These shares are randomized to wipe out the side-channel information leakage while they are bound to the original variable through a completeness relation. Under some realistic assumptions, the number of shares d, or alternatively the *masking order* $d-1$, has indeed been argued to be a sound security parameter for the masked implementation [CJRR99, PR13, DDF14]. In standard masking schemes, the evaluation of a nonlinear operation has a complexity $O(d^2)$ while for a linear operation the complexity is of $O(d)$, where the linearity is meant with respect to the sharing operation (which is usually the XOR operation). When a masking of high order d is involved, most of the computation is hence dedicated to the masked nonlinear operations while the linear layers are *virtually free*. Several works have recently shown the primacy of *bitslicing* to obtain the best performances for high-order masked implementations [GLSV15, GR16, GR17, JS17, JSV17, GJRS18]. In such implementations, the nonlinear layers are performed through ℓ-bitwise AND operations (ℓ-`AND`), where ℓ is the size of the underlying architecture (typically, ℓ equals 32, or 64 bits). The obtained performances are then highly correlated to the number of ℓ-`AND` operations in the original computation.

`Pyjamask` has been designed to enjoy such fast bitslice implementations in the presence of high-order masking. Specifically, we have favored

- a minimal number of 32-`AND` operations for efficient implementation on 32-bit platforms,
- a parallelization degree to address 64-bit platforms and/or processor with vector instructions,
- a design with reasonable performances for unmasked and/or hardware implementations,
- a design that relies on the well-studied Substitution-Mixing structure involving an Sbox layer, a linear diffusion layer, and a bitwise key addition.

To fulfill the above criteria, we have opted for a design based on the following choices:

- The nonlinear layer is composed of 32 parallel applications of a small Sbox, either a 3-bit or a 4-bit Sbox, which yield two instances of the cipher with either a 96-bit state (`Pyjamask-96`) or a 128-bit state (`Pyjamask-128`). For each instance, the Sbox has the *minimal* cost in terms of AND gates for a non-linear s-box, i.e., m AND gates for the m-bit Sbox, $m \in \{3, 4\}$. This makes a nonlinear layer that can be evaluated with m `32-AND` operations in total.

- The 4-bit Sbox enjoys a possible parallelization of the AND gates, namely it can be evaluated with two pairs of parallel AND gates. As a result, the nonlinear layer of `Pyjamask-128` can be evaluated with two `64-AND` operations in total, which makes it further well suited for 64-bit architectures (or processors with vector instructions).

- Since linear parts are virtually free in the masking world, the linear layer of the `Pyjamask` block cipher has been conceived to provide high diffusion by means of dense 32×32 binary matrices. Different matrices are used for the different 32-bit rows in order to avoid too much alignment. On the other hand, we chose to use circulant matrices to obtain acceptable performances for unmasked and/or hardware implementations.

- The key-schedule of the cipher has been designed to only involve linear operations for an optimal performances in the presence of masking.

We further describe these design choices in the rest of this section.

3.2 Choice of the Sboxes

For concise discussion, we express the lookup table of Sboxes using a sequence of hexadecimal without spacing or comma. For instance, $\mathsf{S}_3 =$ `13652470` and $\mathsf{S}_4 =$ `2d397ba6e0f4851c`.

Our Sboxes selection criteria are as follows:

(C1) To obtain optimal differential and linear properties with as few non-linear gates as possible.

(C2) Avoid cycles in the differential and (resp. linear) transitions with both input and output difference (resp. mask) of Hamming weight one.

(C3) If such cycles cannot be avoided, select one with the longest cycles.

The first criterion **(C1)** is self-explanatory. Note that the best known 3- and 4-bit Sboxes have maximum differential probability (m.d.p.) 2^{-2} and maximum linear approximation (m.l.a) 2^{-2}. To construct the Sboxes used in `Pyjamask` that reach those bounds, we use simple operations as the building blocks: namely, $(a, b, c) \mapsto (b, c \oplus (a \wedge b), a)$ for the 3-bit Sbox and $(a, b, c, d) \mapsto (b, c, d \oplus (a \wedge b), a)$ for the 4-bit Sbox. The choice of these elementary operations is inspired by `PICCOLO` (2011) and `SKINNY` (2016), while the construction in the same philosophy appeared several times, e.g., RadioGatún [BDPA06] (2006) and Xoodoo [DHAK18] (2018). By simply iterating these operations three times for the 3-bit Sbox (resp. four times for the 4-bit Sbox), we obtain Sboxes $\mathsf{S}'_3 =$ `01254736` and (resp. $\mathsf{S}'_4 =$ `012745e98badfc36`) with optimal differential and linear properties.

The criteria **(C2)** and **(C3)** focus on the sub-tables of the differential distribution table (DDT) and the linear approximation table (LAT) where the input and output values have Hamming weight exactly one. Indeed, if there is a 1-cycle (or fixed point) in the sub-table, it implies that active bits in that particular row of the internal state can stay in that row without propagating to other rows. To avoid this undesirable property, we apply some linear transformations L_3^{in} and L_3^{out} (resp. L_4^{in} and L_4^{out}) before and after the Sbox

S_3' (resp. S_4') to obtain linearly equivalent optimal Sboxes but without short cycle (resp. without any cycle) in the differential transitions with both input and output difference of Hamming weight one, same goes for the linear aspects of the Sboxes.

$$L_3^{in} = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix}, \qquad L_3^{out} = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix},$$

$$L_4^{in} = \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \qquad L_4^{out} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix}.$$

Last but not least, we introduce some offset value to both Sboxes to remove fixed points, the offset is denoted by $A_3(x) = x \oplus \texttt{0x1}$ and $A_4(x) = x \oplus \texttt{0x2}$. In the end, the Sboxes that we use in `Pyjamask` are defined as:

$$\mathsf{S}_3 = A_3 \circ L_3^{out} \circ \mathsf{S}_3' \circ L_3^{in},$$
$$\mathsf{S}_4 = A_4 \circ L_4^{out} \circ \mathsf{S}_4' \circ L_4^{in}.$$

In the end, we arrive at our Sboxes S_3 and S_4, The DDT and LAT of S_3 are presented in Table 4 and Table 4, where we highlighted the entries that have both input and output differences/masks having Hamming weight one. Similarly, we give the DDT and LAT of S_4 in Table 5 and Table 6. In all these four tables, rows (resp. columns) represent input (resp. output) differences or masks.

Table 4: DDT and LAT of S_3.

DDT	0	1	2	3	4	5	6	7
0	8	.	.	.	.	.	.	-
1	.	$\boxed{.}$	$\boxed{2}$	2	$\boxed{.}$	.	2	2
2	.	$\boxed{.}$	$\boxed{.}$	.	$\boxed{2}$	2	2	2
3	.	.	2	2	2	2	.	-
4	.	$\boxed{2}$	$\boxed{.}$	2	$\boxed{.}$	2	.	2
5	.	2	2	.	.	2	2	-
6	.	2	.	2	2	.	2	-
7	.	2	2	.	2	.	.	2

LAT	0	1	2	3	4	5	6	7
0	4	.	.	.	.	.	.	.
1	.	$\boxed{.}$	$\boxed{-2}$	-2	$\boxed{.}$	.	2	-2
2	.	$\boxed{.}$	$\boxed{.}$	.	$\boxed{2}$	-2	-2	-2
3	.	.	2	-2	2	2	.	.
4	.	$\boxed{-2}$	$\boxed{.}$	-2	$\boxed{.}$	-2	.	2
5	.	2	2	.	.	-2	2	.
6	.	-2	.	2	2	.	2	.
7	.	-2	2	.	-2	.	.	-2

Table 5: DDT of S_4.

DDT	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0	16	.	.	.	.	.	.	.	.	.	.	.	.	.	.	-
1	.	$\boxed{.}$	$\boxed{.}$	.	$\boxed{.}$	.	.	.	$\boxed{.}$	.	2	2	4	4	2	2
2	.	$\boxed{4}$	$\boxed{.}$	.	$\boxed{4}$	.	.	.	$\boxed{.}$	4	.	.	.	4	.	-
3	.	4	.	.	4	.	.	.	.	.	2	2	.	.	2	2
4	.	$\boxed{.}$	$\boxed{.}$	.	$\boxed{.}$	4	4	.	$\boxed{2}$	2	.	.	.	.	2	2
5	.	.	.	4	.	4	.	.	2	2	2	2	.	.	.	-
6	.	2	2	.	2	.	.	2	2	.	.	2	2	.	.	2
7	.	2	2	.	2	.	.	2	2	.	2	.	2	.	2	-
8	.	$\boxed{.}$	$\boxed{.}$	.	$\boxed{.}$	.	.	.	$\boxed{.}$	.	2	2	4	4	2	2
9	.	.	4	4	.	.	4	4	.	.	.	.	.	.	.	-
a	.	.	2	2	.	.	2	2	.	4	.	.	.	4	.	-
b	.	.	2	2	.	.	2	2	.	.	2	2	.	.	2	2
c	.	.	4	.	.	4	.	.	2	2	2	2	.	.	.	-
d	.	.	.	.	.	4	.	4	2	2	.	.	.	.	2	2
e	.	2	.	2	2	.	2	.	2	.	.	2	2	.	.	2
f	.	2	.	2	2	.	2	.	2	.	2	.	2	.	2	-

Table 6: LAT of S_4.

LAT	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0	8	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
1	.	$\boxed{.}$	$\boxed{-4}$	.	$\boxed{2}$	2	2	−2	$\boxed{.}$	−4	.	.	2	−2	−2	−2
2	.	$\boxed{.}$	$\boxed{.}$	.	$\boxed{.}$	4	.	4	$\boxed{.}$	.	.	.	.	4	.	−4
3	.	4	.	.	−2	−2	2	−2	.	.	−4	.	−2	2	−2	−2
4	.	$\boxed{.}$	$\boxed{.}$	.	$\boxed{.}$	.	.	.	$\boxed{.}$	4	.	4	4	.	−4	.
5	.	.	−4	.	−2	−2	−2	2	.	.	.	−4	2	2	−2	2
6	.	4	.	−4	.	.	.	.	.	.	.	.	4	.	4	.
7	.	.	.	−4	2	−2	−2	−2	.	.	4	.	−2	2	−2	−2
8	.	$\boxed{-2}$	$\boxed{-4}$	−2	$\boxed{2}$	.	−2	.	$\boxed{.}$	2	−4	2	−2	.	2	.
9	.	2	.	2	.	2	−4	−2	4	−2	.	2	.	2	.	2
a	.	−2	.	2	−2	.	−2	−4	.	2	.	−2	2	.	2	−4
b	.	−2	.	−2	.	2	4	−2	4	2	.	−2	.	2	.	2
c	.	−2	4	−2	2	.	−2	.	.	−2	−4	−2	2	.	−2	.
d	.	2	.	2	4	−2	.	2	4	2	.	−2	.	−2	.	−2
e	.	2	.	−2	−2	4	−2	.	.	2	.	−2	−2	−4	−2	.
f	.	2	.	2	4	2	.	−2	−4	2	.	−2	.	2	.	2

3.3 Choice of the Diffusion Matrices

To choose the diffusion matrices, we have run a probabilistic search in a particular subspace fitting the constraints of the ciphers, and simply picked five matrices that ranked best in terms of implementation sizes.

To elaborate on the actual subspace, we first recall the constraints imposed by the design (refer to Subsection 2.1). The matrices have to be defined over $\mathbb{F}_2$ and must be of dimension 32. In terms of security, we would like them to achieve the best possible branch number [Dae95]. Looking at the best known linear codes of these dimensions, one knows that the best theoretically achievable minimum distance is 16 [Gra07, Bro98]. However, one does not know any linear code that reaches that bound: the best achievable one has minimum distance 12. Consequently, in the choice of the diffusion matrices for the `Pyjamask` block cipher, we looked for 32×32 binary diffusion matrices with branch number 12.

To compare two binary matrices having the targeted branch number, we use an implementation-related metric that counts the number of bitwise XORs required to evaluate the matrix multiplication as done in a recent series of academic papers, e.g., [KLSW17, JPST17, DL18]. More specifically, for each candidate matrix, we have run Paar1 algorithm [Paa97], which returns the number of 2-input XOR gates required to implement the evaluation. This measure allows to rank the various matrices and eventually pick the ones that reach branch number 12 and a low number of XOR in the implementation at the same time.

Finally, to restrict the search space, rather than randomly picking 32×32 binary matrices, we have chosen to rely on *circulant* matrices, which can be defined by a single 32-element vector over $\mathbb{F}_2$. To reach branch number 12, this vector necessarily has to have a least 11 nonzero coefficients. As a result, we randomly picked circulant matrices defined by a vector having exactly 11 nonzero elements, checked that their branch numbers was 12, and ranked them accordingly to Paar1's algorithm. We then picked five matrices in the best candidates. Note that this does not ensure that the matrices are optimally lightweight.

3.4 Choice of the Key Schedule

In the key schedule, to differentiate every steps, we chose to inject a round counter to 4 bits of the first row of state. Additionally, to break potential symmetries, it is customary for symmetric ciphers to embed round constant within the key schedule. In `Pyjamask`, we have decided to derive a 28-bit constant from the hexadecimal encoding of the fractional part of $\pi = $ `3.243f 6a88 85a3`, which therefore yields `0x243f6a8`. The same choice has been followed by the designers of `MIDORI` [BBI+15]. We determined to separate this 7-nibble constant and 1-nibble counter to 2 nibbles each and to added each of them for each row. This is to provide better security against the invariant cryptanalysis which will be explained in the security analysis section.

The rotation constants in the key schedule have been chosen to maximize diffusion and to be as close as possible from a multiple of 8. Indeed, as remarked in [BSS+17], on a typical 8-bit micro-controller a rotation by $8k + 2$ is twice as expensive as a rotation by $8k + 1$, a rotation by $8k + 3$ three times as expensive, etc.

3.5 Parameters for the 96-bit Version of the `OCB` Mode

Irreducible Polynomial. The irreducible polynomial of degree 96 has been chosen for its low weight, as listed in [Ser98]. In addition, we note $\mathtt{x} = 2$ is a primitive element.

Stretch-then-shift Hash Function. In [KR11], the empirical result shows that the hash function $H : \{0,1\}^{128} \times [0..63] \to \{0,1\}^{128}$ defined by

$$H(K, x) = (Stretch \ll x)[1..128],$$

where $Stretch = K \| (K \oplus (K \ll c))$ is strongly XOR-universal for $c = 8$. This implies two properties, $H_K(x)$ is uniformly distributed in $\{0,1\}^n$ (universal-1), and for all $x \neq x'$, $H_K(x) \oplus H_K(x')$ is uniformly distributed in $\{0,1\}^n$ (XOR-universal).

We did a similar analysis as described in [KR11] for our 96-bit hash function $H_{96} : \{0,1\}^{96} \times [0..63] \to \{0,1\}^{96}$ defined by

$$H_{96}(K, x) = (Stretch \ll x)[1..96],$$

where $Stretch = K \| (K \oplus (K \ll c))$. We have found several candidates $c \in \{2, 6, 7, 9, 10, 14, \dots\}$ to construct a 96-bit strongly XOR-universal hash function. Notice that for $n = 96$, $c = 8$ does not result in a strongly XOR-universal hash function.

We chose $c = 9$ to be as close as possible from a multiple of 8 for it is minimally better on some platforms (8-bit micro-controllers, when one can only shift by 1, therefore any multiple of 8 ± 1 would be preferred [BSS+17].

4 Security Analysis

We present in this section a preliminary analysis of the block ciphers introduced in `Pyjamask`. While we try to give convincing security arguments and cover the most commonly known cryptanalysis techniques, we are aware we do not cover all possible attack vectors but believe this is sufficient for a design document.

4.1 Differential Analysis

We give in Table 9 lower bounds on the number of active Sboxes for up to four rounds of `Pyjamask-96` and `Pyjamask-128`. To derive those bounds, we have used a SAT approach based on the CryptoSMT framework proposed by Kölbl in [Ste]. We have added both variants of `Pyjamask` to the tool which allows us to search for the optimal differential characteristics taking into account the exact transitions of the difference through the Sbox. We note that due to the high number of variables present in the SAT models, reaching more than four rounds requires long computations which we could not afford. Nonetheless, the bounds obtained provide a strong indication that no high probability characteristic exist for both variants of `Pyjamask`.

In Table 9, we give the bounds on the best differential characteristics possible in terms of the number of active Sboxes. In order to explore the possibility of characteristics with a low number of active Sboxes for more rounds we use the optimal 2-round characteristic and extend it in both directions. Note that the extension in both directions finds the best possible trail, but this does not imply that there is no better trail for 6 rounds exist.

We emphasize that the computations to derive bounds for higher number of rounds by using a general-purpose tool such as SAT are computationally intensive: covering three rounds is still within practical range, but four rounds involve long optimization periods. We may communicate on updated figures in the future.

Searching for Efficient Differential Characteristics

Regarding `Pyjamask-96`, it is still possible to find a highly efficient differential characteristic owing to the differential behaviors of the 3-bit Sbox S_3. At a high level, we first introduce a method to compress the 96-bit state to a 32-bit state, which we call `MiniPyjamask-96`, and then find efficient characteristics by exhaustively trying all differential propagations for `MiniPyjamask-96`.

As indicated by the DDT in Table 4, S_3 allows the iteration of the differential propagations from 1-bit difference to 1-bit difference, namely, the difference `0x1` is propagated to the difference `0x2` with probability 2^{-2}, the difference `0x2` is propagated to the difference `0x4` with probability 2^{-2}, and the difference `0x4` is propagated to the difference `0x1` with probability 2^{-2}. Given this property, we set that all active Sboxes in Round i (resp. $i+1$ and $i+2$) have the input difference `0x1` (resp. `0x2` and `0x4`) and produce the output difference `0x2` (resp. `0x4` and `0x1`). Hence in any round, only one of three rows are active and the other two rows are inactive. This allows us to focus only on the active row to analyze the differential propagation through $\mathsf{MixRows}$. Note that the MSB (resp. LSB) of the Sbox is the top (resp. bottom) row of the state. Therefore,

- After the difference becomes `0x1`, $\mathbf{M}_2$ is applied.
- After the difference becomes `0x2`, $\mathbf{M}_1$ is applied.
- After the difference becomes `0x4`, $\mathbf{M}_0$ is applied.

We are now ready to define `MiniPyjamask-96`. It takes a 32-bit value as input and the round function is a linear function $\mathbf{M}_2$, $\mathbf{M}_1$, and $\mathbf{M}_0$. The order of the linear functions is

- $\mathbf{M}_2$, $\mathbf{M}_1$, and $\mathbf{M}_0$ when the input difference of all active Sboxes in Round 1 is `0x1`.
- $\mathbf{M}_1$, $\mathbf{M}_0$, and $\mathbf{M}_2$ when the input difference of all active Sboxes in Round 1 is `0x2`.
- $\mathbf{M}_0$, $\mathbf{M}_2$, and $\mathbf{M}_1$ when the input difference of all active Sboxes in Round 1 is `0x4`.

In the end, `MiniPyjamask-96` is a 32-bit linear code and the most efficient differential characteristic can be found by searching for the propagation with the lowest Hamming weight. Because the input size is only 32 bits, exhaustive search is feasible. As a result, we found a 5-round propagation with weight 43, which is shown below.

$$\texttt{00a04e67}\ (\mathrm{wt}11) \xrightarrow{\mathbf{M}_1} \texttt{a900010a}\ (\mathrm{wt}7) \xrightarrow{\mathbf{M}_0} \texttt{2040b886}\ (\mathrm{wt}9) \xrightarrow{\mathbf{M}_2}$$
$$\texttt{04010c62}\ (\mathrm{wt}7) \xrightarrow{\mathbf{M}_1} \texttt{0a3a0841}\ (\mathrm{wt}9) \xrightarrow{\mathbf{M}_0} \texttt{d22a6797}$$

This corresponds to the differential characteristic with probability $2^{-2\times 43} = 2^{-86}$ of `Pyjamask-96`. To be precise, the corresponding differential characteristic for `Pyjamask-96` is given in Table 7.

We also confirmed that there is no differential propagation for 6 rounds in this strategy whose probability is higher than 2^{-96} (the weight for `MiniPyjamask-96` is less than 48).

Table 7: Differential characteristic for 5-round Pyjamask-96.

Round	Input to Sbox Layer	Input to Linear Layer	Active
0	00000000 00a04e67 00000000	00a04e67 00000000 00000000	11
1	a900010a 00000000 00000000	00000000 00000000 a900010a	9
2	00000000 00000000 2040b886	00000000 2040b886 00000000	7
3	00000000 04010c62 00000000	04010c62 00000000 00000000	9
4	0a3a0841 00000000 00000000	00000000 00000000 0a3a0841	7
5	00000000 00000000 d22a6797		

Regarding Pyjamask-128, the 4-bit Sbox S_4 does not allow the iteration of the propagation from 1-bit difference to 1-bit difference, which prevents the application of a similar strategy. The best characteristic we found for Pyjamask-128 is shown in Table 8.

Table 8: Differential characteristic for 6-round Pyjamask-128.

Round	Input to Sbox Layer	Input to Linear Layer	Active
0	281a088b200200020000200000080001	08100888280a088b081808092012200a	11
1	1b8983b0175328ad345a10f629c9b369	00000000031b2a090cd88bb03b99b3ff	26
2	00000000000000001180040c9000040c8	0000000000000000000000000180040c9	7
3	000000000000000000000000114a000	0114a000011480000114200000000000	5
4	e6e2431674f49dd216e2eb1900000000	c684f6152430b9cec4b29804b6c6eac3	27
5	041000c802100180060000c8061000c8	061001c800100180061001c804100148	7
6	7d31d40c9f26e70a5b4dcd134fa24e25		

Table 9: Lower bounds on the number of active Sboxes in Pyjamask for one up to four rounds.

Cipher	1	2	3	4
Pyjamask-96	1	12	19	$\in \{27, \ldots, 30\}$
Pyjamask-128	1	12	$\in \{18, 19\}$	≥ 24

4.2 Linear Cryptanalysis

In order to evaluate the security against linear cryptanalysis, we use another approach based on graph search [HV18]. We implemented Pyjamask-96 and Pyjamask-128 in order to search for linear (and differential) trails with the cryptagraph tool provided in the paper. One advantage of this approach is that it allows us to find many trails contributing to the probability of a linear hull (or differential), although we expect this effect to be small due to the weak alignment in the design of Pyjamask.

Finding trails takes a few minutes for 3 rounds (parameters -patterns 1000 -anchors 22 for linear, -patterns 5000 -anchors 21 for differential). For more rounds we could not find any trails due to the computational and memory complexity increasing significantly. The results of our search are summarized in Table 10. Nevertheless, this shows that Pyjamask does not have any high probability trails and is resistant against linear cryptanalysis.

4.3 Algebraic Analysis

In order to estimate the security of Pyjamask against algebraic attacks we first compute a bound on the maximum algebraic degree (see Table 11) for different number of rounds according to the degree estimate given in [BCC11].

A first third-party cryptanalysis paper analyzed Pyjamask-96 in the context of higher-order differential distinguishers [DRS20]. First, the authors corrected the bounds for the

Table 10: Highest linear and differential probability of any r-round differential characteristic found with the graph search tool for `Pyjamask-96` and `Pyjamask-128`.

Cipher	$r = 1$	$r = 2$	$r = 3$
`Pyjamask-96` (linear)	2^{-2}	2^{-24}	2^{-44}
`Pyjamask-96` (diff)	2^{-2}	2^{-24}	2^{-44}
`Pyjamask-128` (linear)	2^{-2}	2^{-24}	2^{-44}
`Pyjamask-128` (diff)	2^{-2}	2^{-24}	2^{-46}

Table 11: Bound on the algebraic degree of `Pyjamask` from 1 to 14 rounds.

Cipher	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**
`Pyjamask-96`	2	4	8	16	32	64	80	88	92	94	95	95	95	95
`Pyjamask-128`	3	9	27	81	112	122	126	127	127	127	127	127	127	127

algebraic degrees of the round-reduced variant of `Pyjamask-128`: we report their values in Table 11 and thank them for spotting this. In the same paper, the authors investigate the resistance of the smallest internal primitive used in `Pyjamask`, namely `Pyjamask-96`, against higher-order differential cryptanalysis. They show that, due to the small algebraic degree of the nonlinear component, it is possible to mount a full-codebook attack that recovers the 128-bit key of `Pyjamask-96` in about 2^{115} calls to the cipher (with all the 2^{96} encrypted blocks). Moreover, they show that for 13 rounds and 12 rounds, variants of the attack require respectively 2^{99} and 2^{96} calls to the primitive. Finally, the authors investigate the impact of their analysis on the actual AEAD primitive `Pyjamask`: due to the data limitation imposed by the mode of operations, the attack can reach up to 6 rounds (with 2^{41} blocks of data and a time complexity of 2^{86} cipher calls).

4.4 Resistance against Invariant Subspace Cryptanalysis

Invariant subspace cryptanalysis is a weak-key attack. The attacker chooses the plaintext and the key values so that the state value only takes a subspace of all the possible values. By observing that the ciphertext is included in the subspace, the attacker can distinguish the cipher. The simplest solution to prevent invariant subspace cryptanalysis is to avoid simple key schedule and the round constant so that the state value can escape from the subspace. Indeed, `Pyjamask` computes several linear computations during the key schedule, which makes the invariant subspace cryptanalysis difficult. The purpose of this section is to discuss that the adoption of the different row-wise linear computations give significant impact on the security against the invariant subspace cryptanalysis for `Pyjamask`.

General Description. Let Q_1, Q_2, Q_3 and Q_4 be the subspace of state values before `AddRoundKey`, before `SubBytes`, after `SubBytes` and after `MixRows`, respectively. The subspace is usually defined as some affine space. In particular, we are interested in subspaces of dimension 1 because they are the smallest non-trivial subspaces and require the least amount of chosen plaintext to launch the attack. We denote such an affine space by $<A> \oplus V$, where $<A>$ is a linear space spanned by a vector A, i.e. $\{0, A\}$, and V is a constant vector. Hence, $Q_i := <A_i> \oplus V_i$. Q_1 is mapped to Q_2 only by the round key addition, thus $A_1 = A_2$ must hold, moreover the round key must be in $<A_1> \oplus V_1 \oplus V_2$, which is a weak-key class for this attack. Attacker's task is to detect such subspaces for a given specification of the `SubBytes`, `MixRows`, and `AddRoundKey`.

Structural Analysis. We suppose that it is possible to find a weak key and constant. Then the possibility of the attack only depends on the `SubBytes` and the `MixRows`. Here we discuss that if the matrices multiplied in `MixRows` are identical for all rows, it is difficult to prevent the attack. Indeed, `Pyjamask` allows an attack if the matrix is the same for all rows. From a different viewpoint, `Pyjamask` avoids the invariant subspace cryptanalysis by adopting different matrcies in the `MixRows` even by assuming the ideal control of the round key and the constant.

We explain the case of `Pyjamask-96`. Suppose that the S-box output value is in the subspace $<A_3> \oplus V_3$. Note that any set of two values $\{u_1, u_2\}$ can be described as the form of $<A_3> \oplus V_3$. Indeed, $\{u_1, u_2\} = \{0, u_1 \oplus u_2\} \oplus u_1$, hence $<u_1 \oplus u_2> \oplus u_1$. (It can also be described as $<u_1 \oplus u_2> \oplus u_2$. In this analysis, the choice of the constant does not have a big impact.) Suppose that the output value is in this subspace for all the S-boxes. The 96-bit state can be described as $(<A_3> \oplus V_3)^{32}$. We then consider how this state is transformed by the `MixRows`, i.e. we analyze the structure of $\texttt{MixRows}(<A_3> \oplus V_3)^{32}$. Due to the linearity, we divide it into two terms: $\texttt{MixRows}((<A_3>)^{32}) \oplus \texttt{MixRows}((V_3)^{32})$. $(V_3)^{32}$ is a constant. Let $v_3^2 v_3^1 v_3^0$ be the bit representation of V_3. Then, $(V_3)^{32}$ looks as

$$\begin{pmatrix} v_3^0 v_3^0 v_3^0 v_3^0 v_3^0 v_3^0 v_3^0 v_3^0 v_3^0 \cdots v_3^0 \\ v_3^1 v_3^1 v_3^1 v_3^1 v_3^1 v_3^1 v_3^1 v_3^1 v_3^1 \cdots v_3^1 \\ v_3^2 v_3^2 v_3^2 v_3^2 v_3^2 v_3^2 v_3^2 v_3^2 v_3^2 \cdots v_3^2 \end{pmatrix}.$$

During `MixRows`, we multiply the matrices $\mathbf{M}_i$ to row i, in which $\mathbf{M}_i$ is a 32×32 circulant matrices of the binary vector whose Hamming weight is an odd number. It is easy to see that the multiplication in each row does not change the value, thus $\texttt{MixRows}((V_3)^{32}) = (V_3)^{32}$, or the constant term is invariant for `MixRows`. Regarding the linear space, let $a^2 a^1 a^0$ be the bitwise representation of the base vector A. We consider that each S-box output is uniformly distributed in $(<A_3> \oplus V_3)^{32}$. Then, $(<A_3>)^{32}$ looks as follows.

$$\begin{pmatrix} s^0 s^0 s^0 s^0 s^0 s^0 s^0 s^0 \cdots s^0 \\ s^1 s^1 s^1 s^1 s^1 s^1 s^1 s^1 \cdots s^1 \\ s^2 s^2 s^2 s^2 s^2 s^2 s^2 s^2 \cdots s^2 \end{pmatrix},$$

where $s^i = 0$ when $a^i = 0$ and s^i is uniformly distributed in $\{0, 1\}$ when $a^i = 1$. (We have an abuse of the notation. Each s^i is chosen independently when $a^i = 1$.) We now consider the matrix multiplication. For the rows with $s^i = 0$, the result of the multiplication is 0, thus the row is invariant. For the rows with s^i uniformly distributed in $\{0, 1\}$, the row covers all possible 2^{32} values. Because the multiplication by $\mathbf{M}_i$ is bijective, the output space covers all possible 2^{32} values. Hence the row is invariant. Therefore, when only 1 bit of $a^2 a^1 a^0$ is 1, the subspace $(<A_3> \oplus V_3)^{32}$ is invariant for the `MixRows` of `Pyjamask`. When more than 1 bits of $a^2 a^1 a^0$ are 1, we need to consider the behaviors of multiple rows. As long as $\mathbf{M}_i$ is identical in all rows, the subspace is invariant. However if $\mathbf{M}_i$ is different for each i like `Pyjamask`, linear subspaces will be different and the attack would not work.

Example. Let us define $Q_3 := \{2, 4\}$, which is $<6> \oplus 2$, hence $a_2 a_1 a_0 = \texttt{110}$ and $v_2 v_1 v_0 = \texttt{010}$. The 96-bit state value before the `MixRows` looks as

$$\begin{pmatrix} \texttt{000000000} \cdots \texttt{0} \\ \texttt{110100010} \cdots \texttt{1} \\ \texttt{001011101} \cdots \texttt{0} \end{pmatrix}.$$

We separate the linear space and the constant vector, and apply the `MixRows`;

$$\texttt{MixRows}\begin{pmatrix} \texttt{000000000} \cdots \texttt{0} \\ \texttt{001011101} \cdots \texttt{0} \\ \texttt{001011101} \cdots \texttt{0} \end{pmatrix} \oplus \texttt{MixRows}\begin{pmatrix} \texttt{000000000} \cdots \texttt{0} \\ \texttt{111111111} \cdots \texttt{1} \\ \texttt{000000000} \cdots \texttt{0} \end{pmatrix}.$$

The value after the `MixRows` is

$$\begin{pmatrix} \mathtt{000000000\cdots0} \\ l_0l_1l_2l_3l_4l_5l_6l_7l_8l_9\cdots l_{31} \\ \overline{l_0l_1l_2l_3l_4l_5l_6l_7l_8l_9}\cdots\overline{l_{31}} \end{pmatrix} \oplus \begin{pmatrix} \mathtt{000000000\cdots0} \\ \mathtt{111111111\cdots1} \\ \mathtt{000000000\cdots0} \end{pmatrix}.$$

l_i and $\overline{l_i}$ are dependent on the matrices. If the same matrix is used in all the rows, $l_i = \overline{l_i}$ holds hence the subspace $<6> \oplus \mathtt{2}$ is invariant. If different matrices are used, the column value does no longer stay in $<6>$.

It is easy to see that if the weight of $a_2a_1a_0$ is 1, namely A_3 is either `1`, `2` or `4`, the subspace is invariant even with different choices of $\mathbf{M}_i$.

In order to construct the invariant subspace for the whole cipher, the property of the S-box that maps $<A> \oplus V$ to $<A'> \oplus V'$ needs to be found. Moreover, to be iterative for each round, we further need the condition $A = A'$. The S-box applications to two values, from $\{u_1, u_2\}$ to $\{S(u_1), S(u_2)\}$, is a mapping from $<u_1 \oplus u_2> \oplus u_1$ to $<S(u_1) \oplus S(u_2)> \oplus S(u_1)$. The exhaustive list of such a propagation is given in Table 12.

Table 12: Output subspace of S_3 for the input $<A> \oplus V$.

$A \backslash V$	0	1	2	3	4	5	6	7
1	$<2> \oplus 1$	$<2> \oplus 1$	$<3> \oplus 5$	$<3> \oplus 5$	$<6> \oplus 2$	$<6> \oplus 2$	$<7> \oplus 0$	$<7> \oplus 0$
2	$<7> \oplus 1$	$<6> \oplus 3$	$<7> \oplus 1$	$<6> \oplus 3$	$<5> \oplus 2$	$<4> \oplus 0$	$<5> \oplus 2$	$<4> \oplus 0$
3	$<4> \oplus 1$	$<5> \oplus 3$	$<5> \oplus 3$	$<4> \oplus 1$	$<2> \oplus 0$	$<3> \oplus 4$	$<3> \oplus 4$	$<2> \oplus 0$
4	$<3> \oplus 1$	$<7> \oplus 3$	$<1> \oplus 6$	$<5> \oplus 0$	$<3> \oplus 1$	$<7> \oplus 3$	$<1> \oplus 6$	$<5> \oplus 0$
5	$<5> \oplus 1$	$<1> \oplus 2$	$<6> \oplus 0$	$<2> \oplus 5$	$<1> \oplus 2$	$<5> \oplus 1$	$<2> \oplus 5$	$<6> \oplus 0$
6	$<6> \oplus 1$	$<3> \oplus 0$	$<4> \oplus 2$	$<1> \oplus 4$	$<4> \oplus 2$	$<1> \oplus 4$	$<6> \oplus 1$	$<3> \oplus 0$
7	$<1> \oplus 0$	$<4> \oplus 3$	$<2> \oplus 4$	$<7> \oplus 2$	$<7> \oplus 2$	$<2> \oplus 4$	$<4> \oplus 3$	$<1> \oplus 0$

From Table 12, there are 6 patterns satisfying $A = A'$.

$$<3> \oplus 5 \to <3> \oplus 4, \qquad <3> \oplus 6 \to <3> \oplus 4, \qquad <5> \oplus 0 \to <5> \oplus 1,$$
$$<5> \oplus 5 \to <5> \oplus 1, \qquad <7> \oplus 3 \to <7> \oplus 2, \qquad <7> \oplus 4 \to <7> \oplus 2.$$

Recall that when the Hamming weight of $A = A'$ is 1, then the subspace is invariant even with different matrices for each row during `MixRows`. As shown above, the Hamming weight of A is more than 1 for all the 6 patterns. Hence, the use of different matrices during `MixRows` helps to resist the invariant subspace cryptanalysis under the full control of the round key and the constant by the attacker.

Note that the actual round key and the constant is not controlled by the attacker. In fact, the key schedule of `Pyjamask` seems hard to control. By applying the same argument, `Pyjamask-128` also resists the invariant subspace cryptanalysis.

5 Implementation

This section focuses on the implementation aspects of `Pyjamask`. We describe efficient implementations in software based on a bitslice strategy and in the presence of masking. We also provide implementation results on a Cortex-M4 processor and compare the masking performance of `Pyjamask` to `AES-128`. We finally give some estimations of the performances in hardware which show that, although optimized for (masked) software implementation, `Pyjamask` is still fairly lightweight in hardware.

5.1 Bitslice Implementation

Bitslice is an general implementation strategy initially imagined for `3-WAY` [DGV93], and later used for `DES` and named bitslice by Biham in [Bih97]. It consists in performing several parallel evaluations of a Boolean circuit in software where the logic gates are replaced by instructions working on registers of several bits. More precisely, each software bitwise instruction corresponds to the simultaneous execution of ℓ Boolean logical gates, where ℓ is the register size on the target architecture. This strategy was originally applied to compute ℓ parallel evaluations of a full block cipher when several blocks must be processed and when parallelism is possible [Bih97]. It can also be applied to speed-up the encryption of a single block with parallel evaluations of the Sboxes [GLSV15]. For standard block ciphers made of SBoxes and linear operations, this implies that the only nonlinear operations in the parallel Sbox processing (and hence in the full cipher) are bitwise AND (or, NAND, OR, NOR) instructions between ℓ-bit registers which is particularly well suited for the efficient application of high-order masking [GR17].

Similarly to `3-WAY` [DGV93], `BASEKING` [Dae95] or `NOEKEON` [DPAR00] and LS-designs [GLSV15], `Pyjamask` is especially tailored for bitslice implementation with a parallel computation of the Sboxes on architectures of size $\ell = 32$. In a bitslice implementation of `Pyjamask`, each row of the state is stored in a 32-bit register (three registers for `Pyjamask-96` and four registers for `Pyjamask-128`). The key state is equally stored row-wise, which makes the key addition very simple (3 or 4 32-`XOR`).

The Sboxes enjoy simple Boolean representations, which makes their bitslice implementation very efficient. Let R_i denotes the ith row register, with $i \in \{0, 1, 2\}$ for `Pyjamask-96` and $i \in \{0, 1, 2, 3\}$ for `Pyjamask-128`. Let $\oplus$ and $\wedge$ respectively denote the 32-`XOR` and 32-`AND` instructions. The Sbox layer can be implemented as follows:

```
SubBytes (Pyjamask-96):
1: $R_0 \leftarrow R_0 \oplus R_1$
2: $R_1 \leftarrow R_1 \oplus R_2$
3: $R_2 \leftarrow R_2 \oplus (R_0 \wedge R_1)$
4: $R_0 \leftarrow R_0 \oplus (R_1 \wedge R_2)$
5: $R_1 \leftarrow R_1 \oplus (R_0 \wedge R_2)$
6: $R_2 \leftarrow R_2 \oplus R_0$
7: $R_0 \leftarrow R_0 \oplus R_1$
8: $R_2 \leftarrow \mathsf{not}(R_2)$
9: $\mathsf{swap}(R_0, R_1)$
```

```
SubBytes (Pyjamask-128):
1: $R_0 \leftarrow R_0 \oplus R_3$
2: $R_3 \leftarrow R_3 \oplus (R_0 \wedge R_1)$
3: $R_0 \leftarrow R_0 \oplus (R_1 \wedge R_2)$
4: $R_1 \leftarrow R_1 \oplus (R_2 \wedge R_3)$
5: $R_2 \leftarrow R_2 \oplus (R_0 \wedge R_3)$
6: $R_2 \leftarrow R_2 \oplus R_1$
7: $R_1 \leftarrow R_1 \oplus R_0$
8: $R_3 \leftarrow \mathsf{not}(R_3)$
9: $\mathsf{swap}(R_2, R_3)$
```

The binary matrix multiplication can be efficiently implemented thanks to the circulant property of the matrix. Let R denote an input row register, let M denote a circulant binary matrix and let C denote the leftmost column of M. By the circulant property, the product $M \cdot R$ satisfies

$$M \cdot R = \big(R[0] \cdot (C \ggg 0)\big) \oplus \big(R[1] \cdot (C \ggg 1)\big) \oplus \cdots \oplus \big(R[31] \cdot (C \ggg 31)\big)$$

where $\ggg$ denotes the right rotation operator and $R[i]$ denotes the ith (leftmost) bit of R. In the above equation, $R[i] \cdot (C \ggg i)$ stands for the scalar product of the 32-bit vector $(C \ggg i)$ by the bit $R[i]$. The binary matrix multiplication can hence be implemented in 32 steps which

- extract the ith bit of R and spread it to 32 bits to get a mask `msk`:

$$\texttt{msk} = \begin{cases} \texttt{0x00000000} & \text{if } R[i] = 0 \\ \texttt{0xffffffff} & \text{if } R[i] = 1 \end{cases}$$

- update an accumulator `acc`:

$$\texttt{acc} = \texttt{acc} \oplus (\texttt{msk} \wedge (C \ggg i)) \ .$$

In C, the computation of msk can be done as $\texttt{msk} = 0 - R[i]$. In ARM v7, it comes for free thanks to the *arithmetic shift right* (ASR), which can be applied to an instruction operand. A slightly faster implementation could be obtained by the use of look-up tables. We purposely avoided such an implementation strategy for the sake of security against cache timing attacks.

5.2 Masked Implementation

In the following, we assume that a source of randomness is available to the masked implementation, such as a physical true random number generator. When queried, the source of randomness output *fresh* random bits, i.e., unpredictable random bits which are statistically independent of the previously generated bits.

In a masked implementation of Pyjamask, the state is split into d shares $\texttt{state}[0], \ldots, \texttt{state}[d-1]$. This number of shares d is called the *masking order* of the implementation in the following. All along the computation, the shares are processed in such a way that the following relation is always satisfied:

$$\texttt{state}[0] \oplus \texttt{state}[1] \oplus \cdots \oplus \texttt{state}[d-1] = \texttt{state} \ .$$

At the beginning of the computation, $d-1$ of the shares are filled with fresh randomness and the last one is computed according to the above equation. The same applies to the key state, which yields shared round keys $\texttt{roundkey}[i][j]$, where $i \in [0, 14]$ is the round index and $j \in [0, d-1]$ is the share index.

The linear operations are applied share-wisely. Namely, the MixRows operation is performed as

$$\textbf{for } j = 0 \textbf{ to } d-1 \textbf{ do: } \texttt{state}[j] \leftarrow \textsf{MixRows}(\texttt{state}[j]) \ .$$

The AddRoundKey operation (for the ith round key) is performed as

$$\textbf{for } j = 0 \textbf{ to } d-1 \textbf{ do: } \texttt{state}[j] \leftarrow \textsf{AddRoundKey}(\texttt{state}[j], \texttt{roundkey}[i][j]) \ .$$

Being fully linear, the key schedule can also be applied share-wisely. Let us denote $\texttt{key}[0], \ldots, \texttt{key}[d-1]$, the shares of the secret key. The key schedule is initially performed as

$$\textbf{for } j = 0 \textbf{ to } d-1 \textbf{ do: } \texttt{roundkey}[0:14][j] \leftarrow \textsf{KeySchedule}(\texttt{key}[j]) \ .$$

Note that in order to keep the consistency, the constant addition is applied in the key schedule for a single share, let's say for $i = 0$, and it is skipped for the other shares.

The Sbox layer is computed according to the circuits described above where each 32-XOR operation is replaced by d share-wise 32-XOR operations and where the 32-AND are performed using a secure masked multiplication scheme. Specifically, we use an ISW *multiply and accumulate* (MACC), which computes the following operation

$$C \leftarrow C \oplus (A \wedge B) \ , \tag{1}$$

on the shares of A, B and C. From the input shares $(A_1, \ldots, A_d)$, $(B_1, \ldots, B_d)$, and $(C_1, \ldots, C_d)$. The ISW MACC operation is simply obtained from the standard ISW AND operation [ISW03] in which the output shares C_i are not initialized to 0 but seen as a third input. Specifically, the ISW MACC proceeds as follows:

```
ISW MACC:
1: for $i = 1$ to $d$ do
2:    $C_i \leftarrow C_i \oplus (A_i \wedge B_i)$
3:    for $j = i + 1$ to $d$ do
4:       $R \leftarrow \$$
5:       $C_i \leftarrow C_i \oplus R$
6:       $C_j \leftarrow C_j \oplus ((A_i \wedge B_j) \oplus R)$
7:       $C_j \leftarrow C_j \oplus (A_j \wedge B_i)$
8:    end for
9: end for
```

In the above pseudocode, $R \leftarrow \$$ denotes the sampling of a random 32-bit value, through a (physical true, or pseudo) random number generator. It can be checked that the output shares satisfy

$$\bigoplus_{j=1}^{d} C_j^{(out)} = \bigoplus_{j=1}^{d} C_j^{(in)} \oplus \left(\bigoplus_{j=1}^{d} A_j\right) \wedge \left(\bigoplus_{j=1}^{d} B_j\right) = C \oplus (A \wedge B) \ .$$

For high-order masking, where d is up to several dozens, the ISW MACC is the most time-consuming operation of the masked implementation of `Pyjamask` since it requires $O(d^2)$ elementary operations against $O(d)$ for the linear parts. This is hence the operation to be primarily optimized. In practice, an implementation of the ISW MACC is composed of logical instructions as well as memory accesses to read and write the input shares and the intermediate results. While the number of logical operations $\{\oplus, \wedge\}$ and the number of RNG invocations are fully determined by the masking order d, an efficient implementation should try to optimize the memory accesses and the loop management.

Cortex-M4 implementations. Our benchmark implementations target ARM (v7) architectures and have been benchmarked on a Cortex-M4 processor. The binary matrix multiplication and ISW MACC routines have been written and optimized at the assembly level. Using the implementation strategy described above, we get a binary matrix multiplication with a total of 32×3 CPU instructions. For the ISW MACC, we have developed two variants. In the basic setting (variant v1) the shares A_i, B_i, C_i are kept in CPU registers during the whole iteration i. The shares A_j, B_j, C_j are read (from memory) and the share C_j is written (in memory) at each iteration j. Three pointers are used for the three sharings. Given the loop indexes and the RNG address, this ISW MACC routine makes full usage of the CPU registers. In the speed-optimized setting (variant v2) the iteration of the main loop are processed by pairs $(i, i+1)$. The shares A_i, B_i, C_i, A_{i+1}, B_{i+1}, C_{i+1} are kept in CPU registers during the whole pair of iterations $(i, i+1)$. This is made possible by only keeping the address of the state and by hardcoding the mapping between the indexes of the state rows and the operands A, B and C. We hence need one instance of the ISW MACC per MACC instruction in the Sbox (i.e. 3 for `Pyjamask-96` and 4 for `Pyjamask-128`). This variant (v2) is hence faster but slightly heavier in code size.

Source Code. The software source code of Pyjamask block cipher is available at:

https://github.com/pyjamask-cipher

5.3 Implementation Results

Our implementation have been benchmarked on an STM32F4 Discovery board. This board embeds an ARM Cortex-M4 processor, which can be clocked up to 168 MHz, and multiple peripherals among which a hardware Random Number Generator (RNG). The

RNG comprises a hardware status register indicating when a new 32-bit word of fresh randomness is available, which occurs every 65 clock cycles (duration of the entropy pooling phase). When fresh randomness is available, it can be accessed through a load instruction in a single clock cycle. We have benchmarked our implementation with the two following RNG modes.

– *Pooling mode:* The RNG routine checks the availability of fresh randomness before reading the RNG output register. This takes a few clock cycles for testing, possibly waiting up to 65 clock cycles (depending on the last read), plus a few clock cycles for reading and managing the routine call. This mode is typically what one should do on the considered STM32F4 board.

– *Fast mode:* The RNG routine simply reads the RNG output register (without wondering whether fresh randomness is ready). This mode simulates a context in which the target architecture has a fast hardware RNG with a pooling phase taking a small number of clock cycles (so that fresh randomness is always ready when the RNG is read).

Table 13 summarizes the obtained performances for the two implementation variants (v1 / v2), the two RNG modes (pooling / fast), and for a masking order d scaling from 4 to 128. These results have been obtained using the `-Ofast` compilation option (which optimizes the timings). In all the scenarios, the performances of encryption and decryption are similar. We observe in particular that for a masking order $d = 128$ our implementations of `Pyjamask-96` and `Pyjamask-128` run in 6.3 and 8.1 megacycles in fast RNG mode, which makes 38 and 48.5 milliseconds assuming a 168 MHz clock. In pooling RNG mode this increases to 28.5 and 37.9 megacycles (which makes 170 and 225.5 milliseconds with a 168 MHz clock).

We note that the code size slightly increases with the masking order up to $d = 16$ and then drops by a factor 2. This is presumably due to the fact that the compiler unrolls the loops in the C code up to a certain number of iterations.

5.4 Comparison

Implementation Results. Up to our knowledge, only a few papers in the literature provide implementation results for masking of high orders (e.g., $d > 4$). In [WVGX15], Wang et al. describe an efficient implementation of `AES` in ARM NEON (typically on a Cortex-A8 processor) for a masking order up to $d = 8$. Their implementation takes advantage of the NEON 128-bit vector instructions, which makes it hard to compare to our implementations.

In [GR17], Goudarzi and Rivain presents several low-level optimization of various masking schemes on ARM v7 architectures. In particular, they benchmark efficient bitslice implementations of `AES` and `PRESENT` for a masking order up to $d = 11$. We have benchmarked their bitslice `AES` implementation on the STM32F4 board. The results are given in Table 14 and compared to our implementations of `Pyjamask-128`. We note that the `AES` implementation of [GR17] takes an expanded masked key as input and does not perform the `AES` key schedule. We see that compared to this optimized implementation, our implementation of `Pyjamask-128` (v2) is between 3 and 4 times faster at high orders.

Finally, Journault and Standaert report efficient masked bitslice implementations of `AES` and `Fantomas` in [JS17] at the order $d = 32$. They give performance results for a Cortex-M4 processor embedded on a SAM4C-EK evaluation board. On this board the pooling phase of the RNG takes 80 clock cycles, which is slightly slower than on the STM32F4 board but the results are still comparable. Their `AES` implementation runs in 9.7 megacycles and their `Fantomas` implementation in 4.1 megacycles. In comparison, our

Table 13: Performance benchmark on ARM Cortex-M4.

	Variant	TRNG	$d = 4$	$d = 8$	$d = 16$	$d = 32$	$d = 64$	$d = 128$
			Timings (kilocycles)					
`Pyjamask-96`	v1	pooling	59	178	606	2213	8173	30772
	v1	fast	41	95	249	736	2419	8253
	v2	pooling	55	165	556	2018	7397	28518
	v2	fast	38	86	215	604	1898	6341
`Pyjamask-128`	v1	pooling	74	230	792	2918	10807	40890
	v1	fast	51	119	316	948	3145	10858
	v2	pooling	69	213	726	2657	9785	37901
	v2	fast	47	106	267	758	2398	8102
			RAM (kilobytes)					
`Pyjamask-96`	v1/v2	-	1.2	2.2	4.1	8.2	16.2	32.3
`Pyjamask-128`	v1/v2	-	1.2	2.2	4.2	8.3	16.6	32.9
			Code size (bytes)					
`Pyjamask-96`	v1	-	3712	5296	5320	2892	2896	2920
	v2	-	5340	6922	6952	4524	4528	4552
`Pyjamask-128`	v1	-	4070	5776	5686	3158	3198	3198
	v2	-	5696	7418	7306	4778	4818	4818
`Pj-96 + Pj-128`	v1	-	6652	9940	9872	4920	4964	4988
	v2	-	8232	11516	11452	6504	6548	6572

Table 14: Performance comparison on ARM Cortex-M4.

	Variant	TRNG	$d = 4$	$d = 8$	$d = 16$	$d = 32$	$d = 64$	$d = 128$
			Timings (kilocycles)					
`AES-128` [GR17]	-	pooling	153	547	2072	8073	30572	121430
	-	fast	86	237	746	2592	9597	36882
`Pyjamask-128`	v1	pooling	74	230	792	2918	10807	40890
	v1	fast	51	119	316	948	3145	10858
	v2	pooling	69	213	726	2657	9785	37901
	v2	fast	47	106	267	758	2398	8102
			RAM (kilobytes)					
`AES-128` [GR17]	-	-	2.4	4.8	9.6	19.2	38.4	76.8
`Pyjamask-128`	v1/v2	-	1.2	2.2	4.2	8.3	16.6	32.9
			Code size (bytes)					
`AES-128` [GR17]	-	-	7532	7532	7532	7532	7532	7532
`Pyjamask-128`	v1	-	4070	5776	5686	3158	3198	3198
	v2	-	5696	7418	7306	4778	4818	4818

implementations of `Pyjamask-128` at order $d = 32$ run in 2.9 megacycles (v1) and 2.6 megacycles (v2) in pooling RNG mode.

High-Level Comparison. We provide hereafter a more general comparison of the `Pyjamask` design to the state of the art. As explained in Section 3, the prime efficiency parameter for a masked bitslice implementation at high orders on a ℓ-bit architecture is the number ℓ-`AND`. We therefore report in Table 15 the counts of 32-`AND` and 64-`AND` for several 96-bit and

128-bit ciphers. For `Pyjamask-96`, the Sbox is composed of 3 multiplications. This implies that we can compute the full Sbox layer with three 32-`AND`. For `Pyjamask-128`, the Sbox is composed of 4 multiplications that can be computed as 2 pairs of parallel multiplications. This implies that we can compute the full Sbox layer with four 32-`AND` or with two 64-`AND`. For completeness we also report the total count of Boolean `AND` operations. Note that we ignore the `AND` operations in the key schedule.

First bitslice oriented block ciphers, such as 96-bit `3-WAY` [DGV93] or its 192-bit variant `BASEKING` [Dae95], already provided a very small number of AND gates count. In fact, our proposal `Pyjamask-96` is very similar to `3-WAY` and its three 32-bit word general structure. We note however that due to the absence of an actual key schedule, `3-WAY` is very vulnerable to related-key cryptanalysis: it was shown [KSW97] that it could be broken with only a single related key query and about 2^{22} chosen plaintexts. This attack also applies to `BASEKING`. In the case of `Pyjamask`, we decided to avoid or at least complicate such practical attacks by using a relatively simple key schedule, with little impact on performances. While the question on the relevance in practice of related-key attacks remains open, such related-key differential paths could have devastating effects if the cipher were to be used in some block-cipher based compression function. We believe this robustness for a very small performance cost is a good trade-off.

Table 15: Comparison of the bitwise multiplicative complexity of several ciphers.

	key size	**# rounds**	**#** `AND`	**#** 32-`AND`	**#** 64-`AND`
96-bit block ciphers					
`SIMON-96/96` [BSS+17]	96	52	4992	104*	52*
`SIMON-96/144` [BSS+17]	144	54	5184	108*	54*
`3-WAY` [DGV93]	96	11	1056	**33**	33*
`Pyjamask-96`	128	14	1344	**42**	42*
128-bit block ciphers					
`LowMC-128` ($m = 3$) [ARS+15]	128	88	792	88*	88*
`AES-128` [GR17]	128	10	5120	160	100*
`SIMON-128/128` [BSS+17]	128	68	4352	136	68
`NOEKEON` [DPAR00]	128	16	2048	64	32
`Robin` [GLSV15]	128	16	3072	96	96*
`Fantomas` [GLSV15]	128	12	2304	72	72*
`Mysterion-128` [JSV17]	128	12	1536	**48**	**24**
`Pyjamask-128`	128	14	1792	**56**	**28**

* Does not achieve full parallelization (i.e. some registers are not full with data).

5.5 Hardware Implementation

We provide hereafter some estimation for an encryption-only round-based implementations of `Pyjamask-128` on ASIC using UMC 180 technology.

In order to minimize the number of rounds needed and thus the amount of non-linear operations, `Pyjamask` uses an important amount of binary XOR operations. As XOR gates are not so cheap (2.67 GE[1] on UMC 180 for example using `MAOI1` gates, compared with 1 GE of a NAND gate), this will have a negative impact on the area of ASIC implementations.

Memory Size. For an internal state of 128 bits and a 128-bit key, 256 bits need to stored, which amounts to about $256 \cdot 4.67 = 1195$ GE.

[1]A *Gate Equivalent* (GE) is the area of the smallest 2-input NAND gate in the cell library under consideration

Sbox. In Pyjamask-128, there are 32 Sboxes of 4 bits, and each can be implemented with 4 AND gates and 7 XOR/XNOR gates. This amounts to about $32 \times (4 \cdot 1.33 + 7 \cdot 2.67) = 768$ GE.

Binary Matrices. The cipher relies on five matrices of dimension 32 over $\mathbb{F}_2$: four to update the internal state, and one for the key update. Using Paar's algorithm [Paa97], we have evaluated that the matrices $\mathbf{M}_0$, $\mathbf{M}_1$, $\mathbf{M}_2$, $\mathbf{M}_3$ and $\mathbf{M}_K$ can be implemented using 175, 144, 199, 144 and 200 XOR gates respectively (maybe less if better implementations are found using more advanced algorithms). This amounts to $(175 + 144 + 199 + 144 + 200) \cdot 2.67 = 2302$ GE for encryption only. Since our operating mode requires inverse operation of the matrices for the decryption process, we evaluated again with Paar's algorithm that the matrices $\mathbf{M}_0$, $\mathbf{M}_1$, $\mathbf{M}_2$, $\mathbf{M}_3$ and $\mathbf{M}_K$ can be implemented using 344, 345, 346, 347 and 345 XOR gates respectively with their inverses. This amounts to $(344 + 345 + 346 + 347 + 345) \cdot 2.67 = 4611$ GE for encryption and decryption.

Column Diffusion of Key Schedule. The key scheduling operation also relies on 32 binary matrices of dimension 4, and each can be implemented with only 6 XORs. This amounts to about $32 \times (6 \cdot 2.67) = 512$ GE.

Key Addition. To XOR the subkey into the state, 128 XOR gates with two inputs are requires. This amounts to about $128 \times 2.67 = 342$ GE.

Constant Addition. The XOR of round constant is negligible, only a dozen 1 bits have to be XORed.

Control Logic. Extra logic to control the execution flow is hard to predict, but for lightweight ciphers it usually contributes to a small percentage of the total area. Moreover, Pyjamask has a very regular structure that should reduce the significance of that part in the whole implementation size. Therefore, we will not count the control logic in our estimation.

In total, we estimate that a Pyjamask-128 round-based implementation (encryption only) should require about 5200 GE (and 14 cycles), which remains much better than an AES round-based implementation [SMTM01]. With both encryption and decryption one would require about 7500 GE. We emphasize that this is only a very rough estimation.

A possible better trade-off than this basic round-based implementation would be to rely on the circulant structure on the diffusion matrices and to compute them in a circulant way: this would allow a important reduction of the implementation size at the expense of using more cycles. We note that other performance improvements could probably be considered: for instance, better implementations of the matrices (requiring less XOR gates), use of more complex gates such as XOR3 that compute the XOR of three values (one XOR3 gate is generally cheaper than two XOR2 gates), etc.

Acknowledgements

The authors would like to thank the anonymous referees for their very valuable and constructive comments. We are especially gratefull to Joan Daemen for his thorough reviews and comments. We are also grateful to Quan Quan Tan for providing an easy-to-use implementation of Paar's algorithm. This work was partially supported by the French FUI-AAP25 VeriSiCC project.

References

[AP13] Nadhem J. AlFardan and Kenneth G. Paterson. Lucky thirteen: Breaking the TLS and DTLS record protocols. In *2013 IEEE Symposium on Security and Privacy*, pages 526–540, Berkeley, CA, USA, May 19–22, 2013. IEEE Computer Society Press.

[ARS+15] Martin R. Albrecht, Christian Rechberger, Thomas Schneider, Tyge Tiessen, and Michael Zohner. Ciphers for MPC and FHE. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology – EUROCRYPT 2015, Part I*, volume 9056 of *Lecture Notes in Computer Science*, pages 430–454, Sofia, Bulgaria, April 26–30, 2015. Springer, Heidelberg, Germany.

[BBI+15] Subhadeep Banik, Andrey Bogdanov, Takanori Isobe, Kyoji Shibutani, Harunaga Hiwatari, Toru Akishita, and Francesco Regazzoni. Midori: A block cipher for low energy. In Tetsu Iwata and Jung Hee Cheon, editors, *Advances in Cryptology - ASIACRYPT 2015 - 21st International Conference on the Theory and Application of Cryptology and Information Security, Auckland, New Zealand, November 29 - December 3, 2015, Proceedings, Part II*, volume 9453 of *Lecture Notes in Computer Science*, pages 411–436. Springer, 2015.

[BCC11] Christina Boura, Anne Canteaut, and Christophe De Cannière. Higher-order differential properties of keccak and *Luffa*. In *FSE*, volume 6733 of *Lecture Notes in Computer Science*, pages 252–269. Springer, 2011.

[BDPA06] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. RadioGatún, a belt-and-mill hash function. *IACR Cryptology ePrint Archive*, 2006:369, 2006.

[Bih97] Eli Biham. A fast new DES implementation in software. In Eli Biham, editor, *Fast Software Encryption – FSE'97*, volume 1267 of *Lecture Notes in Computer Science*, pages 260–272, Haifa, Israel, January 20–22, 1997. Springer, Heidelberg, Germany.

[Bro98] Andries E. Brouwer. Bounds on the size of linear codes. In Vera S. Pless and W.Cary Huffman, editors, *Handbook of Coding Theory*, chapter 4, pages 295–461. Elsevier, Amsterdam, 1998.

[BSS+17] Ray Beaulieu, Douglas Shors, Jason Smith, Stefan Treatman-Clark, Bryan Weeks, and Louis Wingers. Notes on the design and analysis of SIMON and SPECK. Cryptology ePrint Archive, Report 2017/560, 2017. https://eprint.iacr.org/2017/560.

[CAE] Caesar: Competition for authenticated encryption: Security, applicability, and robustness. https://competitions.cr.yp.to/caesar.html.

[CJRR99] Suresh Chari, Charanjit S. Jutla, Josyula R. Rao, and Pankaj Rohatgi. Towards sound approaches to counteract power-analysis attacks. In Michael J. Wiener, editor, *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings*, volume 1666 of *Lecture Notes in Computer Science*, pages 398–412. Springer, 1999.

[Dae95] Joan Daemen. *Cipher and hash function design strategies based on linear and differential cryptanalysis.* PhD thesis, 1995.

[DDF14] Alexandre Duc, Stefan Dziembowski, and Sebastian Faust. Unifying leakage models: From probing attacks to noisy leakage. In Phong Q. Nguyen and Elisabeth Oswald, editors, *Advances in Cryptology - EUROCRYPT 2014 - 33rd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Copenhagen, Denmark, May 11-15, 2014. Proceedings*, volume 8441 of *Lecture Notes in Computer Science*, pages 423–440. Springer, 2014.

[DEG$^+$18] Christoph Dobraunig, Maria Eichlseder, Lorenzo Grassi, Virginie Lallemand, Gregor Leander, Eik List, Florian Mendel, and Christian Rechberger. Rasta: A cipher with low ANDdepth and few ANDs per bit. In Hovav Shacham and Alexandra Boldyreva, editors, *Advances in Cryptology – CRYPTO 2018, Part I*, volume 10991 of *Lecture Notes in Computer Science*, pages 662–692, Santa Barbara, CA, USA, August 19–23, 2018. Springer, Heidelberg, Germany.

[DEMS16] Christoph Dobraunig, Maria Eichlseder, Florian Mendel, and Martin Schläffer. Ascon v1.2. Submission to the CAESAR competition: `http://competitions.cr.yp.to/round3/asconv12.pdf`, 2016.

[DGV93] Joan Daemen, René Govaerts, and Joos Vandewalle. A New Approach to Block Cipher Design. In Ross J. Anderson, editor, *Fast Software Encryption, Cambridge Security Workshop, Cambridge, UK, December 9-11, 1993, Proceedings*, volume 809 of *Lecture Notes in Computer Science*, pages 18–32. Springer, 1993.

[DHAK18] Joan Daemen, Seth Hoffert, Gilles Van Assche, and Ronny Van Keer. The design of Xoodoo and Xoofff. *IACR Trans. Symmetric Cryptol.*, 2018(4):1–38, 2018.

[DL18] Sébastien Duval and Gaëtan Leurent. Mds matrices with lightweight circuits. *IACR Transactions on Symmetric Cryptology*, 2018(2):48–78, Jun. 2018.

[DPAR00] Joan Daemen, Michaël Peeters, Gilles Van Assche, and Vincent Rijmen. Nessie Proposal: the Block Cipher Noekeon. Nessie submission, 2000. `http://gro.noekeon.org/`.

[DRS20] Christoph Dobraunig, Yann Rotella, and Yanne Schoone. Algebraic and higher-order differential cryptanalysis of pyjamask-96. *IACR Transactions on Symmetric Cryptology*, 2020(1), 2020.

[GGNS13] Benoît Gérard, Vincent Grosso, María Naya-Plasencia, and François-Xavier Standaert. Block ciphers that are easier to mask: How far can we go? In Guido Bertoni and Jean-Sébastien Coron, editors, *Cryptographic Hardware and Embedded Systems – CHES 2013*, volume 8086 of *Lecture Notes in Computer Science*, pages 383–399, Santa Barbara, CA, USA, August 20–23, 2013. Springer, Heidelberg, Germany.

[GJRS18] Dahmun Goudarzi, Anthony Journault, Matthieu Rivain, and François-Xavier Standaert. Secure multiplication for bitslice higher-order masking: Optimisation and comparison. In Junfeng Fan and Benedikt Gierlichs, editors, *COSADE 2018: 9th International Workshop on Constructive Side-Channel Analysis and Secure Design*, volume 10815 of *Lecture Notes in Computer Science*, pages 3–22, Singapore, April 23–24, 2018. Springer, Heidelberg, Germany.

[GLSV15] Vincent Grosso, Gaëtan Leurent, François-Xavier Standaert, and Kerem Varici. LS-designs: Bitslice encryption for efficient masked software implementations.

In Carlos Cid and Christian Rechberger, editors, *Fast Software Encryption – FSE 2014*, volume 8540 of *Lecture Notes in Computer Science*, pages 18–37, London, UK, March 3–5, 2015. Springer, Heidelberg, Germany.

[GR16] Dahmun Goudarzi and Matthieu Rivain. On the multiplicative complexity of Boolean functions and bitsliced higher-order masking. In Benedikt Gierlichs and Axel Y. Poschmann, editors, *Cryptographic Hardware and Embedded Systems – CHES 2016*, volume 9813 of *Lecture Notes in Computer Science*, pages 457–478, Santa Barbara, CA, USA, August 17–19, 2016. Springer, Heidelberg, Germany.

[GR17] Dahmun Goudarzi and Matthieu Rivain. How fast can higher-order masking be in software? In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *Advances in Cryptology – EUROCRYPT 2017, Part I*, volume 10210 of *Lecture Notes in Computer Science*, pages 567–597, Paris, France, April 30 – May 4, 2017. Springer, Heidelberg, Germany.

[Gra07] Markus Grassl. Bounds on the minimum distance of linear codes and quantum codes. Online available at http://www.codetables.de, 2007. Accessed on 2019-02-19.

[HV18] Mathias Hall-Andersen and Philip S. Vejre. Generating graphs packed with paths estimation of linear approximations and differentials. *IACR Trans. Symmetric Cryptol.*, 2018(3):265–289, 2018.

[ISW03] Yuval Ishai, Amit Sahai, and David Wagner. Private circuits: Securing hardware against probing attacks. In Dan Boneh, editor, *Advances in Cryptology – CRYPTO 2003*, volume 2729 of *Lecture Notes in Computer Science*, pages 463–481, Santa Barbara, CA, USA, August 17–21, 2003. Springer, Heidelberg, Germany.

[JPST17] Jérémy Jean, Thomas Peyrin, Siang Sim, and Jade Tourteaux. Optimizing implementations of lightweight building blocks. *IACR Transactions on Symmetric Cryptology*, 2017(4):130–168, Dec. 2017.

[JS17] Anthony Journault and François-Xavier Standaert. Very high order masking: Efficient implementation and security evaluation. In Wieland Fischer and Naofumi Homma, editors, *Cryptographic Hardware and Embedded Systems – CHES 2017*, volume 10529 of *Lecture Notes in Computer Science*, pages 623–643, Taipei, Taiwan, September 25–28, 2017. Springer, Heidelberg, Germany.

[JSV17] Anthony Journault, François-Xavier Standaert, and Kerem Varici. Improving the security and efficiency of block ciphers based on ls-designs. *Des. Codes Cryptography*, 82(1-2):495–509, 2017.

[KLSW17] Thorsten Kranz, Gregor Leander, Ko Stoffelen, and Friedrich Wiemer. Shorter linear straight-line programs for mds matrices. *IACR Transactions on Symmetric Cryptology*, 2017(4):188–211, Dec. 2017.

[KR11] Ted Krovetz and Phillip Rogaway. The software performance of authenticated-encryption modes. In *Fast Software Encryption - 18th International Workshop, FSE 2011, Lyngby, Denmark, February 13-16, 2011, Revised Selected Papers*, pages 306–327, 2011.

[KR14] Ted Krovetz and Phillip Rogaway. The OCB authenticated-encryption algorithm. *RFC*, 7253:1–19, 2014.

[Kra01] Hugo Krawczyk. The order of encryption and authentication for protecting communications (or: How secure is SSL?). In Joe Kilian, editor, *Advances in Cryptology – CRYPTO 2001*, volume 2139 of *Lecture Notes in Computer Science*, pages 310–331, Santa Barbara, CA, USA, August 19–23, 2001. Springer, Heidelberg, Germany.

[KSW97] John Kelsey, Bruce Schneier, and David A. Wagner. Related-key cryptanalysis of 3-WAY, Biham-DES, CAST, DES-X, NewDES, RC2, and TEA. In Yongfei Han, Tatsuaki Okamoto, and Sihan Qing, editors, *Information and Communication Security, First International Conference, ICICS'97, Beijing, China, November 11-14, 1997, Proceedings*, volume 1334 of *Lecture Notes in Computer Science*, pages 233–246. Springer, 1997.

[Paa97] Christof Paar. Optimized arithmetic for Reed-Solomon encoders. In *Proceedings of IEEE International Symposium on Information Theory*, page 250. IEEE, 1997.

[PR13] Emmanuel Prouff and Matthieu Rivain. Masking against side-channel attacks: A formal security proof. In Thomas Johansson and Phong Q. Nguyen, editors, *Advances in Cryptology – EUROCRYPT 2013*, volume 7881 of *Lecture Notes in Computer Science*, pages 142–159, Athens, Greece, May 26–30, 2013. Springer, Heidelberg, Germany.

[Rog02] Phillip Rogaway. Authenticated-encryption with associated-data. In Vijayalakshmi Atluri, editor, *ACM CCS 2002: 9th Conference on Computer and Communications Security*, pages 98–107, Washington, DC, USA, November 18–22, 2002. ACM Press.

[Ser98] Gadiel Seroussi. Table of low-weight binary irreducible polynomials. In *HP Labs Technical Reports*, pages 98–135, 1998.

[SMTM01] Akashi Satoh, Sumio Morioka, Kohji Takano, and Seiji Munetoh. A Compact Rijndael Hardware Architecture with S-Box Optimization. In Colin Boyd, editor, *Advances in Cryptology - ASIACRYPT 2001, 7th International Conference on the Theory and Application of Cryptology and Information Security, Gold Coast, Australia, December 9-13, 2001, Proceedings*, volume 2248 of *Lecture Notes in Computer Science*, pages 239–254. Springer, 2001.

[Ste] Stefan Kölbl. CryptoSMT: An easy to use tool for cryptanalysis of symmetric primitives. https://github.com/kste/cryptosmt.

[WVGX15] Junwei Wang, Praveen Kumar Vadnala, Johann Großschädl, and Qiuliang Xu. Higher-order masking in practice: A vector implementation of masked AES for ARM NEON. In Kaisa Nyberg, editor, *Topics in Cryptology – CT-RSA 2015*, volume 9048 of *Lecture Notes in Computer Science*, pages 181–198, San Francisco, CA, USA, April 20–24, 2015. Springer, Heidelberg, Germany.