Pyjamask

v1.0

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] ANSSI, Paris, France
Jean.Jeremy@gmail.com

[3] CyberCrypt A/S, Denmark
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, Japan
Sasaki.Yu@lab.ntt.co.jp

Table of Contents

1 Introduction

This document specifies `Pyjamask`, an authenticated encryption with associated data (AEAD) scheme based on a new block cipher (BC) called `Pyjamask` and on the AEAD operating mode `OCB`.

`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. Our newly designed block cipher `Pyjamask` has thus the smallest number of AND gates per bit as of today (except `LowMC` [2] or `Rasta` [12] which work on unconventional plaintext/key sizes). 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.

As for the operating mode, we adopt the provably secure AEAD mode `OCB` [27]. It has been extensively studied and has the benefit to offer full parallelization. Of course, other block cipher-based modes such as `COFB` [8] can be considered as well if other performance profiles are to be targeted.

Organization of the document. In Section 2, we first introduce the recommended parameter sets, the various members of the `Pyjamask` family as well as their respective security claims. We then describe `Pyjamask` and recall the AEAD mode `OCB`. We provide a security analysis of `Pyjamask` (and in particular `Pyjamask`) in Section 4 and explain the design rationale in Section 3. Finally, we provide performances measurements/estimations of `Pyjamask` in Section 5.

2 Specification of `Pyjamask`

We describe here the full specification of our submission `Pyjamask`. In the first section below, after some preliminary definitions and notations, we start by giving the two members within the submission. Then, we describe the mode of operation for authenticated encryption `OCB` we use in `Pyjamask` in its original form, the small modifications we have made to accommodate it to our constraints, and finally we describe two new block ciphers used within these modes: `Pyjamask-96` and `Pyjamask-128`.

2.1 Preliminary

Notations. We denote by $\mathbb{F}_2$ the finite field having two elements. From a vector r of t elements over $\mathbb{F}_2$, we define the matric $\mathsf{cir}(r)$ as the circulant binary matrix over $\mathbb{F}_2$ where the i-th row equals the vector r rotated by i positions to the right, $0 \leq i < t$.

For a given block cipher E, we denote $E_K(P)$ the encryption of the n-bit plaintext P with k-bit key K. Similarly, D represents the decryption operation, and we have $D_K(E_k(P)) = E_K(D_k(P)) = P$ for all P.

The concatenation operation is represented by $\|$ and $\mathsf{pad10}^*$ is the function that applies the 10^* padding on n bits, i.e. $\mathsf{pad10}^*(X) = X\|1\|0^{n-|X|-1}$ when $|X| < n$. For the empty string ϵ, the 10^* padding does not add any bit: $\mathsf{pad10}^*(\epsilon) = \epsilon$. Finally, we denote by $X \lll a$ the word X rotated by a positions to the left.

High-Level Description. The cryptographic algorithms defined in the `Pyjamask` submission are all authenticated encryption schemes with associated data (AEAD), which 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 $\texttt{tag}$ (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 $\texttt{tag}$ (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.

2.2 Family Members and Security Claims

We further specify two AEAD algorithms in the `Pyjamask` family, as show in Table 13.

Table 1: Submission members for `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.

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. 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 2. The variables in the table denote the required workload, in terms of data complexity, of an adversary to break the cipher, in base-2 logarithm. The data complexity of attacker consists of the number of queries and the total amount of processed message blocks. If it reaches the suggested number, then there is no security guarantee anymore, and the cipher can be broken. For simplicity, small constant factors, which are determined from the concrete security bounds, are neglected in these tables. A more detailed analysis can be found in the `OCB` [27] document.

Table 2: 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

2.3 OCB Mode of Operation

2.3.1 Original Description

In addition to the block cipher E, we require the doubling operation in the finite field dbl, which applies to a 128-bit string as:

$$\mathsf{dbl}(x) = \begin{cases} x \ll 1 & \text{if } \mathrm{msb}(x) = 0 \\ (x \ll 1) \oplus \mathtt{0x87} & \text{otherwise.} \end{cases} \tag{1}$$

We further require the function $\mathtt{ntz}(x)$, which computes the number of trailing zero bits in the base-2 representation of x.

We split the description in three parts as given in RFC 7253 [27]: *Authentication*, *Encryption* and *Decryption/Verification*. For all functions, we first have to compute:

$$\begin{aligned} L_* &= E_K(0), \\ L_\$ &= \mathsf{dbl}(L_*), \\ L_0 &= \mathsf{dbl}(L_\$), \\ L_i &= \mathsf{dbl}(L_{i-1}). \end{aligned} \tag{2}$$

Authentication Part. It takes as input the key K and the associated data A and produces the intermediate value $\mathtt{auth}$ (see Figure 1). We denote this function as $(K, A) \mapsto \mathtt{HASH}(K, A)$. We consider A as a sequence of 128-bit blocks. Let l be the largest integer such that $128l \leq |A|$, then $A = A_1||\ldots||A_m||A_*$. Here, A_* is the last (possibly empty) partial block.

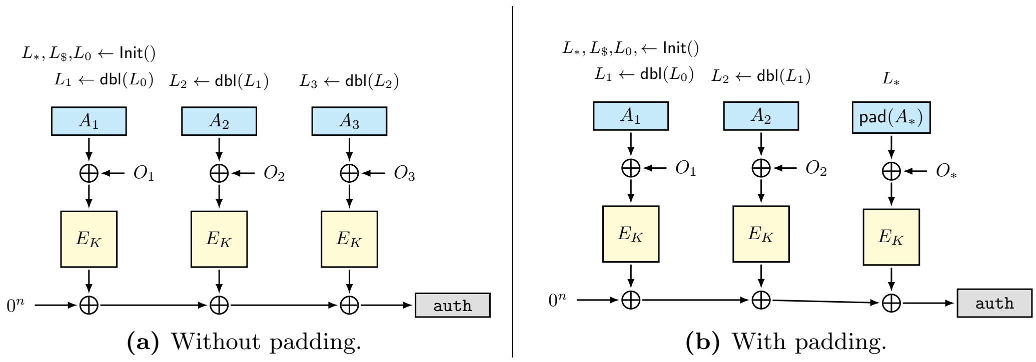
First, we process the full blocks. Let $S_0 = 0$, $O_0 = 0$ and compute for $1 \leq i \leq m$:

$$\begin{aligned} O_i &= O_{i-1} \oplus L_{\mathtt{ntz}(i)}, \\ S_i &= S_{i-1} \oplus E_K(A_i \oplus O_i). \end{aligned} \tag{3}$$

If the length of the partial block A_* is nonzero, we further compute

$$\begin{aligned} O_* &= O_m \oplus L_*, \\ \mathtt{auth} &= S_m \oplus E_K((A_*||1||0^{127-|A_*|}) \oplus O_*), \end{aligned} \tag{4}$$

otherwise we set $\mathtt{auth} = S_m$.



(a) Without padding. **(b)** With padding.

Figure 1: Processing of the associated data in OCB.

Encryption Part. It takes as input the key K, the nonce N, associated data A, plaintext M and produces a ciphertext C and tag $\texttt{tag}$ (see Figure 2). We consider M as a sequence of 128-bit blocks. Let l be the largest integer such that $128l \leq |M|$, then $M = M_1||\ldots||M_m||M_*$. Here, M_* is the last (possibly empty) partial block.

In order to derive O_0 and $\texttt{sum}_0$, we compute

$$
\begin{aligned}
\texttt{nonce} &= \tau||0^{120-|N|}||1||N, \\
\texttt{bottom} &= \texttt{nonce}[123..128], \\
\texttt{Ktop} &= E_K(\texttt{nonce}[1..122]||0^6), \\
\texttt{Stretch} &= \texttt{Ktop}||(\texttt{Ktop}[1..64] \oplus \texttt{Ktop}[9..72]), \\
O_0 &= \texttt{Stretch}[(1+\texttt{bottom})..(128+\texttt{bottom})], \\
\texttt{sum}_0 &= 0^{128}.
\end{aligned} \tag{5}
$$

We then process the full message blocks in the following way for $1 \leq i \leq m$:

$$
\begin{aligned}
O_i &= O_{i-1} \oplus L_{\texttt{ntz}(i)}, \\
C_i &= O_i \oplus E_K(M_i \oplus O_i), \\
\texttt{sum}_i &= \texttt{sum}_{i-1} \oplus M_i.
\end{aligned} \tag{6}
$$

If the length of the partial block P_* nonzero, then we further compute

$$
\begin{aligned}
O_* &= O_m \oplus L_* \\
\texttt{Pad} &= E_K(O_*) \\
C_* &= M_* \oplus \texttt{Pad}[1..|M_*|] \\
\texttt{sum}_* &= \texttt{sum}_m \oplus (M_*||1||0^{127-|M_*|}) \\
O_\$ &= O_* \oplus L_\$ \\
\texttt{tag} &= E_K(\texttt{sum}_* \oplus O_\$) \oplus \texttt{HASH}(K, A),
\end{aligned} \tag{7}
$$

otherwise

$$
\begin{aligned}
O_\$ &= O_m \oplus L_\$ \\
\texttt{tag} &= E_K(\texttt{sum}_m \oplus O_\$) \oplus \texttt{HASH}(K, A).
\end{aligned} \tag{8}
$$

The ciphertext is given by $C = C_1||\ldots||C_m||C_*$.

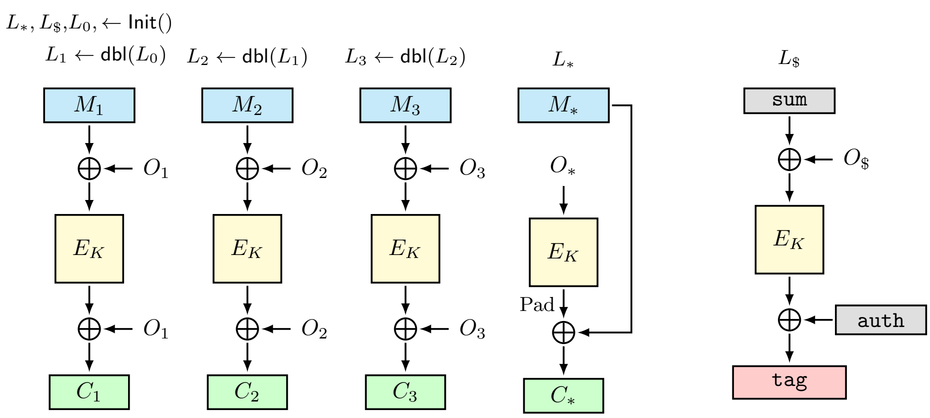


Figure 2: Encryption part of $\texttt{OCB}$.

Verification and Decryption Part. Takes as input the key K, the nonce N, associated data A, ciphertext C and produces a plaintext M. We consider C as a sequence of 128-bit blocks. Let l be the largest integer such that $128l \leq |C|$, then $C = C_1||\ldots||C_m||C_*$. Here, C_* is the last (possibly empty) partial block.

First, we have to derive O_0 and $\texttt{sum}_0$ in the exact same way as given in Equation 5. We then process the full blocks for $1 \leq i \leq m$ as:

$$\begin{aligned} O_i &= O_{i-1} \oplus L_{\texttt{ntz}(i)}, \\ P_i &= O_i \oplus D_K(C_i \oplus O_i), \\ \texttt{sum}_i &= \texttt{sum}_{i-1} \oplus M_i. \end{aligned} \tag{9}$$

If the length of the partial block C_* is nonzero, then we further compute

$$\begin{aligned} O_* &= O_m \oplus L_*, \\ \texttt{Pad} &= E_K(O_*), \\ M_* &= C_* \oplus \texttt{Pad}[1..|C_*|], \\ \texttt{sum}_* &= \texttt{sum}_m \oplus (M_*||1||0^{127-|M_*|}), \\ \texttt{tag}' &= E_K(\texttt{sum}_* \oplus O_* \oplus L_\$) \oplus \texttt{HASH}(K, A), \end{aligned} \tag{10}$$

otherwise

$$\texttt{tag}' = E_K(\texttt{sum}_m \oplus O_m \oplus L_\$) \oplus \texttt{HASH}(K, A). \tag{11}$$

If $\texttt{tag}' = \texttt{tag}$, then output $M = M_1||\ldots||M_m||M_*$ and the authentication succeeds, otherwise output $\perp$ to indicate failure.

2.3.2 `Pyjamask-96-AEAD`

The original `OCB` mode has been designed for 128-bit block ciphers. Consequently, we use it as described in the previous section for `Pyjamask-128`. However, we have made some slight modifications to handle our 96-bit block cipher describe in the next section: the most important changes are reported below.

Finite Field Arithmetic. For the multiplication in $\texttt{GF}(2^{96})$, we define the irreducible polynomial to be $\texttt{x}^{96} + \texttt{x}^{10} + \texttt{x}^9 + \texttt{x}^6 + 1$.

Therefore, the doubling operation on 96-bit dbl_{96} is defined as

$$\mathsf{dbl}_{96}(x) = \begin{cases} x \ll 1 & \text{if } \mathrm{msb}(x) = 0 \\ (x \ll 1) \oplus \texttt{0x641} & \text{otherwise.} \end{cases}$$

Stretch-then-shift Hash Function. The parameter of the stretch-then-shift hash function (that computes `Stretch` in Equation 5) is modified. In particular, the left-shift value is changed from $c = 8$ to $c = 9$. In other words, the stretch-then-shift hash function is defined as

$$\texttt{Stretch}_{96} = \texttt{Ktop}||(\texttt{Ktop}[1..64] \oplus \texttt{Ktop}[10..73]).$$

2.4 The Block Cipher Family

The block cipher family `Pyjamask` used in this submission 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 3 and detailed hereafter. Our cipher share some similarities with existing ciphers, such as `NOEKEON` [11]

Table 3: Parameters of `Pyjamask` block ciphers. All the sizes are in bits.

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

(for its general strucure), `LowMC` [13] (for the different linear layers on each slice) or even `LowMC` [2] (for its general AND gate minimisation).

The ciphers rely on a Substitution-Permutation Network (SPN) 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.

2.4.1 Data Representation

The plaintext is initially loaded into the internal states of the ciphers (see Figure 3) which are viewed as matrices 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 3: Internal state of `Pyjamask-128` with $r = 4$ words 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 word while the cell of greatest index holds the least significant bit of the word. 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`).

2.4.2 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 4):

- AddRoundKey – Bitwise addition of the first n bits of the key state (define below) into 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 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.

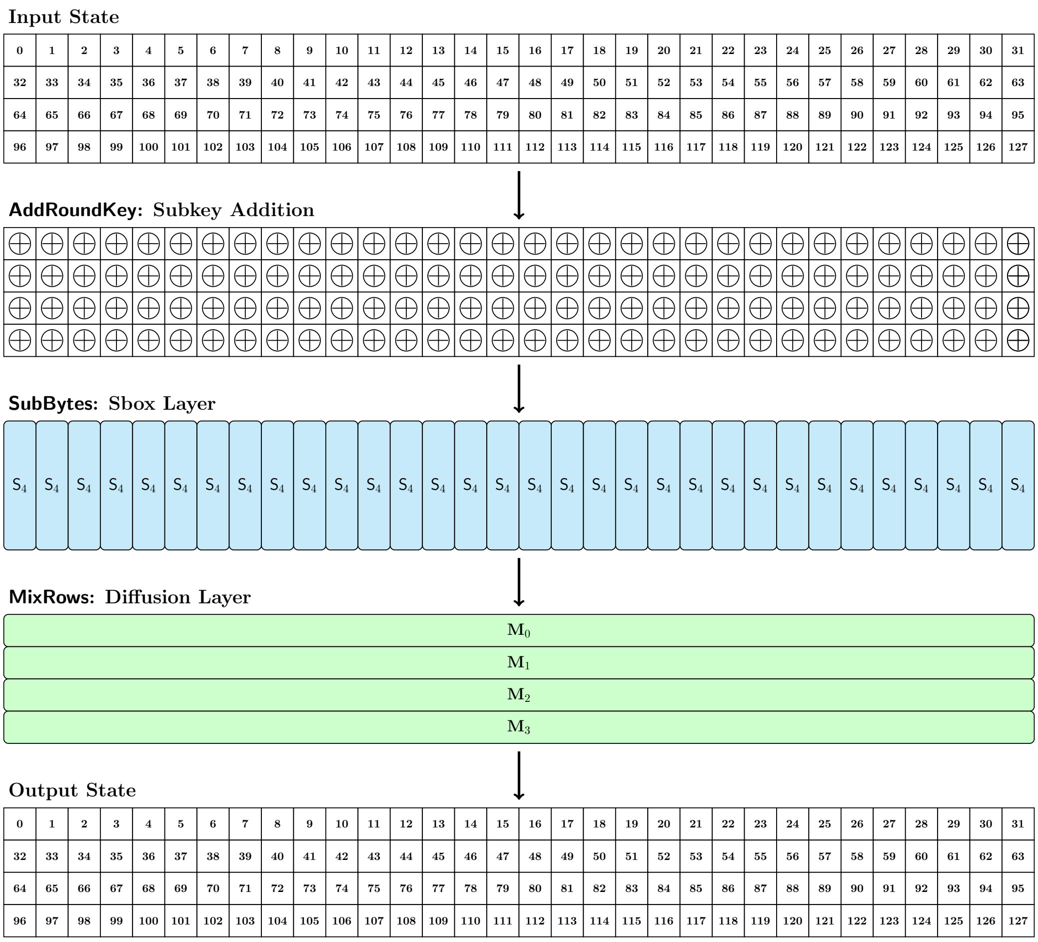


Figure 4: Round function of `Pyjamask-128`.

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

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

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

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

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 4.

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`. In Appendix, we also give the same matrices in regular form.

2.4.3 Inverse Round Function

As the decryption functionality of some mode of operation requires the decryption primitive of the 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:

- invAddRoundKey – Bitwise addition of the first n bits of the key state into the internal state.
- invMixRows – Each row R_i of the 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.

Again, after the last inverse round, a last subkey is XORed to the internal state.The inverse matrices and Sboxes used in `Pyjamask-96` and `Pyjamask-128` are given in Appendix.

2.4.4 Key Schedule

The two ciphers `Pyjamask-96` and `Pyjamask-128` shares 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 3). Then, the 128-bit key state undergoes three elementary transformations (see Figure 5):

- 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:

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

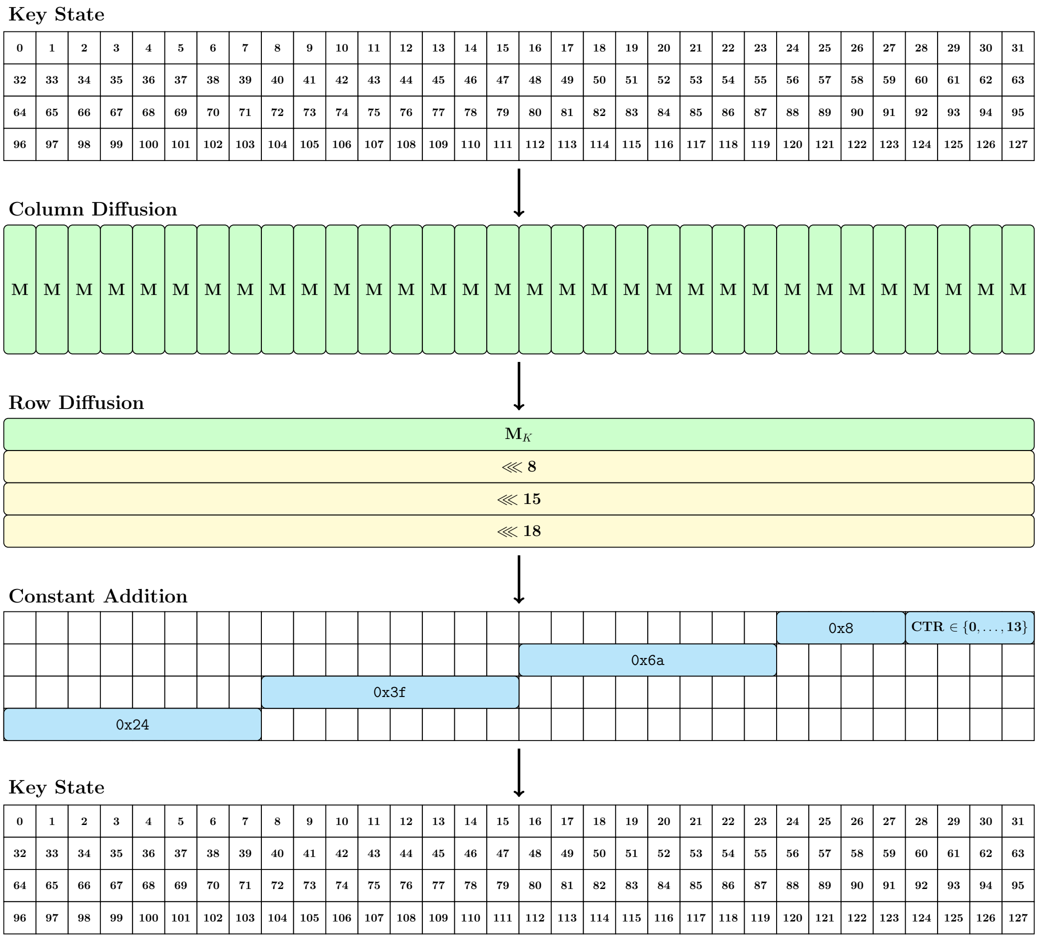


Figure 5: Key schedule of `Pyjamask-96` and `Pyjamask-128`.

- 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 of 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 5 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.

2.4.5 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` encryption of `plaintext` under `key` proceeds as follows:

Encryption:

```
state ← Load(plaintext)
roundkey[0 : 14] ← KeySchedule(key)
for i = 0 to 13 do
    state ← AddRoundKey(state, roundkey[i])
    state ← SubBytes(state)
    state ← MixRows(state)
end for
state ← AddRoundKey(state, roundkey[14])
ciphertext ← Unload(state)
return ciphertext
```

The `Pyjamask` decryption of `ciphertext` under `key` proceeds as follows:

Decryption:

```
state ← Load(ciphertext)
roundkey[0 : 14] ← KeySchedule(key)
state ← AddRoundKey(state, roundkey[14])
for i = 13 downto 0 do
    state ← InvMixRows(state)
    state ← InvSubBytes(state)
    state ← AddRoundKey(state, roundkey[i])
end for
plaintext ← Unload(state)
return plaintext
```

In the following pseudo-code, we denote by MixColumns, MixAndRotateRows and AddConstant the key schedule transformations as defined above. The `Pyjamask` key schedule expand `key` into `roundkey`$[0:14]$ as follows:

Key schedule:

```
keystate ← Load(key)
roundkey[0] ← keystate
for i = 1 to 14 do
    keystate ← MixColumns(keystate)
    keystate ← MixAndRotateRows(keystate)
    keystate ← AddConstant(keystate, i)
    roundkey[i] ← keystate
end for
return roundkey[0 : 14]
```

3 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.

In the presence of masking, each variable in the computation are split into d *shares*, which are bound to the original variable through completeness relation, and which satisfy some randomness property to wipe out the side-channel information leakage. Under some realistic assumptions, the number of shares, or the *masking order* $d - 1$, has indeed been argued to be a sound security parameter for the masked implementation [9, 14, 30]. In the masking world, the evaluation of a nonlinear operation has a complexity $O(d^2)$ while for a linear operation the complexity is of $O(d)$ (the linearity being with respect to the sharing operation, which is usually the bitwise addition). When a masking of high order is involved, most of the computation is hence dedicated to the masked nonlinear operations and the linear layers are *virtually free*. Several works have recently shown that the best performances for high-order masked implementations are obtained through the use of *bitslicing* [17, 18, 19, 21, 23, 24]. In such implementations, the nonlinear layers are performed through ℓ-bitwise AND operations (ℓ-AND), where ℓ is the size of the underlying architecture (e.g., ℓ equals 8, 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 SPN architecture (Sbox layer, linear diffusion layer, and 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, i.e., m AND gates of 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 32×32 binary matrices. Different matrices are used for the different 32-bit slices in order to avoid too much regularity. 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.

3.1 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 [32]. In addition, we note $\mathtt{x} = 2$ is a primitive element.

Stretch-then-shift Hash Function. In [26], the empirical result shows that the hash function $H : \{0,1\}^{128} \times [0..63] \to \{0,1\}^{128}$ defined by

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

where $Stretch = K \| (K \oplus (K \lll c))$ 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 [26] for our 96-bit hash function $H_{96} : \{0,1\}^{96} \times [0..63] \to \{0,1\}^{96}$ defined by

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

where $Stretch = K \| (K \oplus (K \lll c))$. 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 microcontrollers, when one can only shift by 1, therefore any multiple of 8 ± 1 would be preferred [4].

3.1.1 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 are reminiscent of the design of the `PICCOLO` and the `SKINNY` Sboxes. 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 current Sboxes S_3 and S_4, The DDT and LAT of S_3 are presented in Table 4 and Table 5, 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 6 and Table 7. In all these four tables, rows (resp. columns) represent input (resp. output) differences or masks.

Table 4: DDT 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

Table 5: LAT of S_3.

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

3.1.2 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 Section 2.4). 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 branching number [1]. Looking at the best known linear codes of these dimensions, one knows that the best theoretically achievable minimum distance is 16 [7, 20].However, one does know

Table 6: 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 7: 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

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 additions required to evaluate the matrix multiplication as done in a recent series of academic papers, e.g., [15, 22, 25]. More specifically, for each candidate matrix, we have run Paar1 algorithm [28], 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: the resulting matrices are given fully in Appendix.

3.1.3 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` [3]. 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 [4], 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.

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 emphasize that not all the possible attack vectors have been deeply investigated.

4.1 Differential Analysis

We give in Table 10 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 [33]. 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 10, 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.

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

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 8.

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).

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 9.

Table 8: 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		

Table 9: 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	0000000000000001180040c9000040c8	000000000000000000000000180040c9	7
3	0000000000000000000000000114a000	0114a000011480000114200000000000	5
4	e6e2431674f49dd216e2eb1900000000	c684f6152430b9cec4b29804b6c6eac3	27
5	041000c802100180060000c8061000c8	061001c800100180061001c804100148	7
6	7d31d40c9f26e70a5b4dcd134fa24e25		

Table 10: 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	$\in \{18, 19\}$	≥ 20

4.2 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 [6].

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	59	91	109	118	123	125	126	127	127	127	127

4.3 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. We found that `Pyjamask-96` allows the invariant subspace cryptanalysis if we relax the following two operations during the key schedule: the first one is the complete omission of MixColumns, and the second one is modifying the AddConstant so that all the constant nibbles are added to the first row. Here, we describe the attack which implies the rationale of those design choices.

Attack Procedure. We set up the plaintext and the weak key as follows.

- Let $\mathcal{S}^{32}$ be the set of 2^{32} state values in which the first row can take any value, the second row is fixed to `0xffffffff` and the third row is fixed to `0x00000000`.
- Let $\mathcal{K}^{32}$ be the set of 2^{32} key state values in which the first row can take any value and the second, third and forth rows are fixed to `0x00000000`.

The attacker chooses any plaintext from $\mathcal{S}^{32}$ and suppose that the key is chosen from $\mathcal{K}^{32}$. Then, the corresponding ciphertext is in $\mathcal{S}^{32}$ with probability 1.

Analysis. Due to the omission of the MixColumns and the modification of the constant, the second, third, and forth rows of the key state will not change through the key schedule. Hence, all subkeys will be in the subspace of $\mathcal{K}^{32}$. Note that the value of the first row changes but this does not help to escape from the invariant subspace $\mathcal{K}^{32}$.

The plaintext is chosen from $\mathcal{S}^{32}$. Even after adding any key from $\mathcal{K}^{32}$, the state value stays in the subspace $\mathcal{S}^{32}$. During SubBytes, each 3-bit input to the Sbox is now either `0x2` or `0x6` depending on the value for the first row. Here the Sbox of `Pyjamask-96` has the property such that $\mathsf{S}_3(2) = 6$ and $\mathsf{S}_3(6) = 2$, i.e. the subspace $\mathcal{S}$ is invariant by the Sbox. Hence even after SubBytes the state value stays in $\mathcal{S}^{32}$. During MixRows, the input value to the second row `0xffffffff` is mapped to `0xffffffff` by a multiplication of the circulant matrix with an odd weight. The input value to the third row `0x00000000` is mapped to `0x00000000` by any linear operation. Hence even after MixRows the state value stays in $\mathcal{S}^{32}$. In the end, the ciphertext is in the subspace of $\mathcal{S}^{32}$.

To describe the exploited invariant subspace generally, the 3-bit Sbox S_3 maps the affine space $\mathcal{A} :< \texttt{0x4} > + \texttt{0x2}$ to $\mathcal{A}$, where $< v >$ is the vector space spanned by v, and MixRows preserves the space as long as all the Sbox outputs are in $\mathcal{A}$. We then chose the plaintext so that each input to the Sbox is in $\mathcal{A}$ and chose the key such that all columns for all subkeys are in $< \texttt{0x4} >$.

Note that `Pyjamask-96` prevents this attack first by using MixColumns in the key schedule, and second by adding constants to all the rows.

5 Implementations and Performances

5.1 Software

5.1.1 Bitslice Implementation

Bitslice is an implementation strategy initially proposed by Biham in [5]. 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 [5]. It can also be applied to speed-up the encryption of a single block with parallel evaluations of the Sboxes [21]. For standard SPN block ciphers, 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 [19].

Similarly to `NOEKEON` [10] and LS-designs [21], `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`):

$R_0 \leftarrow R_0 \oplus R_1$
$R_1 \leftarrow R_1 \oplus R_2$
$R_2 \leftarrow R_2 \oplus (R_0 \wedge R_1)$
$R_0 \leftarrow R_0 \oplus (R_1 \wedge R_2)$
$R_1 \leftarrow R_1 \oplus (R_0 \wedge R_2)$
$R_2 \leftarrow R_2 \oplus R_0$
$R_0 \leftarrow R_0 \oplus R_1$
$R_2 \leftarrow \mathsf{not}(R_2)$
$\mathsf{swap}(R_0, R_1)$

SubBytes (`Pyjamask-128`):

$R_0 \leftarrow R_0 \oplus R_3$
$R_3 \leftarrow R_3 \oplus (R_0 \wedge R_1)$
$R_0 \leftarrow R_0 \oplus (R_1 \wedge R_2)$
$R_1 \leftarrow R_1 \oplus (R_2 \wedge R_3)$
$R_2 \leftarrow R_2 \oplus (R_0 \wedge R_3)$
$R_2 \leftarrow R_2 \oplus R_1$
$R_1 \leftarrow R_1 \oplus R_0$
$R_3 \leftarrow \mathsf{not}(R_3)$
$\mathsf{swap}(R_2, R_3)$

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.

5.1.2 Masked Implementation

In a masked implementation of `Pyjamask`, the state is split into d shares $\texttt{state}[0], \ldots, \texttt{state}[d-1]$. 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 sharewisely. Namely, the MixRows operation is performed as

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

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

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

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

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

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 sharewise 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

$$C \leftarrow C \oplus (A \wedge B) \ , \tag{12}$$

on the shares of A, B and C. From the input shares $(A_1, \ldots, A_d)$, $(B_1, \ldots, B_d)$, and $(C_1, \ldots, C_d)$, such an ISW MACC proceeds as follows:

ISW MACC:

```
for i = 1 to d do
   C_i ← C_i ⊕ (A_i ∧ B_i)
   for j = i + 1 to d do
      R ← $
      C_i ← C_i ⊕ R
      C_j ← C_j ⊕ ((A_i ∧ B_j) ⊕ R)
      C_j ← C_j ⊕ (A_j ∧ B_i)
   end for
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

$$\bigoplus\nolimits_{j=1}^{d} C_j^{(out)} = \bigoplus\nolimits_{j=1}^{d} C_j^{(in)} \oplus \Big(\bigoplus\nolimits_{j=1}^{d} A_j\Big) \wedge \Big(\bigoplus\nolimits_{j=1}^{d} B_j\Big) = C \oplus (A \wedge B) \ .$$

For high-order masking, where d is up to several dozens, the ISW MACC is the most time-consuming operation 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 and memory accesses to read/write the input shares and intermediate variables. 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.

5.1.3 Performances

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 12 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.

Table 12: 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

5.1.4 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 [34], 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 [19], 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 13 and compared to our implementations of `Pyjamask-128`. We note that the `AES` implementation of [19] 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 [23] 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 implementations

Table 13: Performance comparison on ARM Cortex-M4.

	Variant	TRNG	$d = 4$	$d = 8$	$d = 16$	$d = 32$	$d = 64$	$d = 128$
			Timings (kilocycles)					
`AES-128` [19]	-	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` [19]	-	-	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` [19]	-	-	7532	7532	7532	7532	7532	7532
`Pyjamask-128`	v1	-	4070	5776	5686	3158	3198	3198
	v2	-	5696	7418	7306	4778	4818	4818

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 14 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.

5.1.5 Source Code

The software source code of Pyjamask block cipher is available at https://github.com/pyjamask-cipher.

5.2 Hardware

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[7] on UMC 180 for example using `MAOI1` gates, compared

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

Table 14: Comparision of the bitwise multiplicative complexity of several ciphers.

	key size	**# rounds**	# `AND`	# `32-AND`	# `64-AND`
96-bit block ciphers					
`SIMON-96/96`	96	52	4992	104*	52*
`Pyjamask-96`	128	14	1344	**42**	42*
`SIMON-96/144`	144	54	5184	108*	54*
128-bit block ciphers					
`LowMC-128` ($m = 3$)	128	88	792	88*	88*
`AES-128`	128	10	5120	160	100*
`SIMON-128/128`	128	68	4352	136	68
`NOEKEON`	128	16	2048	64	32
`Robin`	128	16	3072	96	96*
`Fantomas`	128	12	2304	72	72*
`Mysterion-128`	128	12	1536	**48**	**24**
`Pyjamask-128`	128	14	1792	**56**	**28**

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

with 1 GE of a NAND gate), this will have a negative impact on the area of ASIC implementations.

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

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.

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 $32\times(4\cdot1.33+7\cdot2.67) = 768$ 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 algortihm [29], we have evaluated that they can all be implemented using at most 347 XOR gates. This amounts to $5 \times (347 \cdot 2.67) = 4632$ GE.

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 7500 GE (and 14 cycles), which remains much better than an `AES` round-based implementation [31]. We emphasize that this is only a very rough estimation, we will provide real synthesis number in the future.

A possible better tradeoff 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.

6 Test Vectors

6.1 Test Vectors for the Block Ciphers

```
/* Pyjamask-96 */
Key:        00 11 22 33 44 55 66 77 88 99 aa bb cc dd ee ff
Plaintext:  50 79 6a 61 6d 61 73 6b 39 36 3a 29
Ciphertext: ca 9c 6e 1a bb de 4e dc 27 07 3d a6

/* Pyjamask-128 */
Key:        00 11 22 33 44 55 66 77 88 99 aa bb cc dd ee ff
Plaintext:  50 79 6a 61 6d 61 73 6b 2d 31 32 38 3a 29 3a 29
Ciphertext: 48 f1 39 a1 09 bd d9 c0 72 6e 82 61 f8 d6 8e 7d
```

6.2 Test Vectors for the AEAD Schemes

```
/* Pyjamask-128-AEAD */
Key:        00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f
Nonce:      00 01 02 03 04 05 06 07 08 09 0a 0b
Data:       00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f
Plaintext:  00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f

Ciphertext: 08 e4 7f e9 7b d3 76 13 ab 9a 32 2e a2 b2 51 55
Tag:        9f c7 ec 33 3c 01 54 d9 ec 57 2b d6 18 62 24 2b

/* Pyjamask-96-AEAD */
Key:        00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f
Nonce:      00 01 02 03 04 05 06 07
Data:       00 01 02 03 04 05 06 07 08 09 0a 0b
Plaintext:  00 01 02 03 04 05 06 07 08 09 0a 0b

Ciphertext: 91 80 1a be 9a a4 15 62 d3 4d 07 b3
Tag:        ee 0c b7 7d a2 92 43 2b 87 a2 6e bf
```

7 Intellectual Property

`Pyjamask` is not patented and is free for use in any application. We note `Pyjamask` uses the mode `OCB` which, to the best of our knowledge, has "United States Patent No. 7,949,129" and "United States Patent No. 8,321,675". Despite that, the inventor has stated that anyone is

- authorized to make, use, and distribute open-source software implementations of `OCB`,
- (aside from military uses) authorized to make, use, and distribute (1) any software implementation of `OCB` and (2) non-software implementations of `OCB` for noncommercial or research purposes, and
- authorize use of `OCB` in OpenSSL.

References

1. : Advanced Encryption Standard (AES). National Institute of Standards and Technology (NIST), FIPS PUB 197, U.S. Department of Commerce (November 2001)
2. Albrecht, M.R., Rechberger, C., Schneider, T., Tiessen, T., Zohner, M.: Ciphers for MPC and FHE. In Oswald, E., Fischlin, M., eds.: Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part I. Volume 9056 of Lecture Notes in Computer Science., Springer (2015) 430–454
3. Banik, S., Bogdanov, A., Isobe, T., Shibutani, K., Hiwatari, H., Akishita, T., Regazzoni, F.: Midori: A block cipher for low energy. In Iwata, T., Cheon, J.H., eds.: 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., Springer (2015) 411–436
4. Beaulieu, R., Shors, D., Smith, J., Treatman-Clark, S., Weeks, B., Wingers, L.: Notes on the design and analysis of SIMON and SPECK. Cryptology ePrint Archive, Report 2017/560 (2017) https://eprint.iacr.org/2017/560.
5. Biham, E.: A fast new DES implementation in software. In Biham, E., ed.: FSE'97. Volume 1267 of LNCS., Springer, Heidelberg (January 1997) 260–272
6. Boura, C., Canteaut, A., Cannière, C.D.: Higher-order differential properties of keccak and *Luffa*. In: FSE. Volume 6733 of Lecture Notes in Computer Science., Springer (2011) 252–269
7. Brouwer, A.E.: Bounds on the size of linear codes. In Pless, V.S., Huffman, W., eds.: Handbook of Coding Theory. Elsevier, Amsterdam (1998) 295–461
8. Chakraborti, A., Iwata, T., Minematsu, K., Nandi, M.: Blockcipher-based authenticated encryption: How small can we go? [16] 277–298
9. Chari, S., Jutla, C.S., Rao, J.R., Rohatgi, P.: Towards sound approaches to counteract power-analysis attacks. In Wiener, M.J., ed.: 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., Springer (1999) 398–412
10. Daemen, J., Peeters, M., añd Vincent Rijmen, G.V.A.: Nessie Proposal: the Block Cipher Noekeon. Nessie submission (2000) http://gro.noekeon.org/.
11. Daemen, J., Peeters, M., Assche, G.V., Rijmen, V.: Nessie Proposal: the Block Cipher Noekeon. Nessie submission (2000) http://gro.noekeon.org/.
12. Dobraunig, C., Eichlseder, M., Grassi, L., Lallemand, V., Leander, G., List, E., Mendel, F., Rechberger, C.: Rasta: A cipher with low anddepth and few ands per bit. In Shacham, H., Boldyreva, A., eds.: Advances in Cryptology - CRYPTO 2018 - 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2018, Proceedings, Part I. Volume 10991 of Lecture Notes in Computer Science., Springer (2018) 662–692
13. Dobraunig, C., Eichlseder, M., Mendel, F., Schläffer, M.: Ascon v1.2. Submission to the CAESAR competition: http://competitions.cr.yp.to/round3/asconv12.pdf (2016)
14. Duc, A., Dziembowski, S., Faust, S.: Unifying leakage models: From probing attacks to noisy leakage. In Nguyen, P.Q., Oswald, E., eds.: 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., Springer (2014) 423–440
15. Duval, S., Leurent, G.: Mds matrices with lightweight circuits. IACR Transactions on Symmetric Cryptology **2018**(2) (Jun. 2018) 48–78
16. Fischer, W., Homma, N., eds.: CHES 2017. In Fischer, W., Homma, N., eds.: CHES 2017. Volume 10529 of LNCS., Springer, Heidelberg (September 2017)
17. Goudarzi, D., Journault, A., Rivain, M., Standaert, F.X.: Secure multiplication for bitslice higher-order masking: Optimisation and comparison. In Fan, J., Gierlichs, B., eds.: COSADE 2018. Volume 10815 of LNCS., Springer, Heidelberg (April 2018) 3–22
18. Goudarzi, D., Rivain, M.: On the multiplicative complexity of boolean functions and bitsliced higher-order masking. In Gierlichs, B., Poschmann, A.Y., eds.: CHES 2016. Volume 9813 of LNCS., Springer, Heidelberg (August 2016) 457–478

19. Goudarzi, D., Rivain, M.: How fast can higher-order masking be in software? In Coron, J., Nielsen, J.B., eds.: EUROCRYPT 2017, Part I. Volume 10210 of LNCS., Springer, Heidelberg (April / May 2017) 567–597
20. Grassl, M.: Bounds on the minimum distance of linear codes and quantum codes. Online available at http://www.codetables.de (2007) Accessed on 2019-02-19.
21. Grosso, V., Leurent, G., Standaert, F.X., Varici, K.: LS-designs: Bitslice encryption for efficient masked software implementations. In Cid, C., Rechberger, C., eds.: FSE 2014. Volume 8540 of LNCS., Springer, Heidelberg (March 2015) 18–37
22. Jean, J., Peyrin, T., Sim, S., Tourteaux, J.: Optimizing implementations of lightweight building blocks. IACR Transactions on Symmetric Cryptology **2017**(4) (Dec. 2017) 130–168
23. Journault, A., Standaert, F.X.: Very high order masking: Efficient implementation and security evaluation. [16] 623–643
24. Journault, A., Standaert, F., Varici, K.: Improving the security and efficiency of block ciphers based on ls-designs. Des. Codes Cryptography **82**(1-2) (2017) 495–509
25. Kranz, T., Leander, G., Stoffelen, K., Wiemer, F.: Shorter linear straight-line programs for mds matrices. IACR Transactions on Symmetric Cryptology **2017**(4) (Dec. 2017) 188–211
26. Krovetz, T., Rogaway, P.: 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. (2011) 306–327
27. Krovetz, T., Rogaway, P.: The OCB authenticated-encryption algorithm. RFC **7253** (2014) 1–19
28. Paar, C.: Optimized arithmetic for reed-solomon encoders. In: Proceedings of IEEE International Symposium on Information Theory. (June 1997) 250–
29. Paar, C.: Optimized arithmetic for Reed-Solomon encoders. In: Proceedings of IEEE International Symposium on Information Theory, IEEE (1997) 250
30. Prouff, E., Rivain, M.: Masking against side-channel attacks: A formal security proof. In Johansson, T., Nguyen, P.Q., eds.: EUROCRYPT 2013. Volume 7881 of LNCS., Springer, Heidelberg (May 2013) 142–159
31. Satoh, A., Morioka, S., Takano, K., Munetoh, S.: A Compact Rijndael Hardware Architecture with S-Box Optimization. In Boyd, C., ed.: 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., Springer (2001) 239–254
32. Seroussi, G.: Table of low-weight binary irreducible polynomials. In: HP Labs Technical Reports. (1998) 98–135
33. Stefan Kölbl: CryptoSMT: An easy to use tool for cryptanalysis of symmetric primitives https://github.com/kste/cryptosmt.
34. Wang, J., Vadnala, P.K., Großschädl, J., Xu, Q.: Higher-order masking in practice: A vector implementation of masked AES for ARM NEON. In Nyberg, K., ed.: CT-RSA 2015. Volume 9048 of LNCS., Springer, Heidelberg (April 2015) 181–198

A Elementary Components Used in `Pyjamask`

A.1 Sboxes

We give here the inverse of the Sboxes used in `Pyjamask-96` and `Pyjamask-128`:

$$\mathsf{S}_3^{-1} = [\mathtt{7}, \mathtt{0}, \mathtt{4}, \mathtt{1}, \mathtt{5}, \mathtt{3}, \mathtt{2}, \mathtt{6}],$$
$$\mathsf{S}_4^{-1} = [\mathtt{0x9}, \mathtt{0xe}, \mathtt{0x0}, \mathtt{0x2}, \mathtt{0xb}, \mathtt{0xd}, \mathtt{0x7}, \mathtt{0x4}, \mathtt{0xc}, \mathtt{0x3}, \mathtt{0x6}, \mathtt{0x5}, \mathtt{0xf}, \mathtt{0x1}, \mathtt{0x8}, \mathtt{0xa}].$$

A.2 Diffusion Matrices

$$M_0 = \begin{bmatrix}
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\\
0&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\\
1&0&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\\
0&1&0&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&0&1&0&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&0&1&0&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\\
1&0&0&0&1&0&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&1&0&0&0&1&0&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&1&0&0&0&1&0&1&1&0&1&0&0&0&0&1&0&0&0&0&1&0&0&0&0&1&1&0&0&0&0\\
0&1&1&1&0&0&0&1&0&1&1&0&1&0&0&0&0&1&0&0&0&0&1&0&0&0&0&1&1&0&0&0\\
0&0&1&1&1&0&0&0&1&0&1&1&0&1&0&0&0&0&1&0&0&0&0&1&0&0&0&0&1&1&0&0\\
0&0&0&1&1&1&0&0&0&1&0&1&1&0&1&0&0&0&0&1&0&0&0&0&1&0&0&0&0&1&1&0\\
0&0&0&0&1&1&1&0&0&0&1&0&1&1&0&1&0&0&0&0&1&0&0&0&0&1&0&0&0&0&1&1\\
1&0&0&0&0&1&1&1&0&0&0&1&0&1&1&0&1&0&0&0&0&1&0&0&0&0&1&0&0&0&0&1\\
1&1&0&0&0&0&1&1&1&0&0&0&1&0&1&1&0&1&0&0&0&0&1&0&0&0&0&1&0&0&0&0\\
0&1&1&0&0&0&0&1&1&1&0&0&0&1&0&1&1&0&1&0&0&0&0&1&0&0&0&0&1&0&0&0\\
0&0&1&1&0&0&0&0&1&1&1&0&0&0&1&0&1&1&0&1&0&0&0&0&1&0&0&0&0&1&0&0\\
0&0&0&1&1&0&0&0&0&1&1&1&0&0&0&1&0&1&1&0&1&0&0&0&0&1&0&0&0&0&1&0\\
0&0&0&0&1&1&0&0&0&0&1&1&1&0&0&0&1&0&1&1&0&1&0&0&0&0&1&0&0&0&0&1\\
1&0&0&0&0&1&1&0&0&0&0&1&1&1&0&0&0&1&0&1&1&0&1&0&0&0&0&1&0&0&0&0\\
0&1&0&0&0&0&1&1&0&0&0&0&1&1&1&0&0&0&1&0&1&1&0&1&0&0&0&0&1&0&0&0\\
0&0&1&0&0&0&0&1&1&0&0&0&0&1&1&1&0&0&0&1&0&1&1&0&1&0&0&0&0&1&0&0\\
0&0&0&1&0&0&0&0&1&1&0&0&0&0&1&1&1&0&0&0&1&0&1&1&0&1&0&0&0&0&1&0\\
0&0&0&0&1&0&0&0&0&1&1&0&0&0&0&1&1&1&0&0&0&1&0&1&1&0&1&0&0&0&0&1\\
1&0&0&0&0&1&0&0&0&0&1&1&0&0&0&0&1&1&1&0&0&0&1&0&1&1&0&1&0&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&1&1&0&1&0&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&1&1&0&1&0&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&1&1&0&1&0\\
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&1&1&0&1\\
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&1&1&0\\
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&1&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&1
\end{bmatrix}$$

$$
M_0^{-1} = \begin{bmatrix}
0&1&0&0&0&0&1&0&0&1&0&0&0&0&1&0&1&1&1&1&0&1&1&0&0&0&0&0&0&0&1&0 \\
0&0&1&0&0&0&0&1&0&0&1&0&0&0&0&1&0&1&1&1&1&0&1&1&0&0&0&0&0&0&0&1 \\
1&0&0&1&0&0&0&0&1&0&0&1&0&0&0&0&1&0&1&1&1&1&0&1&1&0&0&0&0&0&0&0 \\
0&1&0&0&1&0&0&0&0&1&0&0&1&0&0&0&0&1&0&1&1&1&1&0&1&1&0&0&0&0&0&0 \\
0&0&1&0&0&1&0&0&0&0&1&0&0&1&0&0&0&0&1&0&1&1&1&1&0&1&1&0&0&0&0&0 \\
0&0&0&1&0&0&1&0&0&0&0&1&0&0&1&0&0&0&0&1&0&1&1&1&1&0&1&1&0&0&0&0 \\
0&0&0&0&1&0&0&1&0&0&0&0&1&0&0&1&0&0&0&0&1&0&1&1&1&1&0&1&1&0&0&0 \\
0&0&0&0&0&1&0&0&1&0&0&0&0&1&0&0&1&0&0&0&0&1&0&1&1&1&1&0&1&1&0&0 \\
0&0&0&0&0&0&1&0&0&1&0&0&0&0&1&0&0&1&0&0&0&0&1&0&1&1&1&1&0&1&1&0 \\
0&0&0&0&0&0&0&1&0&0&1&0&0&0&0&1&0&0&1&0&0&0&0&1&0&1&1&1&1&0&1&1 \\
1&0&0&0&0&0&0&0&1&0&0&1&0&0&0&0&1&0&0&1&0&0&0&0&1&0&1&1&1&1&0&1 \\
1&1&0&0&0&0&0&0&0&1&0&0&1&0&0&0&0&1&0&0&1&0&0&0&0&1&0&1&1&1&1&0 \\
0&1&1&0&0&0&0&0&0&0&1&0&0&1&0&0&0&0&1&0&0&1&0&0&0&0&1&0&1&1&1&1 \\
1&0&1&1&0&0&0&0&0&0&0&1&0&0&1&0&0&0&0&1&0&0&1&0&0&0&0&1&0&1&1&1 \\
1&1&0&1&1&0&0&0&0&0&0&0&1&0&0&1&0&0&0&0&1&0&0&1&0&0&0&0&1&0&1&1 \\
1&1&1&0&1&1&0&0&0&0&0&0&0&1&0&0&1&0&0&0&0&1&0&0&1&0&0&0&0&1&0&1 \\
1&1&1&1&0&1&1&0&0&0&0&0&0&0&1&0&0&1&0&0&0&0&1&0&0&1&0&0&0&0&1&0 \\
0&1&1&1&1&0&1&1&0&0&0&0&0&0&0&1&0&0&1&0&0&0&0&1&0&0&1&0&0&0&0&1 \\
1&0&1&1&1&1&0&1&1&0&0&0&0&0&0&0&1&0&0&1&0&0&0&0&1&0&0&1&0&0&0&0 \\
0&1&0&1&1&1&1&0&1&1&0&0&0&0&0&0&0&1&0&0&1&0&0&0&0&1&0&0&1&0&0&0 \\
0&0&1&0&1&1&1&1&0&1&1&0&0&0&0&0&0&0&1&0&0&1&0&0&0&0&1&0&0&1&0&0 \\
0&0&0&1&0&1&1&1&1&0&1&1&0&0&0&0&0&0&0&1&0&0&1&0&0&0&0&1&0&0&1&0 \\
0&0&0&0&1&0&1&1&1&1&0&1&1&0&0&0&0&0&0&0&1&0&0&1&0&0&0&0&1&0&0&1 \\
1&0&0&0&0&1&0&1&1&1&1&0&1&1&0&0&0&0&0&0&0&1&0&0&1&0&0&0&0&1&0&0 \\
0&1&0&0&0&0&1&0&1&1&1&1&0&1&1&0&0&0&0&0&0&0&1&0&0&1&0&0&0&0&1&0 \\
0&0&1&0&0&0&0&1&0&1&1&1&1&0&1&1&0&0&0&0&0&0&0&1&0&0&1&0&0&0&0&1 \\
1&0&0&1&0&0&0&0&1&0&1&1&1&1&0&1&1&0&0&0&0&0&0&0&1&0&0&1&0&0&0&0 \\
0&1&0&0&1&0&0&0&0&1&0&1&1&1&1&0&1&1&0&0&0&0&0&0&0&1&0&0&1&0&0&0 \\
0&0&1&0&0&1&0&0&0&0&1&0&1&1&1&1&0&1&1&0&0&0&0&0&0&0&1&0&0&1&0&0 \\
0&0&0&1&0&0&1&0&0&0&0&1&0&1&1&1&1&0&1&1&0&0&0&0&0&0&0&1&0&0&1&0 \\
0&0&0&0&1&0&0&1&0&0&0&0&1&0&1&1&1&1&0&1&1&0&0&0&0&0&0&0&1&0&0&1 \\
1&0&0&0&0&1&0&0&1&0&0&0&0&1&0&1&1&1&1&0&1&1&0&0&0&0&0&0&0&1&0&0
\end{bmatrix}
$$

$$
M_1 = \begin{bmatrix}
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 \\
1&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&1&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 \\
0&1&1&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&0&1&1&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&0&1&1&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 \\
1&0&0&0&1&1&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&1&0&0&0&1&1&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 \\
0&1&1&0&0&0&1&1&0&1&0&0&0&0&1&0&0&0&0&0&0&1&1&1&0&1&0&0&0&0&0&1 \\
1&0&1&1&0&0&0&1&1&0&1&0&0&0&0&1&0&0&0&0&0&0&1&1&1&0&1&0&0&0&0&0 \\
0&1&0&1&1&0&0&0&1&1&0&1&0&0&0&0&1&0&0&0&0&0&0&1&1&1&0&1&0&0&0&0 \\
0&0&1&0&1&1&0&0&0&1&1&0&1&0&0&0&0&1&0&0&0&0&0&0&1&1&1&0&1&0&0&0 \\
0&0&0&1&0&1&1&0&0&0&1&1&0&1&0&0&0&0&1&0&0&0&0&0&0&1&1&1&0&1&0&0 \\
0&0&0&0&1&0&1&1&0&0&0&1&1&0&1&0&0&0&0&1&0&0&0&0&0&0&1&1&1&0&1&0 \\
0&0&0&0&0&1&0&1&1&0&0&0&1&1&0&1&0&0&0&0&1&0&0&0&0&0&0&1&1&1&0&1 \\
1&0&0&0&0&0&1&0&1&1&0&0&0&1&1&0&1&0&0&0&0&1&0&0&0&0&0&0&1&1&1&0 \\
0&1&0&0&0&0&0&1&0&1&1&0&0&0&1&1&0&1&0&0&0&0&1&0&0&0&0&0&0&1&1&1 \\
1&0&1&0&0&0&0&0&1&0&1&1&0&0&0&1&1&0&1&0&0&0&0&1&0&0&0&0&0&0&1&1 \\
1&1&0&1&0&0&0&0&0&1&0&1&1&0&0&0&1&1&0&1&0&0&0&0&1&0&0&0&0&0&0&1 \\
1&1&1&0&1&0&0&0&0&0&1&0&1&1&0&0&0&1&1&0&1&0&0&0&0&1&0&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&0&1&0&0&0&0&1&0&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&0&1&0&0&0&0&1&0&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&0&1&0&0&0&0&1&0&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&0&1&0&0&0&0&1&0&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&0&1&0&0&0&0&1&0 \\
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&0&1&0&0&0&0&1 \\
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&0&1&0&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&0&1&0&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&0&1&0&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&0&1&0 \\
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&0&1 \\
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&0
\end{bmatrix}
$$

$$M_1^{-1} = \begin{bmatrix}
0&0&0&0&0&0&1&0&1&0&1&0&0&1&1&1&1&1&1&1&1&0&0&0&1&0&0&0&0&1&0&0\\
0&0&0&0&0&0&0&1&0&1&0&1&0&0&1&1&1&1&1&1&1&1&0&0&0&1&0&0&0&0&1&0\\
0&0&0&0&0&0&0&0&1&0&1&0&1&0&0&1&1&1&1&1&1&1&1&0&0&0&1&0&0&0&0&1\\
1&0&0&0&0&0&0&0&0&1&0&1&0&1&0&0&1&1&1&1&1&1&1&1&0&0&0&1&0&0&0&0\\
0&1&0&0&0&0&0&0&0&0&1&0&1&0&1&0&0&1&1&1&1&1&1&1&1&0&0&0&1&0&0&0\\
0&0&1&0&0&0&0&0&0&0&0&1&0&1&0&1&0&0&1&1&1&1&1&1&1&1&0&0&0&1&0&0\\
0&0&0&1&0&0&0&0&0&0&0&0&1&0&1&0&1&0&0&1&1&1&1&1&1&1&1&0&0&0&1&0\\
0&0&0&0&1&0&0&0&0&0&0&0&0&1&0&1&0&1&0&0&1&1&1&1&1&1&1&1&0&0&0&1\\
1&0&0&0&0&1&0&0&0&0&0&0&0&0&1&0&1&0&1&0&0&1&1&1&1&1&1&1&1&0&0&0\\
0&1&0&0&0&0&1&0&0&0&0&0&0&0&0&1&0&1&0&1&0&0&1&1&1&1&1&1&1&1&0&0\\
0&0&1&0&0&0&0&1&0&0&0&0&0&0&0&0&1&0&1&0&1&0&0&1&1&1&1&1&1&1&1&0\\
0&0&0&1&0&0&0&0&1&0&0&0&0&0&0&0&0&1&0&1&0&1&0&0&1&1&1&1&1&1&1&1\\
1&0&0&0&1&0&0&0&0&1&0&0&0&0&0&0&0&0&1&0&1&0&1&0&0&1&1&1&1&1&1&1\\
1&1&0&0&0&1&0&0&0&0&1&0&0&0&0&0&0&0&0&1&0&1&0&1&0&0&1&1&1&1&1&1\\
1&1&1&0&0&0&1&0&0&0&0&1&0&0&0&0&0&0&0&0&1&0&1&0&1&0&0&1&1&1&1&1\\
1&1&1&1&0&0&0&1&0&0&0&0&1&0&0&0&0&0&0&0&0&1&0&1&0&1&0&0&1&1&1&1\\
1&1&1&1&1&0&0&0&1&0&0&0&0&1&0&0&0&0&0&0&0&0&1&0&1&0&1&0&0&1&1&1\\
1&1&1&1&1&1&0&0&0&1&0&0&0&0&1&0&0&0&0&0&0&0&0&1&0&1&0&1&0&0&1&1\\
1&1&1&1&1&1&1&0&0&0&1&0&0&0&0&1&0&0&0&0&0&0&0&0&1&0&1&0&1&0&0&1\\
1&1&1&1&1&1&1&1&0&0&0&1&0&0&0&0&1&0&0&0&0&0&0&0&0&1&0&1&0&1&0&0\\
0&1&1&1&1&1&1&1&1&0&0&0&1&0&0&0&0&1&0&0&0&0&0&0&0&0&1&0&1&0&1&0\\
0&0&1&1&1&1&1&1&1&1&0&0&0&1&0&0&0&0&1&0&0&0&0&0&0&0&0&1&0&1&0&1\\
1&0&0&1&1&1&1&1&1&1&1&0&0&0&1&0&0&0&0&1&0&0&0&0&0&0&0&0&1&0&1&0\\
0&1&0&0&1&1&1&1&1&1&1&1&0&0&0&1&0&0&0&0&1&0&0&0&0&0&0&0&0&1&0&1\\
1&0&1&0&0&1&1&1&1&1&1&1&1&0&0&0&1&0&0&0&0&1&0&0&0&0&0&0&0&0&1&0\\
0&1&0&1&0&0&1&1&1&1&1&1&1&1&0&0&0&1&0&0&0&0&1&0&0&0&0&0&0&0&0&1\\
1&0&1&0&1&0&0&1&1&1&1&1&1&1&1&0&0&0&1&0&0&0&0&1&0&0&0&0&0&0&0&0\\
0&1&0&1&0&1&0&0&1&1&1&1&1&1&1&1&0&0&0&1&0&0&0&0&1&0&0&0&0&0&0&0\\
0&0&1&0&1&0&1&0&0&1&1&1&1&1&1&1&1&0&0&0&1&0&0&0&0&1&0&0&0&0&0&0\\
0&0&0&1&0&1&0&1&0&0&1&1&1&1&1&1&1&1&0&0&0&1&0&0&0&0&1&0&0&0&0&0\\
0&0&0&0&1&0&1&0&1&0&0&1&1&1&1&1&1&1&1&0&0&0&1&0&0&0&0&1&0&0&0&0\\
0&0&0&0&0&1&0&1&0&1&0&0&1&1&1&1&1&1&1&1&0&0&0&1&0&0&0&0&1&0&0&0
\end{bmatrix}$$

$$M_2 = \begin{bmatrix}
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\\
1&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&1&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\\
0&1&1&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\\
1&0&1&1&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\\
0&1&0&1&1&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&0&1&0&1&1&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\\
1&0&0&1&0&1&1&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\\
0&1&0&0&1&0&1&1&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&0&1&0&0&1&0&1&1&0&0&0&0&0&0&0&0&1&0&1&0&0&1&1&1&1&0&0&1&1&0&1\\
1&0&0&1&0&0&1&0&1&1&0&0&0&0&0&0&0&0&1&0&1&0&0&1&1&1&1&0&0&1&1&0\\
0&1&0&0&1&0&0&1&0&1&1&0&0&0&0&0&0&0&0&1&0&1&0&0&1&1&1&1&0&0&1&1\\
1&0&1&0&0&1&0&0&1&0&1&1&0&0&0&0&0&0&0&0&1&0&1&0&0&1&1&1&1&0&0&1\\
1&1&0&1&0&0&1&0&0&1&0&1&1&0&0&0&0&0&0&0&0&1&0&1&0&0&1&1&1&1&0&0\\
0&1&1&0&1&0&0&1&0&0&1&0&1&1&0&0&0&0&0&0&0&0&1&0&1&0&0&1&1&1&1&0\\
0&0&1&1&0&1&0&0&1&0&0&1&0&1&1&0&0&0&0&0&0&0&0&1&0&1&0&0&1&1&1&1\\
1&0&0&1&1&0&1&0&0&1&0&0&1&0&1&1&0&0&0&0&0&0&0&0&1&0&1&0&0&1&1&1\\
1&1&0&0&1&1&0&1&0&0&1&0&0&1&0&1&1&0&0&0&0&0&0&0&0&1&0&1&0&0&1&1\\
1&1&1&0&0&1&1&0&1&0&0&1&0&0&1&0&1&1&0&0&0&0&0&0&0&0&1&0&1&0&0&1\\
1&1&1&1&0&0&1&1&0&1&0&0&1&0&0&1&0&1&1&0&0&0&0&0&0&0&0&1&0&1&0&0\\
0&1&1&1&1&0&0&1&1&0&1&0&0&1&0&0&1&0&1&1&0&0&0&0&0&0&0&0&1&0&1&0\\
0&0&1&1&1&1&0&0&1&1&0&1&0&0&1&0&0&1&0&1&1&0&0&0&0&0&0&0&0&1&0&1\\
1&0&0&1&1&1&1&0&0&1&1&0&1&0&0&1&0&0&1&0&1&1&0&0&0&0&0&0&0&0&1&0\\
0&1&0&0&1&1&1&1&0&0&1&1&0&1&0&0&1&0&0&1&0&1&1&0&0&0&0&0&0&0&0&1\\
1&0&1&0&0&1&1&1&1&0&0&1&1&0&1&0&0&1&0&0&1&0&1&1&0&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&0&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&0&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&0&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&0&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&0&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&0&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&0
\end{bmatrix}$$

$$M_2^{-1} = \begin{bmatrix}
1&0&0&0&0&0&0&1&1&0&0&0&1&1&0&1&1&0&0&1&0&1&0&1&0&0&0&0&0&1&0&0\\
0&1&0&0&0&0&0&0&1&1&0&0&0&1&1&0&1&1&0&0&1&0&1&0&1&0&0&0&0&0&1&0\\
0&0&1&0&0&0&0&0&0&1&1&0&0&0&1&1&0&1&1&0&0&1&0&1&0&1&0&0&0&0&0&1\\
1&0&0&1&0&0&0&0&0&0&1&1&0&0&0&1&1&0&1&1&0&0&1&0&1&0&1&0&0&0&0&0\\
0&1&0&0&1&0&0&0&0&0&0&1&1&0&0&0&1&1&0&1&1&0&0&1&0&1&0&1&0&0&0&0\\
0&0&1&0&0&1&0&0&0&0&0&0&1&1&0&0&0&1&1&0&1&1&0&0&1&0&1&0&1&0&0&0\\
0&0&0&1&0&0&1&0&0&0&0&0&0&1&1&0&0&0&1&1&0&1&1&0&0&1&0&1&0&1&0&0\\
0&0&0&0&1&0&0&1&0&0&0&0&0&0&1&1&0&0&0&1&1&0&1&1&0&0&1&0&1&0&1&0\\
0&0&0&0&0&1&0&0&1&0&0&0&0&0&0&1&1&0&0&0&1&1&0&1&1&0&0&1&0&1&0&1\\
1&0&0&0&0&0&1&0&0&1&0&0&0&0&0&0&1&1&0&0&0&1&1&0&1&1&0&0&1&0&1&0\\
0&1&0&0&0&0&0&1&0&0&1&0&0&0&0&0&0&1&1&0&0&0&1&1&0&1&1&0&0&1&0&1\\
1&0&1&0&0&0&0&0&1&0&0&1&0&0&0&0&0&0&1&1&0&0&0&1&1&0&1&1&0&0&1&0\\
0&1&0&1&0&0&0&0&0&1&0&0&1&0&0&0&0&0&0&1&1&0&0&0&1&1&0&1&1&0&0&1\\
1&0&1&0&1&0&0&0&0&0&1&0&0&1&0&0&0&0&0&0&1&1&0&0&0&1&1&0&1&1&0&0\\
0&1&0&1&0&1&0&0&0&0&0&1&0&0&1&0&0&0&0&0&0&1&1&0&0&0&1&1&0&1&1&0\\
0&0&1&0&1&0&1&0&0&0&0&0&1&0&0&1&0&0&0&0&0&0&1&1&0&0&0&1&1&0&1&1\\
1&0&0&1&0&1&0&1&0&0&0&0&0&1&0&0&1&0&0&0&0&0&0&1&1&0&0&0&1&1&0&1\\
1&1&0&0&1&0&1&0&1&0&0&0&0&0&1&0&0&1&0&0&0&0&0&0&1&1&0&0&0&1&1&0\\
0&1&1&0&0&1&0&1&0&1&0&0&0&0&0&1&0&0&1&0&0&0&0&0&0&1&1&0&0&0&1&1\\
1&0&1&1&0&0&1&0&1&0&1&0&0&0&0&0&1&0&0&1&0&0&0&0&0&0&1&1&0&0&0&1\\
1&1&0&1&1&0&0&1&0&1&0&1&0&0&0&0&0&1&0&0&1&0&0&0&0&0&0&1&1&0&0&0\\
0&1&1&0&1&1&0&0&1&0&1&0&1&0&0&0&0&0&1&0&0&1&0&0&0&0&0&0&1&1&0&0\\
0&0&1&1&0&1&1&0&0&1&0&1&0&1&0&0&0&0&0&1&0&0&1&0&0&0&0&0&0&1&1&0\\
0&0&0&1&1&0&1&1&0&0&1&0&1&0&1&0&0&0&0&0&1&0&0&1&0&0&0&0&0&0&1&1\\
1&0&0&0&1&1&0&1&1&0&0&1&0&1&0&1&0&0&0&0&0&1&0&0&1&0&0&0&0&0&0&1\\
1&1&0&0&0&1&1&0&1&1&0&0&1&0&1&0&1&0&0&0&0&0&1&0&0&1&0&0&0&0&0&0\\
0&1&1&0&0&0&1&1&0&1&1&0&0&1&0&1&0&1&0&0&0&0&0&1&0&0&1&0&0&0&0&0\\
0&0&1&1&0&0&0&1&1&0&1&1&0&0&1&0&1&0&1&0&0&0&0&0&1&0&0&1&0&0&0&0\\
0&0&0&1&1&0&0&0&1&1&0&1&1&0&0&1&0&1&0&1&0&0&0&0&0&1&0&0&1&0&0&0\\
0&0&0&0&1&1&0&0&0&1&1&0&1&1&0&0&1&0&1&0&1&0&0&0&0&0&1&0&0&1&0&0\\
0&0&0&0&0&1&1&0&0&0&1&1&0&1&1&0&0&1&0&1&0&1&0&0&0&0&0&1&0&0&1&0\\
0&0&0&0&0&0&1&1&0&0&0&1&1&0&1&1&0&0&1&0&1&0&1&0&0&0&0&0&1&0&0&1
\end{bmatrix}$$

$$M_3 = \begin{bmatrix}
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\\
1&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\\
0&1&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&0&1&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\\
1&0&0&1&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\\
0&1&0&0&1&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&0&1&0&0&1&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&0&1&0&0&1&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\\
1&0&0&0&1&0&0&1&0&1&1&0&0&1&0&0&0&0&0&0&1&0&0&1&0&1&0&1&0&0&1&0\\
0&1&0&0&0&1&0&0&1&0&1&1&0&0&1&0&0&0&0&0&0&1&0&0&1&0&1&0&1&0&0&1\\
1&0&1&0&0&0&1&0&0&1&0&1&1&0&0&1&0&0&0&0&0&0&1&0&0&1&0&1&0&1&0&0\\
0&1&0&1&0&0&0&1&0&0&1&0&1&1&0&0&1&0&0&0&0&0&0&1&0&0&1&0&1&0&1&0\\
0&0&1&0&1&0&0&0&1&0&0&1&0&1&1&0&0&1&0&0&0&0&0&0&1&0&0&1&0&1&0&1\\
1&0&0&1&0&1&0&0&0&1&0&0&1&0&1&1&0&0&1&0&0&0&0&0&0&1&0&0&1&0&1&0\\
0&1&0&0&1&0&1&0&0&0&1&0&0&1&0&1&1&0&0&1&0&0&0&0&0&0&1&0&0&1&0&1\\
1&0&1&0&0&1&0&1&0&0&0&1&0&0&1&0&1&1&0&0&1&0&0&0&0&0&0&1&0&0&1&0\\
0&1&0&1&0&0&1&0&1&0&0&0&1&0&0&1&0&1&1&0&0&1&0&0&0&0&0&0&1&0&0&1\\
1&0&1&0&1&0&0&1&0&1&0&0&0&1&0&0&1&0&1&1&0&0&1&0&0&0&0&0&0&1&0&0\\
0&1&0&1&0&1&0&0&1&0&1&0&0&0&1&0&0&1&0&1&1&0&0&1&0&0&0&0&0&0&1&0\\
0&0&1&0&1&0&1&0&0&1&0&1&0&0&0&1&0&0&1&0&1&1&0&0&1&0&0&0&0&0&0&1\\
1&0&0&1&0&1&0&1&0&0&1&0&1&0&0&0&1&0&0&1&0&1&1&0&0&1&0&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&0&1&1&0&0&1&0&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&0&1&1&0&0&1&0&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&0&1&1&0&0&1&0&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&0&1&1&0&0&1&0&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&0&1&1&0&0&1&0\\
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&0&1&1&0&0&1\\
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&0&1&1&0&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&0&1&1&0\\
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&0&1&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&0&1\\
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&0
\end{bmatrix}$$

$$
M_3^{-1} = \begin{bmatrix}
0&1&1&1&0&1&0&0&0&1&0&0&0&1&1&0&1&0&0&1&0&1&0&1&0&1&1&0&0&1&1&0\\
0&0&1&1&1&0&1&0&0&0&1&0&0&0&1&1&0&1&0&0&1&0&1&0&1&0&1&1&0&0&1&1\\
1&0&0&1&1&1&0&1&0&0&0&1&0&0&0&1&1&0&1&0&0&1&0&1&0&1&0&1&1&0&0&1\\
1&1&0&0&1&1&1&0&1&0&0&0&1&0&0&0&1&1&0&1&0&0&1&0&1&0&1&0&1&1&0&0\\
0&1&1&0&0&1&1&1&0&1&0&0&0&1&0&0&0&1&1&0&1&0&0&1&0&1&0&1&0&1&1&0\\
0&0&1&1&0&0&1&1&1&0&1&0&0&0&1&0&0&0&1&1&0&1&0&0&1&0&1&0&1&0&1&1\\
1&0&0&1&1&0&0&1&1&1&0&1&0&0&0&1&0&0&0&1&1&0&1&0&0&1&0&1&0&1&0&1\\
1&1&0&0&1&1&0&0&1&1&1&0&1&0&0&0&1&0&0&0&1&1&0&1&0&0&1&0&1&0&1&0\\
0&1&1&0&0&1&1&0&0&1&1&1&0&1&0&0&0&1&0&0&0&1&1&0&1&0&0&1&0&1&0&1\\
1&0&1&1&0&0&1&1&0&0&1&1&1&0&1&0&0&0&1&0&0&0&1&1&0&1&0&0&1&0&1&0\\
0&1&0&1&1&0&0&1&1&0&0&1&1&1&0&1&0&0&0&1&0&0&0&1&1&0&1&0&0&1&0&1\\
1&0&1&0&1&1&0&0&1&1&0&0&1&1&1&0&1&0&0&0&1&0&0&0&1&1&0&1&0&0&1&0\\
0&1&0&1&0&1&1&0&0&1&1&0&0&1&1&1&0&1&0&0&0&1&0&0&0&1&1&0&1&0&0&1\\
1&0&1&0&1&0&1&1&0&0&1&1&0&0&1&1&1&0&1&0&0&0&1&0&0&0&1&1&0&1&0&0\\
0&1&0&1&0&1&0&1&1&0&0&1&1&0&0&1&1&1&0&1&0&0&0&1&0&0&0&1&1&0&1&0\\
0&0&1&0&1&0&1&0&1&1&0&0&1&1&0&0&1&1&1&0&1&0&0&0&1&0&0&0&1&1&0&1\\
1&0&0&1&0&1&0&1&0&1&1&0&0&1&1&0&0&1&1&1&0&1&0&0&0&1&0&0&0&1&1&0\\
0&1&0&0&1&0&1&0&1&0&1&1&0&0&1&1&0&0&1&1&1&0&1&0&0&0&1&0&0&0&1&1\\
1&0&1&0&0&1&0&1&0&1&0&1&1&0&0&1&1&0&0&1&1&1&0&1&0&0&0&1&0&0&0&1\\
1&1&0&1&0&0&1&0&1&0&1&0&1&1&0&0&1&1&0&0&1&1&1&0&1&0&0&0&1&0&0&0\\
0&1&1&0&1&0&0&1&0&1&0&1&0&1&1&0&0&1&1&0&0&1&1&1&0&1&0&0&0&1&0&0\\
0&0&1&1&0&1&0&0&1&0&1&0&1&0&1&1&0&0&1&1&0&0&1&1&1&0&1&0&0&0&1&0\\
0&0&0&1&1&0&1&0&0&1&0&1&0&1&0&1&1&0&0&1&1&0&0&1&1&1&0&1&0&0&0&1\\
1&0&0&0&1&1&0&1&0&0&1&0&1&0&1&0&1&1&0&0&1&1&0&0&1&1&1&0&1&0&0&0\\
0&1&0&0&0&1&1&0&1&0&0&1&0&1&0&1&0&1&1&0&0&1&1&0&0&1&1&1&0&1&0&0\\
0&0&1&0&0&0&1&1&0&1&0&0&1&0&1&0&1&0&1&1&0&0&1&1&0&0&1&1&1&0&1&0\\
0&0&0&1&0&0&0&1&1&0&1&0&0&1&0&1&0&1&0&1&1&0&0&1&1&0&0&1&1&1&0&1\\
1&0&0&0&1&0&0&0&1&1&0&1&0&0&1&0&1&0&1&0&1&1&0&0&1&1&0&0&1&1&1&0\\
0&1&0&0&0&1&0&0&0&1&1&0&1&0&0&1&0&1&0&1&0&1&1&0&0&1&1&0&0&1&1&1\\
1&0&1&0&0&0&1&0&0&0&1&1&0&1&0&0&1&0&1&0&1&0&1&1&0&0&1&1&0&0&1&1\\
1&1&0&1&0&0&0&1&0&0&0&1&1&0&1&0&0&1&0&1&0&1&0&1&1&0&0&1&1&0&0&1\\
1&1&1&0&1&0&0&0&1&0&0&0&1&1&0&1&0&0&1&0&1&0&1&0&1&1&0&0&1&1&0&0
\end{bmatrix}
$$

$$
M_k = \begin{bmatrix}
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\\
0&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\\
1&0&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&1&0&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&1&0&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\\
0&1&1&1&0&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&0&1&1&1&0&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&0&1&1&1&0&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\\
1&0&0&0&1&1&1&0&1&0&1&0&1&0&0&1&1&1&0&0&1&1&1&0&1&1&0&0&0&0&0&0\\
0&1&0&0&0&1&1&1&0&1&0&1&0&1&0&0&1&1&1&0&0&1&1&1&0&1&1&0&0&0&0&0\\
0&0&1&0&0&0&1&1&1&0&1&0&1&0&1&0&0&1&1&1&0&0&1&1&1&0&1&1&0&0&0&0\\
0&0&0&1&0&0&0&1&1&1&0&1&0&1&0&1&0&0&1&1&1&0&0&1&1&1&0&1&1&0&0&0\\
0&0&0&0&1&0&0&0&1&1&1&0&1&0&1&0&1&0&0&1&1&1&0&0&1&1&1&0&1&1&0&0\\
0&0&0&0&0&1&0&0&0&1&1&1&0&1&0&1&0&1&0&0&1&1&1&0&0&1&1&1&0&1&1&0\\
0&0&0&0&0&0&1&0&0&0&1&1&1&0&1&0&1&0&1&0&0&1&1&1&0&0&1&1&1&0&1&1\\
1&0&0&0&0&0&0&1&0&0&0&1&1&1&0&1&0&1&0&1&0&0&1&1&1&0&0&1&1&1&0&1\\
1&1&0&0&0&0&0&0&1&0&0&0&1&1&1&0&1&0&1&0&1&0&0&1&1&1&0&0&1&1&1&0\\
0&1&1&0&0&0&0&0&0&1&0&0&0&1&1&1&0&1&0&1&0&1&0&0&1&1&1&0&0&1&1&1\\
1&0&1&1&0&0&0&0&0&0&1&0&0&0&1&1&1&0&1&0&1&0&1&0&0&1&1&1&0&0&1&1\\
1&1&0&1&1&0&0&0&0&0&0&1&0&0&0&1&1&1&0&1&0&1&0&1&0&0&1&1&1&0&0&1\\
1&1&1&0&1&1&0&0&0&0&0&0&1&0&0&0&1&1&1&0&1&0&1&0&1&0&0&1&1&1&0&0\\
0&1&1&1&0&1&1&0&0&0&0&0&0&1&0&0&0&1&1&1&0&1&0&1&0&1&0&0&1&1&1&0\\
0&0&1&1&1&0&1&1&0&0&0&0&0&0&1&0&0&0&1&1&1&0&1&0&1&0&1&0&0&1&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&1&0&1&0&1&0&0&1&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&1&0&1&0&1&0&0&1\\
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&1&0&1&0&1&0&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&1&0&1&0&1&0\\
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&1&0&1&0&1\\
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&1&0&1&0\\
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&1&0&1\\
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&1&0\\
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&1
\end{bmatrix}
$$

$$M_k^{-1} = \begin{bmatrix}
1&1&0&1&0&1&1&1&0&0&1&0&1&0&0&0&0&0&0&0&0&1&1&0&0&0&0&0&1&0&0&0\\
0&1&1&0&1&0&1&1&1&0&0&1&0&1&0&0&0&0&0&0&0&0&1&1&0&0&0&0&0&1&0&0\\
0&0&1&1&0&1&0&1&1&1&0&0&1&0&1&0&0&0&0&0&0&0&0&1&1&0&0&0&0&0&1&0\\
0&0&0&1&1&0&1&0&1&1&1&0&0&1&0&1&0&0&0&0&0&0&0&0&1&1&0&0&0&0&0&1\\
1&0&0&0&1&1&0&1&0&1&1&1&0&0&1&0&1&0&0&0&0&0&0&0&0&1&1&0&0&0&0&0\\
0&1&0&0&0&1&1&0&1&0&1&1&1&0&0&1&0&1&0&0&0&0&0&0&0&0&1&1&0&0&0&0\\
0&0&1&0&0&0&1&1&0&1&0&1&1&1&0&0&1&0&1&0&0&0&0&0&0&0&0&1&1&0&0&0\\
0&0&0&1&0&0&0&1&1&0&1&0&1&1&1&0&0&1&0&1&0&0&0&0&0&0&0&0&1&1&0&0\\
0&0&0&0&1&0&0&0&1&1&0&1&0&1&1&1&0&0&1&0&1&0&0&0&0&0&0&0&0&1&1&0\\
0&0&0&0&0&1&0&0&0&1&1&0&1&0&1&1&1&0&0&1&0&1&0&0&0&0&0&0&0&0&1&1\\
1&0&0&0&0&0&1&0&0&0&1&1&0&1&0&1&1&1&0&0&1&0&1&0&0&0&0&0&0&0&0&1\\
1&1&0&0&0&0&0&1&0&0&0&1&1&0&1&0&1&1&1&0&0&1&0&1&0&0&0&0&0&0&0&0\\
0&1&1&0&0&0&0&0&1&0&0&0&1&1&0&1&0&1&1&1&0&0&1&0&1&0&0&0&0&0&0&0\\
0&0&1&1&0&0&0&0&0&1&0&0&0&1&1&0&1&0&1&1&1&0&0&1&0&1&0&0&0&0&0&0\\
0&0&0&1&1&0&0&0&0&0&1&0&0&0&1&1&0&1&0&1&1&1&0&0&1&0&1&0&0&0&0&0\\
0&0&0&0&1&1&0&0&0&0&0&1&0&0&0&1&1&0&1&0&1&1&1&0&0&1&0&1&0&0&0&0\\
0&0&0&0&0&1&1&0&0&0&0&0&1&0&0&0&1&1&0&1&0&1&1&1&0&0&1&0&1&0&0&0\\
0&0&0&0&0&0&1&1&0&0&0&0&0&1&0&0&0&1&1&0&1&0&1&1&1&0&0&1&0&1&0&0\\
0&0&0&0&0&0&0&1&1&0&0&0&0&0&1&0&0&0&1&1&0&1&0&1&1&1&0&0&1&0&1&0\\
0&0&0&0&0&0&0&0&1&1&0&0&0&0&0&1&0&0&0&1&1&0&1&0&1&1&1&0&0&1&0&1\\
1&0&0&0&0&0&0&0&0&1&1&0&0&0&0&0&1&0&0&0&1&1&0&1&0&1&1&1&0&0&1&0\\
0&1&0&0&0&0&0&0&0&0&1&1&0&0&0&0&0&1&0&0&0&1&1&0&1&0&1&1&1&0&0&1\\
1&0&1&0&0&0&0&0&0&0&0&1&1&0&0&0&0&0&1&0&0&0&1&1&0&1&0&1&1&1&0&0\\
0&1&0&1&0&0&0&0&0&0&0&0&1&1&0&0&0&0&0&1&0&0&0&1&1&0&1&0&1&1&1&0\\
0&0&1&0&1&0&0&0&0&0&0&0&0&1&1&0&0&0&0&0&1&0&0&0&1&1&0&1&0&1&1&1\\
1&0&0&1&0&1&0&0&0&0&0&0&0&0&1&1&0&0&0&0&0&1&0&0&0&1&1&0&1&0&1&1\\
1&1&0&0&1&0&1&0&0&0&0&0&0&0&0&1&1&0&0&0&0&0&1&0&0&0&1&1&0&1&0&1\\
1&1&1&0&0&1&0&1&0&0&0&0&0&0&0&0&1&1&0&0&0&0&0&1&0&0&0&1&1&0&1&0\\
0&1&1&1&0&0&1&0&1&0&0&0&0&0&0&0&0&1&1&0&0&0&0&0&1&0&0&0&1&1&0&1\\
1&0&1&1&1&0&0&1&0&1&0&0&0&0&0&0&0&0&1&1&0&0&0&0&0&1&0&0&0&1&1&0\\
0&1&0&1&1&1&0&0&1&0&1&0&0&0&0&0&0&0&0&1&1&0&0&0&0&0&1&0&0&0&1&1\\
1&0&1&0&1&1&1&0&0&1&0&1&0&0&0&0&0&0&0&0&1&1&0&0&0&0&0&1&0&0&0&1
\end{bmatrix}$$

B Changelog

- 2019-03-29: version v1.0.