SATURNIN: a suite of lightweight symmetric algorithms for post-quantum security

Version 1 (March 29, 2019)

https://project.inria.fr/saturnin/

Anne Canteaut[1], Sébastien Duval[2], Gaëtan Leurent[1], María Naya-Plasencia[1], Léo Perrin[1], Thomas Pornin[3] and André Schrottenloher[1]

[1] Inria, France, {anne.canteaut,gaetan.leurent,maria.naya_plasencia,leo.perrin,andre.schrottenloher}@inria.fr
[2] UCL Crypto Group, Belgium, sebastien.pf.duval@gmail.com
[3] NCC Group, Canada pornin@bolet.org

Abstract. The cryptographic algorithms needed to ensure the security of our communications have a cost. For devices with little computing power, whose number is expected to grow significantly with the spread of the Internet of Things (IoT), this cost can be a problem. A simple answer to this problem is a compromise on the security level: through a weaker round function or a smaller number of rounds, the security level can be decreased in order to cheapen the implementation of the cipher. At the same time, quantum computers are expected to disrupt the state of the art in cryptography in the near future. For public key cryptography, the NIST has organized a dedicated process to standardize new algorithms. The impact of quantum computing is harder to assess in the symmetric case but its study is an active research area.

In this document, we specify a new block cipher, SATURNIN, and its usage in different modes to provide hashing and authenticated encryption in such a way that we can rigorously argue its security in the post-quantum setting. Its security analysis follows naturally from that of the AES, while our use of components that are easily implemented in a bitsliced fashion ensures a low cost for our primitives. Our aim is to provide a new lightweight suite of algorithms that performs well on small devices, in particular micro-controllers, while providing a high security level *even in the presence of quantum computers*.

SATURNIN is a 256-bit block cipher with a 256-bit key and an additional 9-bit parameter for domain separation. Using it, we built two authenticated ciphers and a hash function.

- SATURNIN-CTR-Cascade is an authenticated cipher using the counter mode and a separate MAC. It requires two passes over the data but its implementation does not require the inverse block cipher.
- SATURNIN-Short is an authenticated cipher intended for messages with a length strictly smaller than 128 bits which uses only one call to SATURNIN to provide confidentiality and integrity.
- SATURNIN-Hash is a 256-bit hash function.

In this document, we specify this suite of algorithms and argue about their security in both the classical and the post-quantum setting.

Contents

1 Introduction

In the report on lightweight cryptography NISTIR8114[1], it was explicitly asked that the algorithms submitted to the project should be quantum-safe when long-term security is needed:

> *"When long-term security is needed, these algorithms should either aim for post-quantum security, or the application should allow them to be easily replaceable by algorithms with post-quantum security."*

This completely makes sense, as the effort for recommendations for post-quantum asymmetric cryptography is being made, and this would only be effective if the symmetric cryptography used with it is also quantum resistant.

However, to achieve an effective quantum security of 128 bits, it does not seem enough to use a 256-bit key-size. Indeed, the security of most modes of operation is limited by the complexity of finding collisions, which may benefit of a quantum acceleration, depending only on the block size. This yields a first challenge: the design of a block cipher with a bigger state (we will take 256 bits).

A second challenge is motivated by the results from [KLLN16a] which show that most authentication modes suffer from polynomial-time attacks in the message-superposition setting. Whether these attacks could be avoided was an open question related to the influence of the nonce on the different calls to the block cipher.

A third challenge is that of lightness: it is not sufficient that our algorithm be post-quantum secure, it should also be suitable for efficient implementation on devices with little computing power.

In this proposal we overcome all those issues first by designing a block cipher with a 256-bit internal state and 256-bit key that inherits the AES security properties while allowing an efficient bitsliced implementation, and secondly by proposing modes of operation resisting the aforementioned attacks.

How to efficiently build an AES with a 256-bit state? Our proposal, SATURNIN is a suite of lightweight symmetric algorithms for post-quantum security. As previously explained, our first aim was to design a block cipher, not only with a key-size equal to twice the wanted security level, but also a double state-size. SATURNIN therefore works on blocks of 256 bits, yet also aims at being particularly suitable for high-security microcontroller applications, thanks to an efficient bitslice implementation. We wanted to take advantage of the knowledge obtained from the AES analysis and the wide-trail strategy [DR02a], since the AES can be safely considered as the most analyzed block cipher. Indeed, in order to benefit from this 20 year-long cryptanalysis effort, we have built a 3-dimensional AES, on which the wide-trail strategy can still be applied.

On larger versions of Rijndael. In the original Rijndael submission to the NIST competition [DR99], larger-state versions of the cipher were proposed, that were not kept in the standard. The diffusion within the internal state was less efficient since the state was represented, in almost all versions, by a rectangle and not a square. This slower diffusion was exploited in several attacks, taking advantage of the larger internal state [GM08, MPP09, NP07, ZWP+08, WGR+13, Sas10]. The long-key versions also seem relatively less secure than the versions using shorter keys, especially regarding related-key attacks [BKN09, BK09]. Moreover, from an implementation point of view, their performances make them less competitive than the version with a 128-bit internal state defined in the standard.

[1] https://nvlpubs.nist.gov/nistpubs/ir/2017/NIST.IR.8114.pdf

Previous attempts. The idea of an AES version with three dimensions was mentioned in [DR02b], without further development. In 2008 the block cipher 3D was proposed by Nakahara Jr. in [Nak08]. SATURNIN follows an overall approach reminiscent of 3D in that it defines a 3D AES-like block cipher which uses similar subroutines. However, SATURNIN differs from 3D in at least two key ways. First, the order in which we apply these operations differs. This allows us to claim 125 active S-boxes for 8 rounds, unlike in 3D. Furthermore, all of our operations are intended for an efficient bitsliced implementation, meaning that our cipher is much more suited for the lightweight context.

Some hash function proposals based on the AES transformations with a large internal state were also submitted to the SHA-3 competition, like LANE [IAC+08], Grøstl [GKM+08], and ECHO [GBB+08]. However, in each case, the approach used to handle a larger state is different: LANE uses several independent AES states, Grøstl uses a large MDS matrix, and ECHO resembles a four-dimensional AES, but the wide-trail arguments only go through three dimensions[2].

1.1 Post-quantum Symmetric Cryptography

Quantum computation was first introduced in the late 80s as a general framework and potential tool for simulating quantum systems. Since then, it has been the subject of much more attention in computer science since the introduction by Shor [Sho94] of a quantum algorithm for solving factorization and discrete logarithms in polynomial time. Since then, the cryptographic community has been concerned with the impact of large or intermediate-scale quantum computers which, although they are yet to be built, would have massive consequences on these cryptosystems, breaking most of those that are in use today.

It is widely acknowledged that new cryptographic designs should take into account the quantum threat. As examples of this new direction, one may cite the NIST post-quantum standardization project [Nat16], which structures most of the efforts of the asymmetric cryptographic community, or the report of the National Academies of Sciences [Nat18], which gives a precise evaluation of the quantum threat, up to the uncertainties inherent to the evolution of cutting-edge technologies.

Until recently, such concerns did not seem to apply to symmetric cryptography, which does not rely on structured mathematical problems such as factorization. In contrast, Grover's algorithm [Gro96] provides a quadratic speedup for a wide range of exhaustive search problems, which are relevant to symmetric cryptography. In particular, such a speedup occurs when performing an exhaustive search for secret keys, which brings the cost of this search from 2^{128} to its square root 2^{64} in the case of a 128-bit keyed block cipher such as AES-128. This leads to the natural countermeasure of increasing key sizes, as the report [Nat18] indicates:

> *"Even if a computer existed that could run Grover's algorithm to attack AES-GCM, the solution is quite simple: increase the key size of AES-GCM from 128-bit to 256-bit keys. "*

This indeed brings the cost back to 2^{128}.

Many works have dealt with the precise complexity of Grover's search for practical attacks [GLRS16], which may be more difficult to run in practice than the first estimate of 2^{64} time. For example, one needs to implement symmetric cryptographic operations on a quantum computer, and this is not trivial (see for instance the careful evaluation of applying Grover on AES from [GLRS16]). But, as the report [Nat18] also mentions, this exhaustive search of the key is only an upper bound on the security, not a lower bound;

[2] The bounds given in [GBB+08] show at least 200 active S-Boxes for 8 S-box layers, but a 4D AES would have 625 active S-Boxes for 16 S-box layers

more analysis is needed to assert the security of AES (or other ciphers) against quantum computers.

> *"More precisely, it is possible that there is some currently unknown clever quantum attack on AES-GCM that is far more efficient than Grover's algorithm. "*

Recent works have shed a new light on the quantum security (or insecurity) of some symmetric cryptographic constructions [KLLN16a, CNS17, BNPS19]. In general, it can be studied in two models defined *e.g.* in [Gag17, GHS16, KLLN16b].

Q1 Model. A first natural question is the security with an *offline* quantum computer. While the adversary can only recover classical data, for instance by making queries to a secret-key oracle, he can run quantum computations. Secret-key exhaustive search using Grover's algorithm runs in this model, as the adversary only needs a few classical plaintext-ciphertext queries to check his key guesses. Quantum collision search for hash functions runs in this model, as the specification of a hash function is public, and so a quantum adversary is perfectly capable of implementing this function as a quantum circuit.

Q2 Model. The Q2 model is a strictly more powerful setting for the adversary, since the adversary is allowed to perform *quantum superposition* access to secret-key oracles. There exist classical primitives enjoying a classical security proof, for example the Even-Mansour cipher with a random permutation, which are broken in the Q2 model in polynomial time [KM10, KM12]. In [KLLN16a], it was shown that such attacks could actually target many modes of operation, and accelerate exponentially some classical slide attacks, which concern ciphers with a repetitive structure.

This is a powerful model, but there are many good reasons to consider it, like for instance that it is simple and easy to define; that it is non-trivial, as many primitives and constructions remain resistant in this setting; and also the fact that it includes all possible intermediate scenarios, like the ones were the primitives could be implemented in no-safe manners as components of more complex protocols that might include obfuscation, or implementations on hybrid systems. Countering these attacks using *ad hoc* methods, for instance by enforcing an input measure that would make the superposition collide, does not seem easy to guarantee nor simple to implement for now. Due to all these reasons, Q2 is the most-widely used model in quantum security proofs, e.g., for quantum proofs of MAC constructions [BZ13b, AMRS], and we have chosen to build a candidate offering security in this model. Throughout this specification document, whenever we consider an oracle such as a secret-key encryption or decryption oracle, or a tag verification oracle, we consider a quantum adversary to have access to the corresponding Q2 oracle, and our quantum security claims are *all* in this model.

Other Limitations. Some classically efficient algorithms do not enjoy a clear quantum speedup and, in particular, their quantum versions may encounter new hardware limitations. Such an example is quantum collision search. For an n-bit random function, classical collision search runs in time $O(2^{n/2})$, corresponding to the birthday bound, and in $O(n)$ memory thanks to Pollard's rho method. A quantum collision search algorithm found in 1998 [BHT98] reached a complexity of $O(2^{n/3})$ time and queries for the same problem, later proven to be optimal. However, it requires $2^{n/3}$ quantum memory, so a more than significant amount of hardware. To date, no quantum algorithm optimal in time and as efficient in hardware as Pollard's rho exists for this problem. A tradeoff was given in [CNS17] with a suboptimal time complexity of $2^{2n/5}$, but only $O(n)$ quantum memory, showing that quantum collision search should not be immediately ruled out because of its apparent impracticality. The most conservative approach should take $2^{n/3}$ as a quantum

security level, but we choose later to give a bound depending on the amount of quantum memory available to our offline adversary. Additionally, some quantum algorithms such as Grover's encounter inherent difficulties at parallelization. We choose to remain conservative in this setting and consider only a quantum time complexity for a single processor. We aim at a resistance against quantum adversaries even allowing them to use big quantities of quantum memory.

1.2 Security claims

In general, we believe that, for each element in the SATURNIN suite, there is no attack *significantly* better than the generic attacks against the corresponding construction. We now formulate specific security claims taking into account the requirements mentioned in the NIST call for submissions, and we also formulate security claims regarding quantum adversaries.

Classically, the complexity of an attack is determined by the following quantities: $\mathcal{T}$ is the time complexity, expressed in units equivalent to the cost of one evaluation of the involved SATURNIN block cipher; $\mathcal{D}$ is the data complexity (encryption and verification queries), expressed as a number of 256-bit blocks, and p is the success probability of an adversary. It is worth noticing that $\mathcal{T} \geq \mathcal{D}$ since the time for generating the data must be taken into account.

Quantumly, we adapt these definitions. Quantum computations are traditionally written using *quantum circuits* and the *time complexity* is given by the number of *quantum gates* in the circuit. For us, $\mathcal{T}$ is counted in units equivalent to the number of gates of one evaluation of SATURNIN *implemented as a quantum circuit* (please, see appendix B for more details and definitions regarding quantum computing). Hence our security levels are independent of the cost of a quantum circuit for SATURNIN. $\mathcal{D}$ is the number of 256-bit blocks queried to a single superposition oracle. For example, for a mode of encryption with variable message-length, the adversary chooses the message-length she wants to query and queries the oracle in superposition over these messages. These oracle calls are interleaved with the quantum computations and each one requires as much time units as the number of (quantum) secret-key computations of SATURNIN required. Notice that classical oracle calls are also available with the same cost, since they represent the special case of a collapsed input superposition.

Block cipher. In the following claims, by default, SATURNIN denotes the block cipher with at least 10 super-rounds, *i.e.* 20 single-rounds as a super-round is formed by 2 rounds. When explicitly mentioned, SATURNIN_{16} corresponds to the block cipher with 16 super-rounds or more.

Security Claim for SATURNIN block cipher (Section 2.1)

There exists no **classical** attack in the single-key setting with $\mathcal{T}/p < 2^{224}$. SATURNIN_{16} provides a similar security level against related-key attacks involving a small number of keys[a].
There exists no **quantum** attack in the single-key setting with $\mathcal{T}/p < 2^{112}$. SATURNIN does not provide security against related-key superposition attacks (as is the case of all known block ciphers).

[a] with related-key deriving functions satisfying the conditions of [BK03].

Authenticated encryption. The SATURNIN suite includes two AE schemes (one with Associated Data). In the following security claims, t is the length of the tags in bits (t is 256 by default, but can be less if the tags are truncated).

None of the AE schemes in SATURNIN provides security in nonce-misuse, nonce repetition or nonce-superposition scenarios. In the single-key setting, we make the following claims:

Security Claim for SATURNIN-CTR-Cascade (Section 2.2)

There exists no **classical** attack satisfying $\frac{\mathcal{D}^2+\mathcal{T}+\mathcal{D}2^{256-t}}{p} < 2^{224}$.
There exists no **quantum** attack satisfying $\frac{\mathcal{D}^3+\mathcal{T}^2+\mathcal{D}^2 2^{256-t}}{p} < 2^{224}$.

In particular, in the case where $t = 256$ these claims imply that any classical attack with high success probability must satisfy $\mathcal{DT} > 2^{224}$, which corresponds to the birthday bound at the 224-bit security level.

SATURNIN-Short is an efficient AE scheme (without Associated Data) dedicated to short messages, of length strictly less than 128 bits.

Security Claim for SATURNIN-Short (Section 2.3)

There exists no **classical** attack with $\frac{\mathcal{D}^2+\mathcal{T}+\mathcal{D}2^{128}}{p} < 2^{224}$.
There exists no **quantum** attack with $\frac{\mathcal{D}^3+\mathcal{T}^2+\mathcal{D}^2 2^{128}}{p} < 2^{224}$.

The AE claims above are written in the single-key setting, assuming that related-key issues are dealt with at the protocol level. However, we also claim security against classical related-key attacks (with the same restrictions on the related keys as for the block cipher claims) when SATURNIN is replaced by SATURNIN$_{16}$.

Hash function. The hash function in the SATURNIN suite is based on SATURNIN$_{16}$. A lower number of super-rounds is not recommended. In what follows, $\mathcal{M}_q$ is the size of the quantum memory measured in registers of 256 qubits.

Security Claim for SATURNIN-Hash (Section 2.4)

There exists no **classical** collision attack with $\mathcal{T} < 2^{112}$. There exists no **classical** second-preimage attack with $\mathcal{T} < 2^{224-\ell}$ for messages of length 2^ℓ. There exists no **classical** preimage attack with $\mathcal{T} < 2^{224}$.
There exists no **quantum** collision attack verifying $\mathcal{T}^5 \times \mathcal{M}_q < 2^{448}$. There exists no **quantum** second-preimage attack with $\mathcal{T} < 2^{112-\ell/2}$ for messages of length 2^ℓ. There exists no **quantum** preimage attack with $\mathcal{T} < 2^{112}$.

In particular, the claim for quantum collision attack implies that there is no such attack with $\mathcal{T} < 2^{75}$, because we necessarily have $\mathcal{M}_q < \mathcal{T}$.

1.3 Recommended parameters

Our recommended variants of SATURNIN use 10 super-rounds for authenticated encryption, and 16 super-rounds for hashing.

1.3.1 Primary member

AEAD: Saturnin-CTR-Cascade with 10 super-rounds
Hash: Saturnin-Hash with 16 super-rounds

The primary member combines modes with strong post-quantum security guarantees, no known patent claims, and no need for the block-cipher decryption. As mentioned in the previous section, we claim security up to the birthday bound, in the single-key setting.

1.3.2 Variant: optimized for small messages

AE: Saturnin-Short with 10 super-rounds
Hash: Saturnin-Hash with 16 super-rounds

This variant is a special construction dedicated to short messages of at most 128 bits, without associated data. The authentication security is slightly decreased, equivalent to truncating the tag to 128 bits.

In practice, if there is a need to encrypt both short messages and longer messages, we recommend to use Saturnin-CTR-Cascade with a 128-bit tag for longer messages, with an extra ciphertext bit for domain separation.

2 Specification

2.1 The Block Cipher Saturnin

Saturnin operates on 256-bit blocks, using a 256-bit key. It uses a 256-bit internal state. Blocks, keys and state values can be viewed as several equivalent representations:

- as a $4 \times 4 \times 4$ cube of 4-bit nibbles (Figure 1a);
- as sixteen 16-bit registers, indexed from 0 to 15, known as the "bitsliced representation" (Figure 1b);
- as 256 bits.

In this section, we use the first representation as a cube of nibbles. Section 3 will describe the bitsliced representation, and how conversions are performed between all three representations. Practical implementations are expected to use the bitsliced representation; hence, the description in this section is only for formal presentation.

The nibbles in the cube are numbered from 0 to 63, as on Figure 1a. Each nibble can also be defined via coordinates (x, y, z) such that the coordinates (x, y, z) correspond to the nibble with index $(y + 4x + 16z)$. We number the bits within each nibble from 0 to 3, where 0 is the least significant bit.

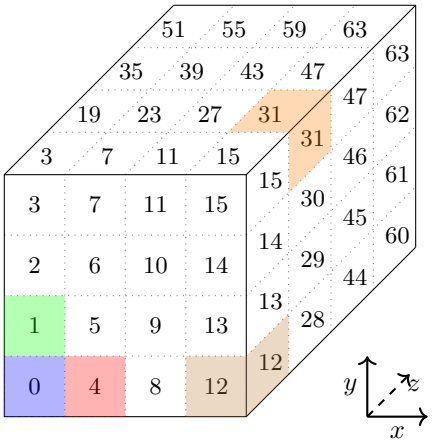
2.1.1 Internal State

The various transformations in Saturnin are defined over different subsets of its state. Below, we define three such subsets of the cube, based on the same terminology as in SHA-3 (see Figure 1 in [Nat15]).

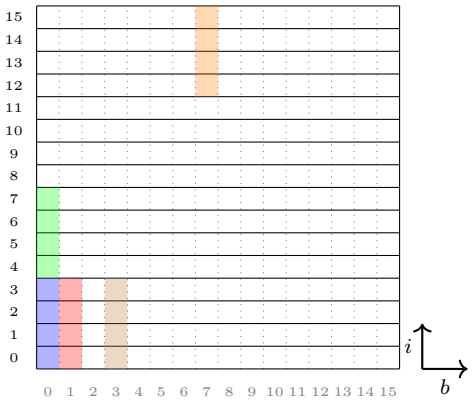
Slice. In the cube, a slice is a subset of the nibbles such that z is constant.

Sheet. In the cube, a sheet is a subset of the nibbles such that x is constant.

Columns. Columns are the intersection of a sheet and a slice. They correspond to the sets of nibbles with x and z constant.



(a) As a $4 \times 4 \times 4$ cube of 4-bit nibbles. The boundaries between the nibbles are in gray.



(b) As sixteen 16-bit registers. The indices and boundaries of the registers are in black, those of the bits are in gray.

Figure 1: The two representations of the 256-bit state of SATURNIN. Nibbles and their corresponding bits are represented with the same color in each representation.

2.1.2 The Specification of the Block Cipher

The SATURNIN block cipher is an SPN with an even number of rounds, numbered from 0. In the following, we call a *super-round* the composition of two consecutive rounds with indices $2r$ and $(2r + 1)$.

The SATURNIN block cipher uses a 256-bit internal state X and a 256-bit key state K, both represented as a $(4 \times 4 \times 4)$-cube of nibbles. Two additional 16-bit word RC_0 and RC_1 are used for generating the successive round constants.

Parameters. The block cipher has two input parameters:

- R: the number of super-rounds, i.e. the total number of rounds divided by 2. R belongs to $\{10, \ldots, 31\}$ and is equal to 10 by default[3].
- D: a 4-bit integer, named the *domain separator*, depending on the operating mode as specified in Section 2.5. The block cipher with domain separator D will be denoted by SATURNIN^D.

Initialization. X and K are respectively initialized with the input and with the master key.

Both 16-bit registers RC_0 and RC_1 are initialized as the bit-string

$$\underbrace{1 \ldots 1}_{7 \text{ ones}} \underbrace{R_4 \ldots R_0}_{R} \underbrace{D_3 \ldots D_0}_{D}$$

where the rightmost bit of the register is the least significant bit. The first four bits are given by the domain separator $\sum_{i=0}^{3} D_i 2^i = D$, while the 5-bit integer $\sum_{i=0}^{4} R_i 2^i$ is equal to R, i.e. to the number of super-rounds.

Round 0 starts by xoring K to the internal state.

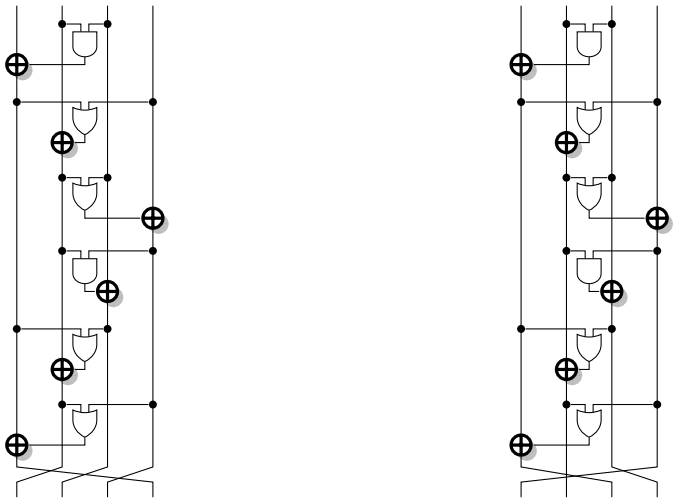
[3]Smaller values of R can be used for defining weakened versions of the cipher for analysis purposes.

Round function. Each round, starting from Round 0, then successively applies the following transformations to the internal state:

- an S-box layer S, which applies the same 4-bit Sbox σ_0 to all nibbles with an even index, and the same 4-bit Sbox σ_1 to all nibbles with an odd index. These two Sboxes are defined by their lookup tables which are given in Table 1, where x such that $\sum_{i=0}^{3} x_i 2^i = x$ corresponds to a nibble containing (x_3, x_2, x_1, x_0). An efficient implementation of the S-boxes is shown in figure 2.

Table 1: The lookup tables of the S-boxes we use.

x	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\sigma_0(x)$	0	6	14	1	15	4	7	13	9	8	12	5	2	10	3	11
$\sigma_1(x)$	0	9	13	2	15	1	11	7	6	4	5	3	8	12	10	14



(a) σ_0 **(b)** σ_1

Figure 2: Bitslice implementation of the S-box layer.

- A nibble permutation SR_r which depends on the round number r. For all even rounds, SR_r is the identity function. For odd rounds of index r with $r \bmod 4 = 1$, $\mathsf{SR}_r = \mathsf{SR}_{\text{slice}}$ consists of the parallel application of R_{slice} on each slice independently. This operation maps the nibble with coordinates (x, y, z) to $(x + y \mod 4, y, z)$. For odd rounds of index r with $r \bmod 4 = 3$, $\mathsf{SR}_r = \mathsf{SR}_{\text{sheet}}$ consists of the parallel application of R_{sheet} on each sheet independently. This operation maps the nibble with coordinates (x, y, z) to $(x, y, z + y \mod 4)$. The SR_r transformation is depicted on Figure 4.

- A linear layer MC composed of 16 copies of a linear operation M over $(\mathbb{F}_2^4)^4$ which is applied in parallel to each column of the internal state. The transformation M is defined as:

$$M : \begin{pmatrix} a \\ b \\ c \\ d \end{pmatrix} \mapsto \begin{pmatrix} \alpha^2(a) \oplus \alpha^2(b) \oplus \alpha(b) \oplus c \oplus d \\ a \oplus \alpha(b) \oplus b \oplus \alpha^2(c) \oplus c \oplus \alpha^2(d) \oplus \alpha(d) \oplus d \\ a \oplus b \oplus \alpha^2(c) \oplus \alpha^2(d) \oplus \alpha(d) \\ \alpha^2(a) \oplus a \oplus \alpha^2(b) \oplus \alpha(b) \oplus b \oplus c \oplus \alpha(d) \oplus d \end{pmatrix}$$

where a is the nibble with the lowest index, and α transforms the four bits $x_0, \ldots, x_3$ of each nibble by the following multiplication

$$\begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix}.$$

This transformation corresponds to the next-state function of an LFSR of length 4, in Fibonacci mode, with feedback polynomial $X^4 + X^3 + 1$.

The transformation M can be implemented efficiently as depicted on Figure 3 (which corresponds to Figure 13 in [DL18] up to a rotation of the nibbles).

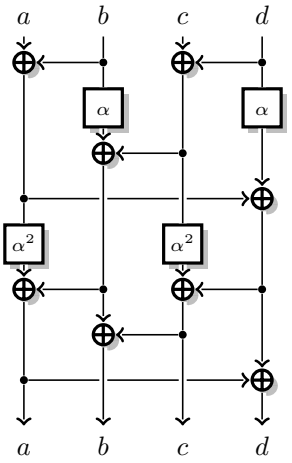


Figure 3: A 4×4 MDS matrix from [DL18]. The input/output (a, b, c, d) corresponds to nibbles with index $(4i, 4i+1, 4i+2, 4i+3)$, for $0 \leq i \leq 15$.

- The inverse of the previous nibble permutation, namely SR_r^{-1}.
- A sub-key addition at odd rounds only (*i.e.* at the end of each super-round). The sub-key is composed of the XOR of a round constant and either the master key or a rotated version of the master key:

  Round constant. The round constants RC_0 and RC_1 are updated by clocking 16 times two independent LFSR of length 16 in Galois mode with respective feedback polynomial $X^{16} + X^5 + X^3 + X^2 + 1$ and $X^{16} + X^6 + X^4 + X + 1$. In other words, we repeat 16 times the following operation: if the most significant bit of RC_i is 0, RC_i is replaced by $\mathsf{RC}_i \ll 1$, otherwise, it is replaced by $(\mathsf{RC}_i \ll 1)\hat{}\,\mathtt{poly}_i$ with $\mathtt{poly}_0 = \mathtt{0x1002d}$ and $\mathtt{poly}_1 = \mathtt{0x10053}$.

  The two 16-bit words $\mathsf{RC}_0, \mathsf{RC}_1$ are then xored to the internal state. Bit number i in RC_0 is added to Bit 0 of the nibble with index $4i$, for $0 \leq i \leq 15$. Similarly, Bit number i in RC_1 is added to Bit 0 of the nibble with index $(4i + 2)$, for $0 \leq i \leq 15$.

  Round key. If the round index r is such that $r \bmod 4 = 3$, the master key K is xored to the internal state; otherwise (*i.e.* when $r \bmod 4 = 1$), a rotated version of the key is added instead: the nibble with index i receives the key nibble with index $(i + 20) \bmod 64$, for $0 \leq i \leq 63$.

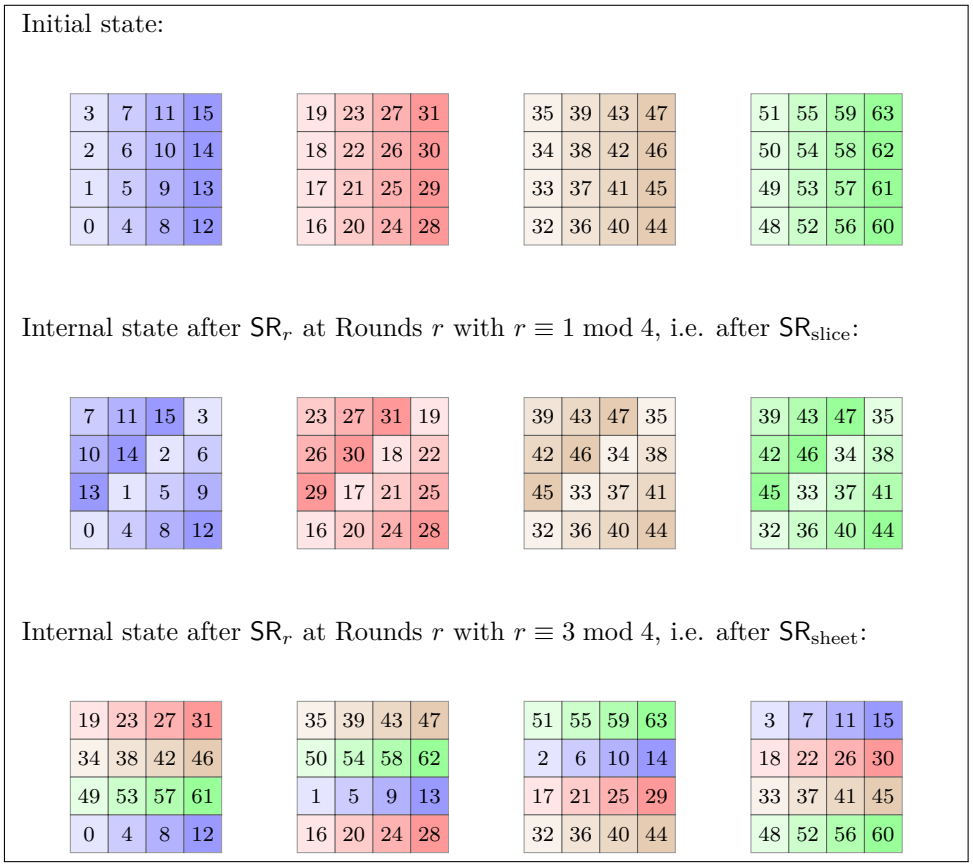
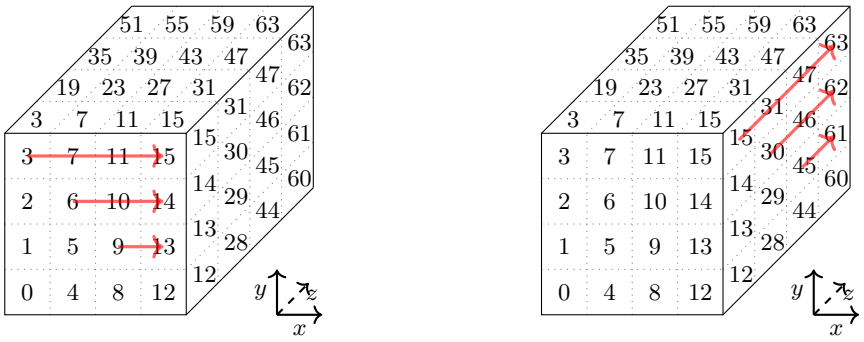


Figure 4: Ordering of the 64 nibbles of the internal state after applying SR$_r$ depending on r mod 4, when the cube is represented as the collection of its 4 slices.



(a) SR$_{\text{slice}}$ (when $r \equiv 1 \bmod 4$) **(b)** SR$_{\text{sheet}}$ (when $r \equiv 3 \bmod 4$)

Figure 5: Representation of the SR$_r$ operations on the cube.

2.2 The Authenticated Cipher SATURNIN-CTR-Cascade

We first propose to use the SATURNIN block cipher with modes known for their robustness against quantum adversaries: we combine the counter mode (Figure 6) for encryption and the Cascade construction [BCK96] for authentication, following the Encrypt-then-MAC composition. The Cascade construction is used, for example, in NMAC, and our MAC is very similar to NMAC based on SATURNIN in MMO mode. The nonce has length up to

160 bits and the *tag* has up to 256 bits (it can be truncated).

The nonce is first padded into the 161-bit string N by appending to the nonce a bit of value 1, then as many bits of value 0 as needed to reach a total length of 161 bits.

In general, whenever our proposed modes (SATURNIN-CTR-Cascade and SATURNIN-Hash) require padding a value of less than 256 bits into a 256-bit block, we use the following padding rule, and denote as $pad(x)$ the padding of block x:

Padding rule: *The padding rule consists in appending a single bit of value 1, followed by as many zeroes as necessary to reach the next block boundary.*

In the counter (CTR) mode depicted on Figure 6, we define a keystream by encrypting blocks composed of N concatenated with a 95-bit counter which is increased by one for each new call: $z_i = \text{SATURNIN}^1(k, N\|i+1)$ (the counter starts at 1 for the block z_0). The counter is encoded with the big-endian convention: if the counter numerical value u is:

$$u = \sum_{i=0}^{94} u_i 2^i$$

then each bit u_i becomes bit $255 - i$ in the input block. Thus, in a byte-oriented implementation and with a 128-bit nonce, the input to SATURNIN^1 for the computation of z_0 will consist in the 16 bytes of the nonce, followed by a byte of value `0x80` (first padding byte for the nonce), followed by 14 bytes of value `0x00`, followed by one byte of value `0x01`.

We split the input plaintext m into full blocks $m_0, m_1, \ldots m_\ell$ and a final partial block m_*. Similarly, the associated data a is split into $a_0, a_1, \ldots a_j, a_*$. The SATURNIN-CTR-Cascade encryption process is then:

- Encryption:
  1. For all $i = 0$ to ℓ: $c_i \leftarrow m_i \oplus \text{SATURNIN}^1(k, N\|i+1)$
  2. $c_* \leftarrow m_* \oplus \text{trunc}_n(\text{SATURNIN}^1(k, N\|\ell+2))$ (where trunc_n truncates its input to its first n bits, n being the length, in bits, of m_*).
- Authentication tag:
  1. $t \leftarrow (N\|0) \oplus \text{SATURNIN}^2(k, N\|0)$
  2. For all $i = 0$ to j: $t \leftarrow a_i \oplus \text{SATURNIN}^2(t, a_i)$
  3. $t \leftarrow \text{pad}(a_*) \oplus \text{SATURNIN}^3(t, \text{pad}(a_*))$
  4. For all $i = 0$ to ℓ: $t \leftarrow c_i \oplus \text{SATURNIN}^4(t, c_i)$
  5. $t \leftarrow \text{pad}(c_*) \oplus \text{SATURNIN}^5(t, \text{pad}(c_*))$

The ciphertext is $c = c_0\|c_1\|\ldots\|c_\ell\|c_*$; it has the same length as the plaintext. The authentication tag is the last value of t. When decrypting, the authentication tag is first recomputed, and compared with the received value; on mismatch, the encrypted message is rejected. If the authentication tags match, then decryption is identical to encryption. The comparison between authentication tags should endeavour not to reveal the bit position at which mismatched tags diverge.

All SATURNIN calls use $R = 10$ super-rounds. Five domain values are used:

- SATURNIN^1: for CTR encryption;
- SATURNIN^2: for Cascade over the associated data full blocks;
- SATURNIN^3: for Cascade over the padded last block of associated data;
- SATURNIN^4: for Cascade over the ciphertext full blocks;
- SATURNIN^5: for Cascade over the padded last block of ciphertext.

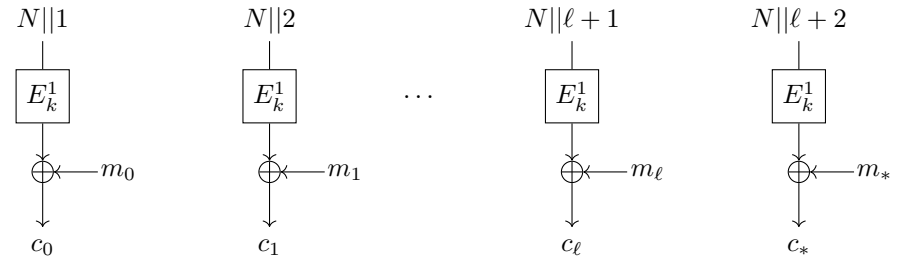


Figure 6: Counter Mode (CTR) encryption (E_k^i denotes SATURNIN^i under Key k).

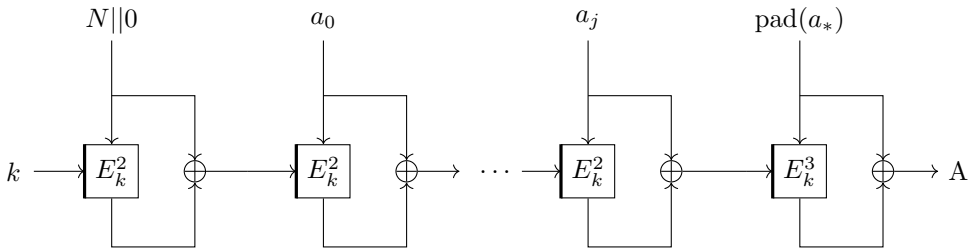


Figure 7: Cascade construction, processing of the associated data. A thick line represents the input of the key.

2.3 The Authenticated Cipher SATURNIN-Short

For some practical usages, we propose a third authenticated cipher for handling messages of length strictly less than 128 bits (without additional data) and nonces of size up to 128 bits: SATURNIN-Short (Figure 9). We denote by || concatenation of bit-strings and again, adopt the same padding convention as in Section 2.2 for messages shorter than 128 bits.

In SATURNIN-Short, we use the fact that SATURNIN has a 256-bit block size, allowing to mix together the nonce and the message. We use the variant SATURNIN^6. It is worth noticing that the ciphertext and the tag are not two separate values. Given a nonce N and a ciphertext c, the tag can be verified by deciphering c and comparing the left half with N.

Since SATURNIN-Short does not allow additional data, we recommend that protocols based on SATURNIN-Short use a counter as the nonce to prevent reordering of the ciphertexts and to detect replay attacks.

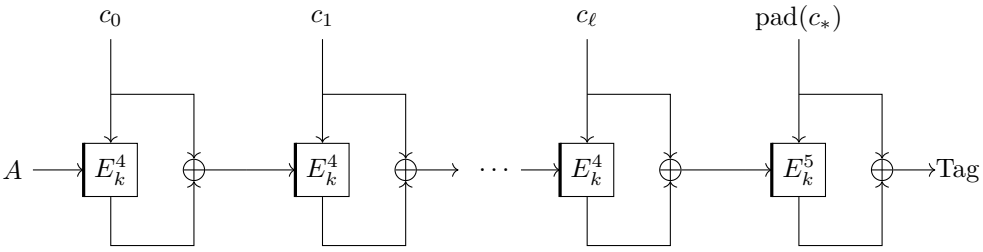


Figure 8: Cascade construction, processing of the ciphertext and computation of the tag. A thick line represents the input of the key.

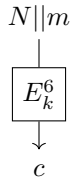


Figure 9: SATURNIN-Short with key k, message m and nonce N.

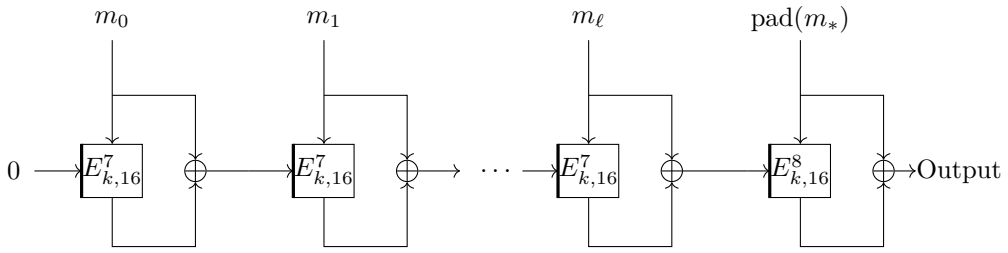


Figure 10: The hash function SATURNIN-Hash.

2.4 The Hash Function SATURNIN-Hash

We also propose a 256-bit hash function based on SATURNIN with the Merkle-Damgård construction, as shown in Figure 10. We use SATURNIN_{16} (*i.e.* SATURNIN with 16 super-rounds) because we need the compression function to be resistant in the related-key setting. The compression function uses the MMO mode ($f(c_i, m) = E_{c_i}(m) \oplus m$) to compress a 256-bit chaining value c_i and a 256-bit message block. Therefore, SATURNIN-hash is similar to the above Cascade, except that there is no key as starting point but a fixed value, 0.

As in SATURNIN-CTR-Cascade, the input message m is split into full blocks $m_0, m_1, \ldots m_\ell$, and a final partial block m_* (which may be empty, but is never full). Processing is then:

1. $t \leftarrow 0$ (the all-zero 256-bit block)
2. For all $i = 0$ to ℓ: $t \leftarrow m_i \oplus \text{SATURNIN}^7_{16}(t, m_i)$
3. $t \leftarrow \mathrm{pad}(m_*) \oplus \text{SATURNIN}^8_{16}(t, \mathrm{pad}(m_*))$

The hash output is the final value of t.

2.5 Values of the Domain Separator

The values of the domain separator D are used to describe the variant of SATURNIN that will be used in each role of each mode. We summarize the correspondence in Table 2.

3 Implementation

3.1 Bitslice Representation and Conversions

The "cube of nibbles" representation of SATURNIN blocks, keys and state values uses 64 nibbles with coordinates (x, y, z) (all three coordinates range from 0 to 3). Each nibble contains four bits, numbered from 0 (least significant) to 3 (most significant).

The "bitsliced" representation splits the value into sixteen registers of 16 bits each. A bit within a register is indexed by values (i, b), where i is the register number (0 to 15),

Table 2: Correspondence between the value of the domain separator D and the usage of SATURNIN^D.

Value of D	Use
0	SATURNIN block cipher
1	SATURNIN-CTR
2	SATURNIN-Cascade AD
3	SATURNIN-Cascade AD final
4	SATURNIN-Cascade message
5	SATURNIN-Cascade message final
6	SATURNIN-Short
7	SATURNIN-Hash
8	SATURNIN-Hash final

and b is the bit number (0 to 15). Within a register, bit 0 is least significant, and 15 is most significant. The register bits map to the nibble bits in the following way:

$$(4j + k, b) \longrightarrow (b \bmod 4, j, \lfloor b/4 \rfloor)_k \quad (0 \leq j \leq 3, 0 \leq k \leq 3, 0 \leq b \leq 15)$$

i.e. the bits 0 to 3 of the nibble $(b \bmod 4, j, \lfloor b/4 \rfloor)$ are the bits b of registers $4j$, $4j + 1$, $4j + 2$ and $4j + 3$, respectively.

The "bits" representation encodes the registers of the bitsliced representation into bits. The traditional mixed-endian convention is used: the two *octets* (bytes) of a register are encoded in little-endian order (least significant octet first), but bits within each octet are reputed to be ordered from most significant to least significant. Registers are encoded in ascending numerical order (0 to 15).

In most practical implementations, SATURNIN will operate on keys and blocks which are already grouped into octets. In that case, this preexisting grouping of bits into octets is assumed to already follow the convention prescribed above. Therefore, the first two octets t_0 and t_1, each with numerical value 0 to 255, encode register 0 with numerical value $t_0 + 256t_1$; the next two octets t_2 and t_3 encode register 1 with numerical value $t_2 + 256t_3$; and so on.

In a byte-oriented implementation, the ordering of bits within a byte matters only for the padding rule used in SATURNIN-Hash and the AEAD modes: when appending a bit of value 1 followed by zeros, this translates to appending a byte of value `0x80` followed by bytes of value `0x00`.

It shall be noted that in SATURNIN-CTR-Cascade, we specified the encoding of the counter to use big-endian, since this is the standard encoding in other AEAD modes such as GCM, CCM or EAX. The conversion of the counter numerical values into the values of the high-index registers will thus imply some byteswapping.

3.2 Re-interpreting the Operations of SATURNIN

While specified over a cube, SATURNIN is best implemented using a bit-sliced approach. For implementation purposes, the internal state of SATURNIN is represented as sixteen 16-bit registers indexed from 0 to 15 (Figure 1b). In this register representation, the bit of lowest weight of each register has index 0 and the bit of highest weight as index 15. A bit is then defined by the index i of its register and its index b within its register. The correspondence between the register representation and the cube representation is as follows: the bits with coordinates $(4i, b), (4i + 1, b), (4i + 2, b)$ and $(4i + 3, b)$, which are therefore taken from registers $4i$ to $4i + 3$, correspond to nibble $(b \mod 4, i, \lfloor b/4 \rfloor)$ in the cube (see Figure 1).

A slice in the cube corresponds to bits with indices (i, b) such that $\lfloor b/4 \rfloor$ is constant, while a sheet corresponds to bits with indices (i, b) such that $(b \mod 4)$ is constant. Therefore, the columns in the cube are composed of bits (i, b) with a constant b in the registers.

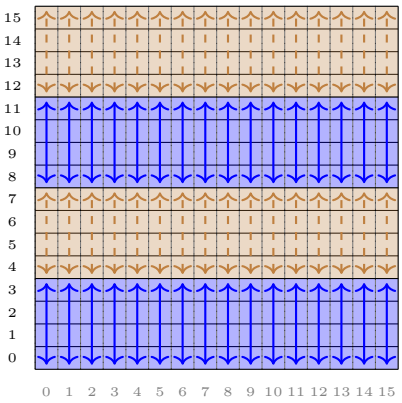
The transformations involved in the round function of SATURNIN are then implemented as follows.

Sbox Layer. The 4-bit Sboxes σ_0 and σ_1 are applied in parallel, each over a half of the whole state. These permutations are such that $\sigma_0 = \pi_0 \circ \sigma$ and $\sigma_1 = \pi_1 \circ \sigma$ where π_0 and π_1 are bit permutations. Their action on the index of the bits in each nibble is given in Table 3.

Table 3: The correspondance between the input and output bit indices in π_0 and π_1.

i	0	1	2	3
$\pi_0(i)$	3	0	1	2
$\pi_1(i)$	2	1	3	0

In the register representation, the Sboxes are applied in a bitsliced fashion over all registers with indices $4i + 0, ..., 4i + 3$, where $i \in \{0, 2\}$ for σ_0 and $i \in \{1, 3\}$ for σ_1. The full parallel application of the Sboxes is denoted S and, in the register representation, it is summarized in Figure 11.



(a) As applied in the registers.

```
#define S_LAYER(a, b, c, d) { \
        a ^= b & c;           \
        b ^= a | d;           \
        d ^= b | c;           \
        c ^= b & d;           \
        b ^= a | c;           \
        a ^= b | d;           \
    }
```

```
#define PI_0(a, b, c, d, tmp) {          \
        tmp = a; a = b; b = c;           \
        c = d; d = tmp;                  \
    }

#define PI_1(a, b, c, d, tmp) {          \
        tmp = a; a = d;                  \
        d = c; c = tmp;                  \
    }
```

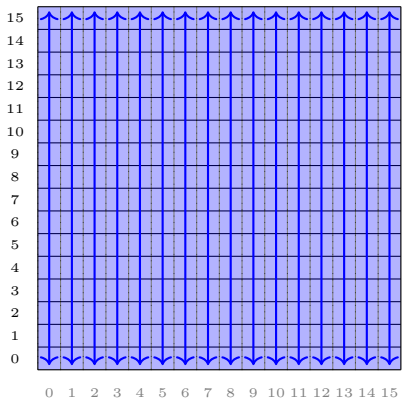
(b) The main Sbox σ and the bit permutations π_0 and π_1.

Figure 11: The application of the 4-bit Sboxes (S) in the register representation. σ_0 is represented by continuous blue arrows, σ_1 by dashed brown ones.

The lookup-tables of the Sboxes are given in Section 2.1.2. They can be implemented in bitslice over 4 words using the C macros in Figure 11b. The bitsliced implementations of the inverse Sboxes are given in Figure 22, in Appendix A.

MDS Matrix. The matrix M is defined over $(\mathbb{F}_{2^4})^4$ and is applied in parallel on each column of the state. In the register representation, it is applied in a bitsliced fashion over the whole state at once as summarized in Figure 12a.

The matrix is one of the low-cost MDS matrices found in [DL18]. It can be implemented in a bitsliced fashion using the C macro in Figure 12b. The bitsliced implementation of its inverse is provided in Figure 23, in Appendix A.



(a) In register representation.

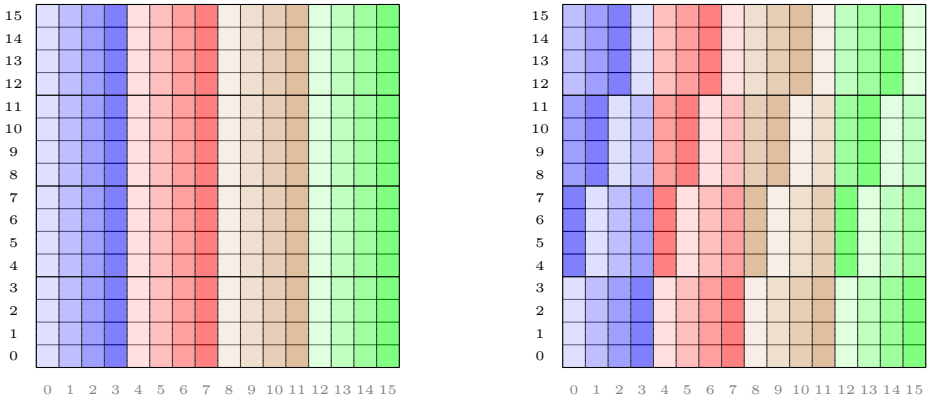
```c
#define MUL(x0, x1, x2, x3, tmp) {                                  \
        tmp = x3; x3 = x2 ; x2 = x1; x1 = x0; x0 = tmp; /* rotation */  \
        x1 ^= x0;                                       /* XOR */       \
    }

#define MDS(x0, x1, x2, x3, x4, x5, x6, x7,                         \
            x8, x9, xa, xb, xc, xd, xe, xf, tmp) {                  \
        x8 ^= xc; x9 ^= xd; xa ^= xe; xb ^= xf; /* C ^= D */        \
        x0 ^= x4; x1 ^= x5; x2 ^= x6; x3 ^= x7; /* A ^= B */        \
        MUL(x4, x5, x6, x7, tmp);               /* B = MUL(B) */    \
        MUL(xc, xd, xe, xf, tmp);               /* D = MUL(D) */    \
        x4 ^= x8; x5 ^= x9; x6 ^= xa; x7 ^= xb; /* B ^= C */        \
        xc ^= x0; xd ^= x1; xe ^= x2; xf ^= x3; /* D ^= A */        \
        MUL(x0, x1, x2, x3, tmp);               /* A = MUL(A) */    \
        MUL(x0, x1, x2, x3, tmp);               /* A = MUL(A) */    \
        MUL(x8, x9, xa, xb, tmp);               /* C = MUL(C) */    \
        MUL(x8, x9, xa, xb, tmp);               /* C = MUL(C) */    \
        x8 ^= xc; x9 ^= xd; xa ^= xe; xb ^= xf; /* C ^= D */        \
        x0 ^= x4; x1 ^= x5; x2 ^= x6; x3 ^= x7; /* A ^= B */        \
        x4 ^= x8; x5 ^= x9; x6 ^= xa; x7 ^= xb; /* B ^= C */        \
        xc ^= x0; xd ^= x1; xe ^= x2; xf ^= x3; /* D ^= A */        \
    }
```

(b) As a C macro.

Figure 12: The parallel application of the 16×16 matrix M, i.e. the operation MC.

$\mathsf{SR}_{\text{slice}}$. In order to mix the columns in each slice, we use $\mathsf{SR}_{\text{slice}}$ which consists in the parallel application of R_{slice} on each slice independently. This operation maps the nibble with coordinates (x, y, z) to $(x + y \mod 4, y, z)$. Its implementation on 16-bit registers is summarized in Figure 13.



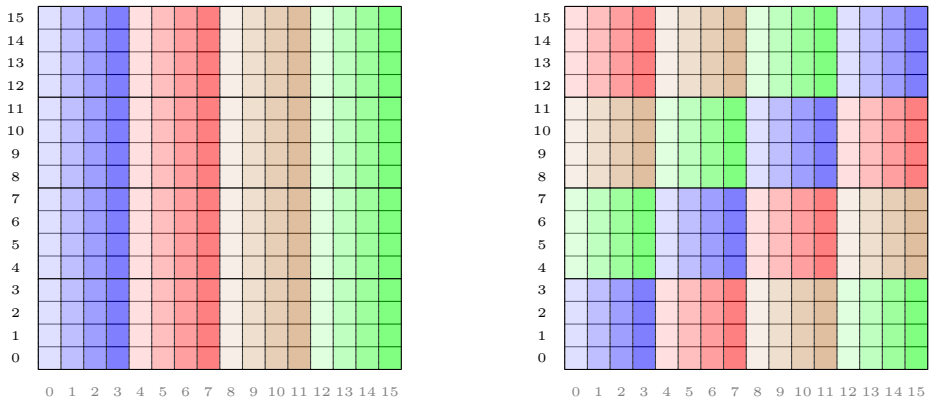
(a) Before. **(b)** After.

Figure 13: The $\mathsf{SR}_{\text{slice}}$ operation that mixes the columns in each slice separately.

It can be implemented by applying to register i a rotation of each 4-bit word in it by $\lfloor i/4 \rfloor$. The following C macros implement the 4 such functions we need:

- the rotation by 1 is `((x & 0x7777) << 1) | ((x & 0x8888) >> 3)`,
- the rotation by 2 is `((x & 0x3333) << 2) | ((x & 0xcccc) >> 2)`, and
- the rotation by 3 is `((x & 0x1111) << 3) | ((x & 0xeeee) >> 1)`.

ShiftRows in a Sheet. We proceed with sheets as we did with slices. In order to mix the columns in each sheet, we use $\mathsf{SR}_{\text{sheet}}$ which consists in the parallel application of R_{sheet} on each sheet independently. This operation maps the nibble with coordinates (x, y, z) to $(x, y, z + y \mod 4)$. It is summarized in Figure 13. It can be implemented with a rotation of the 16-bit register i by $4\lfloor i/4 \rfloor$.



(a) Before. **(b)** After.

Figure 14: The $\mathsf{SR}_{\text{sheet}}$ operation that mixes the columns in each sheet separately.

Key Addition. The key addition is applied in a very straightforward way by XORing 16-bit registers in the key state with their counterparts in the internal state of the cipher. In odd super-rounds, each register of the master key is rotated just before being added, meaning we can use operators combining the XOR and the rotation when available.

Constant Addition. The bits in which the constants RC_0 and RC_1 are XORed correspond to registers 0 and 8 respectively. Hence, this operation is implemented using two word-wise XORs. The 16-bit state of each LFSR naturally corresponds to one 16-bit register, and one clock of each LFSR is easily implemented using the C macros in Figure 15.

```
1 #define CLOCK_LFSR_0(x)   x = ((x) & 0x8000) ? ((x) << 1) ^ 0x002d : (x) << 1 ;
2 #define CLOCK_LFSR_1(x)   x = ((x) & 0x8000) ? ((x) << 1) ^ 0x0053 : (x) << 1 ;
```

Figure 15: Clocking the LFSRs used to derive the round constants.

3.3 Operations Count

In order to evaluate the efficiency of SATURNIN, we can count the number of operations required to encrypt one 256-bit block of plaintext. If we assume an ideal 16-bit instruction set with three-operand instructions, we need the following instructions for one round of SATURNIN:

- 4×12 instructions for the S-Box layer (6 AND/OR, and 6 XOR)
- 38 XOR instructions for the MDS layer
- depending on the round number, either 12 rotation instructions for $\mathsf{SR}_{\text{slice}}$ or $\mathsf{SR}_{\text{slice}}^{-1}$, or 60 instructions (two ANDs for masking, two shifts, and one OR for each of the 12 registers) for $\mathsf{SR}_{\text{sheet}}$ or $\mathsf{SR}_{\text{sheet}}^{-1}$.
- 16 XOR instructions every two rounds for the key addition

In total, with 20 rounds plus the last key addition (and ignoring the key schedule), this makes 2616 instructions to process 256 bits, or 10.2 instructions per bit.

In a hardware setting, the nibble permutations SR and SR^{-1} are free, and we just need the following operations:

- 64×12 gates for the S-Box layer (6 AND/OR, and 6 XOR)
- 16×38 XOR gates for the MDS layer
- 256 XOR gates every two rounds for the key addition

In total, with 20 rounds, this sums up to 30336 gates to process 256 bits, or 118.5 gates per bit. For reference, we can compare this number with results for `SKINNY`, `SIMON`, `NOEKEON` and `AES` given in [BJK+16], even though those ciphers have a smaller state of 128 bits. If we ignore the cost of the key schedule, at the 128-bit security level, `SKINNY` requires 130 gates per bit, `SIMON` requires 136, `NOEKEON` 100, and `AES` 202.5. Focusing on ciphers with a 256-bit key, we need 156 operations for `SKINNY`, 144 for for `SIMON`, and 283.5 for `AES`. This shows that Saturnin is a good candidate for efficient implementations.

3.4 Software Implementations

In the bitslice representation, the state and keys of Saturnin are naturally expressed as sequences of 16-bit registers. However, many small microcontrollers now offer a 32-bit architecture, in particular the ARM Cortex-M line. Bigger architectures may also offer larger registers. In this section, we discuss possible implementation strategies that leverage such larger registers. The strategies are named out of the chosen representation of the Saturnin state. The Saturnin submission package contains C implementations of Saturnin-CTR-Cascade using the `bs32`, `bs32x` and `bs64` representation described in this section (in addition to the reference implementation called `ref`); a `bs32` implementation of Saturnin-Hash is also provided (there again, in addition to the reference implementation called `ref`).

3.4.1 Platform Types

The ARM Cortex-M3 and M4 processors implement the ARMv7-M architecture and offer sixteen 32-bit registers, out of which three are reserved (the program counter, the stack pointer, and the register `r9` which supports thread-local storage in some operating systems). Memory slots cannot be used directly as operands for computations; thus, using RAM to complement registers requires extra load and store operations, which are expensive (two clock cycles per load or store).

A smaller architecture is the ARM Cortex-M0 (and its variant the M0+) which follows the ARMv6-M architecture. It has the same number of available registers, but a reduced instruction set; in particular, most opcodes, such as Boolean bitwise operations, can operate only on the first eight registers. The M0 and M0+ are very popular in new embedded system designs because they have a very low power consumption, and offer a low gate count that allows embedding such cores in small custom designs.

Other popular 32-bit microcontroller lines include MIPS32 and PowerPC systems, which offer more registers (about 30 usable registers) but also have higher gate counts and are, thus, less "lightweight".

At the other end of the spectrum of software platforms are large systems with 64-bit CPU and vector units, in particular modern x86 systems. Such systems are not the primary optimization targets of "lightweight cryptographic algorithms"; however, in contexts where large numbers of small embedded systems use cryptography to communicate with a single or small set of central servers, the performance of cryptographic algorithms on large systems may matter.

3.4.2 bs32

Given the sixteen 16-bit registers (r_0 to r_{15}) of the Saturnin bitslice representation, we define the `bs32` representation as eight 32-bit registers q_0 to q_7, such that the low 16 bits

of q_i contain r_i, and the high 16 bits of q_i contain r_{i+8}.

In this representation:

- The complete state fits on only eight 32-bit registers. This promotes performance on the ARM Cortex systems, which are register-starved.
- In the S-box layer, σ_0 is applied to registers r_0 to r_3, *and* to registers r_8 to r_{11}; by applying the same sequence of Boolean bitwise operations on the registers q_0 to q_3, the two instances of σ_0 are computed in parallel. Similarly, the two instances of σ_1 are performed in parallel by using registers q_4 to q_7.
- In the linear layer MC, the same kind of parallelism occurs; as can be observed in Figure 3, the same operations occur in the left and right parts of the diagram, which maps well to the `bs32` representation; for instance, the multiplication by α on the b value is done at the same time as the multiplication by α on the d value, and, in `bs32` representation, the same 32-bit registers are involved. The only operations that break this parallelism are the transverse XOR (b with c, d with a), which require a rotation of some of the q_i values by 16 bits. Rotation is natively supported by the ARM architecture, and, on ARMv7-M, a rotation can be applied to an operand of a Boolean operation with no extra cost.
- Addition (XOR) of the round constants is done into register r_0 and r_8; in `bs32` representation, this is a single XOR of q_0 with a 32-bit round constant.

On the other hand, $\mathsf{SR}_{\text{slice}}$ and $\mathsf{SR}_{\text{sheet}}$ require more masking and shifting in `bs32` than in a simple flat representation. Such operations occur only on odd rounds, making that extra cost comparatively less important.

We implemented SATURNIN in ARM Cortex-M4 assembly and found the costs, per element of SATURNIN, shown on Figure 16.

Operation	**Cost**	**Copies**	**Total**
S-box layer S	24	20	480
Linear layer MC	19	20	380
$\mathsf{SR}_{\text{slice}}$	44	5	220
$\mathsf{SR}^{-1}_{\text{slice}}$	44	5	220
$\mathsf{SR}_{\text{sheet}}$	28	5	140
$\mathsf{SR}^{-1}_{\text{sheet}}$	28	5	140
XOR with round constants	1	10	10
XOR with (rotated) key	8	11	88

Figure 16: Minimal cycle counts on ARM Cortex-M4 for SATURNIN[10].

These costs do *not* include data movement: they can be achieved only if assuming that all operands are already in appropriate registers. In practice, assuming that the state is kept in CPU registers, then extra memory reads will be needed for the round constants, the key (rotated and non-rotated), the initial load of the block into registers, and the final write of the resulting block.

These values illustrate that the `bs32` representation minimizes the cost of the most used operations (S-boxes and linear layer).

3.4.3 bs32x

SATURNIN-CTR-Cascade offers some parallelism options:

- CTR encryption is inherently parallel: all SATURNIN invocations are independent of each other.
- The Cascade is *not* parallel (processing of a block cannot start before the previous block is finished), but it can run concurrently with the CTR encryption.

Best performance is achieved if two SATURNIN instances can be computed in parallel; in that case:

- during encryption, the SATURNIN instance for one block of ciphertext will be performed in parallel with the processing of the counter for encrypting the next block of plaintext;
- during decryption, the SATURNIN instance for one block of ciphertext will be performed in parallel with the processing of the counter for decrypting that same block of ciphertext.

The `bs32x` representation uses sixteen 32-bit registers w_0 to w_{15}. If we call (r_i) the sixteen 16-bit state values for the first SATURNIN instance, and (s_i) the state values for the second SATURNIN instance, then we "spread" the bits of r_i into the even-indexed bits of w_i, and the bits of s_i into the odd-indexed bits of w_i: for all values of i from 0 to 15, bit j of r_i goes into bit $2j$ of w_i, while bit j of s_i goes into bit $2j + 1$ of w_i.

Compared with `bs32`, the `bs32x` representation has the following consequences:

- The total number of 32-bit Boolean operations for the S-box and the linear layer of the two SATURNIN instances is unchanged; however, the rotations by 16 bits in the S-box are no longer needed.
- The masking and shifting operations in $\mathsf{SR}_{\text{slice}}$ and $\mathsf{SR}_{\text{sheet}}$ are simplified; in particular, $\mathsf{SR}_{\text{sheet}}$ becomes a sequence of twelve rotations of 32-bit registers by 8, 16 or 24 bits.
- Initial loading of the blocks and of the keys, and final write of the blocks, become more expensive.
- Sixteen 32-bit registers are now needed to store the state, which degrades performance on register-starved architectures.

The bit-spreading of a 16-bit value over the even-indexed bits of a 32-bit register can be done in a relatively short sequence of masking and shiftings:

```
x = (x & 0x000000FF) | ((x & 0x0000FF00) << 8);
x = (x & 0x000F000F) | ((x & 0x00F000F0) << 4);
x = (x & 0x03030303) | ((x & 0x0C0C0C0C) << 2);
x = (x & 0x11111111) | ((x & 0x22222222) << 1);
```

In our experiments, the `bs32x` representation does not appear to offer better performance than `bs32` on architectures with a low number of registers (ARM, x86), but it seems to give a slight advantage on register-rich systems (PowerPC).

3.4.4 bs64

The `bs64` is an extended version of `bs32` for 64-bit architectures. It runs two instances of SATURNIN in parallel; the q_0 to q_7 32-bit words of the `bs32` representation for the first instance of SATURNIN are stored in the low halves of eight 64-bit registers, while the high halves are used for the 32-bit words of the `bs32` representation of the state of the second instance of SATURNIN.

The S-box layer naturally extends to the `bs64` representation. On the other hand, more operations are needed for the linear layer and the SR operations. On a 64-bit x86 system,

a `bs64` implementation of SATURNIN-CTR-Cascade appears to be only slighly faster than a `bs32`. More optimization work is needed to assess whether the speed doubling that we may expect from running two instances in parallel can be achieved in practice.

3.4.5 ssse3

On x86 systems with SSSE3 opcodes, an alternate promising representation is to spread the 16 bits of a state word r_i over all 16 *bytes* of an SSE2 register. In such a way, eight parallel instances of SATURNIN can be executed; the instance j uses the j-th bit each of byte. All Boolean bitwise operations then run at the full 128-bit width of SSE2 registers. Moreover, all operations that permute bits within a register r_i (i.e. $\mathsf{SR}_{\text{slice}}$, $\mathsf{SR}_{\text{sheet}}$ and the rotation of key words) can be implemented with the `pshufb` opcode, which applies an arbitrary mapping of bytes from source to destination in a single clock cycle.

Some preliminary tests showed that, on an Intel Core i7-6567U CPU (Skylake core), processing cost may be lowered down to about 7 cpb; use of AVX2 opcodes should theoretically further halve that cost (the `vpshufb` opcode applies the equivalent of `pshufb` to the two halves of a 256-bit AVX2 register in parallel). However, such performance may be achieved only with modes of operation that allow 8-fold parallelism (16-fold for AVX2), which is not the case of SATURNIN-Hash or SATURNIN-CTR-Cascade. Raw SATURNIN-CTR encryption can benefit from such parallelism.

3.4.6 Code Size Considerations

Lightweight cryptographic algorithms are meant to run on platforms that are constrained in several ways. In particular, the total code footprint size may be limited.

The API mandated by NIST for the submitted implementations is compatible with the SUPERCOP API, which is meant for speed benchmarks; however, it is ill-suited to assessment of code footprint, for the two following reasons:

- The SUPERCOP API expects a message to encrypt, decrypt or hash to be provided as a single chunk in RAM; moreover, that chunk should include the authentication tag in the case of AEAD processing. Constrained systems do not necessarily have enough available RAM to use such an internal data structure. Practical implementations often need to perform streamed processing, in which data is handled by chunks. Moreover, the length of each individual chunk may be driven by application requirements and not comply to any particular property (e.g. chunk lengths are not necessarily a multiple of 32 bytes). Stream processing requires additional API support code, which is not present in the SUPERCOP API, but matters for code footprint assessments.

- Applications that use a lightweight cryptographic algorithms may need *both* an AEAD cipher and a hash function. If the AEAD cipher and the hash function are based on the same primitive, part of the code may be shared. The SUPERCOP API does not allow for such code sharing and would thus prevent measuring the code size gains that would result from it.

In order to make a more realistic assessment of the code footprint of a practical, embeddable SATURNIN implementation, we defined a streamable API for both SATURNIN-CTR-Cascade and SATURNIN-Hash, and implemented it in C, ARM Cortex-M4 assembly, and ARM Cortex-M3 assembly. These implementations are provided in the submission package. Figure 17 lists the obtained code sizes and throughput.

The `saturnin_portable.c` implementation is written in portable C code; for the test, it was compiled for the M4 with GCC 7.3.0 with optimization flags "`-O2`". The `saturnin_m4.s` is written in assembly and is meant for the M4. The `saturnin_m3.s` is almost identical to the M4 implementation, except that it uses only opcodes available

Implementation	Code size	Throughput (c/b)		
		Hash	Encrypt	AAD
`saturnin_portable.c`	3956	183	250	128
`saturnin_m4.s`	2948	111	144	75
`saturnin_m3.s`	3028	113	147	77

Figure 17: Size and performance of streamable implementations of SATURNIN-CTR-Cascade and SATURNIN-Hash on ARM Cortex-M4. The code size is expressed in bytes; the processing speeds for hashing, encryption, and additional authenticated data, are given in cycles-per-byte.

on the M3; in practice, this means that it replaces the M4 "DSP" opcodes `pkhbt` and `pkhtb` with sequences of opcodes that are compatible with the M3, leading to a slightly enlarged and slower code. Speed was measured on an ARM Cortex-M4F code (Nordic nRF52832 microcontroller) with an accuracy of about 1%. According to their respective documentations, the M3 and M4 cores are supposed to offer identical instruction timing characteristics[4]; therefore, measures on the M4 are theoretically representative of M3 performance as well.

In `saturnin_m4.s`, the 2948 bytes of code split as follows:

- SATURNIN block cipher (encrypt direction only): 1694 bytes.
- Round constants for SATURNIN-CTR-Cascade: 200 bytes.
- Round constants for SATURNIN-Hash: 128 bytes.
- API support for SATURNIN-CTR-Cascade: 606 bytes.
- API support for SATURNIN-Hash: 372 bytes.

The API support for SATURNIN-CTR-Cascade and SATURNIN-Hash share two internal functions, that amount to 52 bytes of code.

4 Rationale

4.1 General Structure using the Super-Sbox Representation

The main motivation behind the general structure of SATURNIN is to mimic the structure of the AES, which is a well-understood structure. Indeed, the Super-Sbox view of SATURNIN is very similar to an AES operating on 16-bit words, instead of bytes. Let us denote by S_{16} the 16-bit Super-Sbox in SATURNIN, i.e., the permutation of $\mathbb{F}_2^{16}$, composed of the succession of an Sbox layer, the linear function M, and a second Sbox layer. The composition of two rounds of indices $(2t, 2t+1)$ can then be seen as the application of this Super-Sbox to each column of the internal state, followed by $\mathsf{SR}_{2t+1}^{-1} \circ \mathsf{MC} \circ \mathsf{SR}_{2t+1}$. If t is even, then the slices are invariant under SR_{2t+1}. This implies that the linear layer $\mathsf{SR}_{2t+1}^{-1} \circ \mathsf{MC} \circ \mathsf{SR}_{2t+1}$ consists of the concatenation of four copies of the same function L_{64} of $(\mathbb{F}_2^{16})^4$, which applies to the slices independently.

If t is odd, then the sheets are invariant under SR_{2t+1}. In this case, the linear layer consists of the concatenation of four copies of the same function L_{64}, which applies to the sheets independently. Moreover, it is easy to prove, e.g. by Theorem 1 in [ADK+14] that L_{64} has branch number 5 with respect to $\mathbb{F}_2^{16}$.

[4]Except for the integer multiplication and division opcodes, which are not used in our SATURNIN implementations.

Let us then represent in a 4×4 matrix C the 16-bit words $\mathsf{C}_{0,0}, \ldots, \mathsf{C}_{3,3}$ corresponding to the 16 columns of the cube, where $\mathsf{C}_{i,j}$ corresponds to the column defined by $x = i, z = j$. This means that each slice in the cube is a column of Matrix C, while each sheet in the cube is a row of C. When t is even, the linear function L_{64} then applies to the columns of C independently, while for odd t, it applies to its rows. In other words, a super-round has the following structure: the Sbox S_{16} is applied to each 16-bit word, then L_{64} is applied to the four columns of C, and Matrix C is transposed.

SATURNIN is then very similar to an AES operating on 16-bit words, except that the ShiftRows transformation is here replaced by a transposition exactly as it was in SQUARE, the predecessor of the AES [DKR97]. In the following, this super-Sbox view and the previous notation will be used for analyzing the resistance of SATURNIN to the main classes of attacks.

4.2 On the Building-blocks in the Block Cipher

4.2.1 On the Number of Rounds

The number of rounds has been determined by the security analysis, which shows that, for most attacks, a super-round in SATURNIN offers a resistance similar to a single round in the AES. Therefore, 10 super-rounds, i.e. 20 rounds, appears to be a natural choice.

4.2.2 On the MDS Matrix M

The main building-block in the linear layer is the 4×4-MDS matrix M over $\mathbb{F}_{2^4}$. This matrix is one of the MDS matrices exhibited in [DL18] with the lowest known implementation cost. Its Feistel-like structure guarantees that its inverse also has a low implementation cost.

Moreover, this MDS matrix applies similar operations to a and c (and to b and d), see Figure 3. This allows an efficient 32-bit implementation, where the 16 bits corresponding to a and the 16 bits corresponding to c are stored in the same register (respectively the 16 bits corresponding to b and the 16 bits corresponding to d).

4.2.3 On the Design of the Super-Sbox and the Choice of σ

The two Sboxes σ_0 and σ_1, which apply to the nibbles with an even index and with an odd index respectively, are both chosen as the composition of the same Sbox, σ, followed by two different permutations of the output bits (see Table 3).

Choice of σ. The 4-bit Sbox σ has been chosen among the 4-bit Sboxes with optimal cryptographic parameters, i.e., in one of the equivalence classes named *optimal* in the classification established by Leander and Poschmann [LP07]. These Sboxes are those with differential uniformity 4 and linearity 8 such that all nonzero linear combinations of their coordinates have degree 3. Moreover, we chose for σ an Sbox minimizing the number of operations in the bitslice implementation, both for σ and σ^{-1} in order to compute efficiently the inverse cipher for decryption with efficient 32-bit implementation, where the 16 bits corresponding to a and the 16 bits corresponding to c are stored in the same register (respectively the 16 bits corresponding to b and the 16 bits corresponding to d).

4.2.4 On the Design of the Super-Sbox and the Choice of σ

The two Sboxes σ_0 and σ_1, which apply to the nibbles with an even index and with an odd index respectively, are both chosen as the composition of the same Sbox, σ, followed by two different permutations of the output bits (see Table 3).

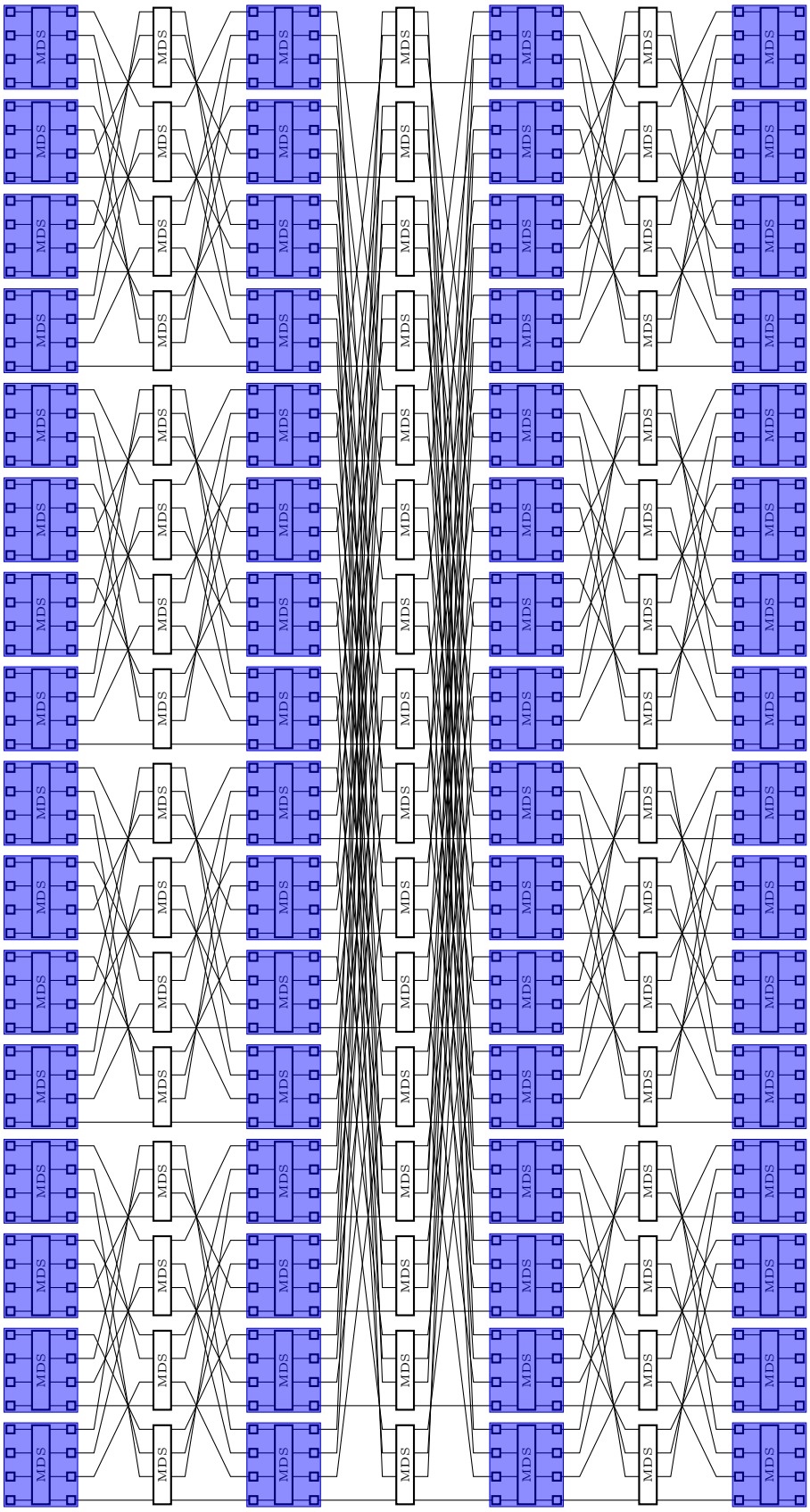


Figure 18: Structure of 8 rounds of SATURNIN, with the Super-Sboxes.

Choice of σ. The 4-bit Sbox σ has been chosen among the 4-bit Sboxes with optimal cryptographic parameters, i.e., in one of the equivalence classes named *optimal* in the classification established by Leander and Poschmann [LP07]. These Sboxes are those with differential uniformity 4 and linearity 8 such that all nonzero linear combinations of their coordinates have degree 3. Moreover, we chose for σ an Sbox minimizing the number of operations in the bitslice implementation, both for σ and σ^{-1} in order to compute efficiently the inverse cipher for decryption with SATURNIN-Short. We used a strategy inspired from [UDI+11] and searched, among all Sboxes with the lowest possible implementation cost, for those having the required cryptographic properties. It is worth noticing that the list of low-cost Sboxes computed by Ullrich *et al.* could not be used in our case. Indeed, since it is restricted to all Sboxes which could be implemented with at most 13 instructions, it does not contain any Sbox having all its components of degree 3. Also, we need to minimize the implementation cost of both the Sbox and its inverse, which makes the approach different from [UDI+11]. Instead, we chose to search for Sboxes with a Feistel-like structure, which can be easily inverted. The best such Sbox we could find requires 6 XOR, 6 nonlinear instructions, corresponding to a total of 12 instructions (with three operands). Its inverse has a similar implementation cost.

Design of S_{16}. A counterpart of the nice implementation properties of the MDS matrix M is that it transforms the subspace of $\mathbb{F}_{2^4}$ defined by $\{(x, x, 0, 0), x \in \mathbb{F}_{2^4}\}$ into the subspace $\{(y, y, 0, y), y \in \mathbb{F}_{2^4}\}$. This implies that, if the nonlinear layer in S_{16} uses four copies of the same Sbox σ, then S_{16} transforms the affine subspace of dimension 4 $\{(x, x, \sigma^{-1}(0), \sigma^{-1}(0)), x \in \mathbb{F}_{2^4}\}$ into the affine subspace $\{(y, y, \sigma(0), y), y \in \mathbb{F}_{2^4}\}$. We considered the propagation of such a 4-dimensional subspace, of a very simple form, as an unsuitable property. Moreover, this particular structure also explains why the Super-Sbox S_{16} based on σ only has a linearity equal to 2^{12} which is higher than expected. For these two reasons, we decided to use two slightly different Sboxes, one applied to the nibbles with an even index, and the other one applied to the nibbles with an odd index. We made an exhaustive search over all pairs of bit permutations (π_0, π_1) of the four output bits, and studied the Super-Sbox S_{16} derived from $\pi_0 \circ \sigma$ and $\pi_1 \circ \sigma$. The smallest differential uniformity that is obtained for these Super-Sboxes is equal to 80, and the smallest linearity (i.e., the highest magnitude of the Walsh transform) is equal to 3072. They are achieved simultaneously for two pairs (π_0, π_1). Among these two possibilities, we chose the one leading to the Super-Sbox with the lowest number of short cycles: most notably, for our choice, S_{16} has one fixed point (the all-zero word) and two cycles of length 6, while for the other choice, S_{16} would have one fixed point, two cycles of length 2, one cycle of length 3, three cycles of length 4, one cycle of length 5 and two cycles of length 6. This motivated the choice of permutations π_0 and π_1 defined in Table 3: if $\sigma(x_0, x_1, x_2, x_3) = (y_0, y_1, y_2, y_3)$, then $\sigma_0(x_0, x_1, x_2, x_3) = (y_1, y_2, y_3, y_0)$ and $\sigma_1(x_0, x_1, x_2, x_3) = (y_3, y_1, y_0, y_2)$.

4.2.5 On the Key Schedule and Constant Addition

Since the key is aligned with the 16-bit structure of the cipher, we can reuse the analysis done on the AES to evaluate the security against related-key attacks. In particular, using a simple MILP model, we get the following lower bounds on the number of active Super-Sboxes in a related-key differential trail:

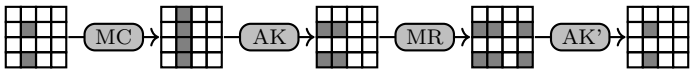
n (super-rounds)	1	2	3	4	5	6	7	8	9	10
Active Super-Sboxes	0	1	5	10	12	16	18	22	24	28

Note that these are just lower bounds for truncated trails. In practice, many paths could be impossible to instantiate with concrete differences; indeed in the truncated model, two

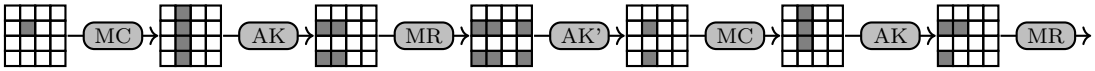
differences can always cancel out, but in a real trail, the constraints can be incompatible. These bounds are slightly better than the bounds given by the AES key schedule.

An example of iterative trail with 6 active Super-Sboxes every two super-rounds is given below, with MR denoting a super-round mixing inside the sheets (*i.e.* the rows in the matrix representation) and MC a super-round mixing inside the slices (*i.e.* the columns in the matrix representation), and AK and AK′ the key additions with the following key differences:

$$\delta\mathsf{K} = \begin{cases} \text{if } r = 2t+3 \text{ (master key K)} \\ \text{if } r = 2t+1 \text{ (rotated key K}') \end{cases}$$

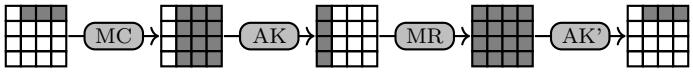


The first and last super-rounds can be optimized to remove one active Super-Sbox, for instance, with 4 super-rounds:



Therefore, this gives a trail with $3r-2$ active Super-Sboxes for r super-rounds (with r even), which reaches the bound given above (in this example, we have $1+4+2+3$ active Super-Sboxes).

This is close to optimal for a linear key schedule, because there are simple trails where the full key is active with 7 active Super-Sboxes every two super-rounds, such as:



Round Constants. The sequence of round constants is generated by two 16-bit LFSRs run in parallel from the same seed. This aims at an efficient hardware implementation, while the whole sequence can be stored in software environments. Both feedback polynomials are primitive polynomials of degree 16 with the smallest number of monomials. They correspond to the first pair of primitive polynomials with 5 monomials in lexicographical order satisfying the following property: let us denote by $(x_t(b))_{0\leq t<512}$ (resp. $(y_t(b))_{0\leq t<512}$) the binary sequence formed by the concatenation of the successive values of RC_0 (resp. RC_1) from the seed defined by the 9-bit integer b (corresponding to the concatenation of the domain separator and the number of super-rounds). Then, in the round-constant sequences generated from the 2^9 seeds, all sets of 16 consecutive pairs (x_t, y_t) are distinct. In other words, there is no pair of distinct 9-bit integers b and b' for which there exists $0 < t_0 < 496$ such that $x_t(b) = x_{t+t_0}(b')$ and $y_t(b) = y_{t+t_0}(b')$ for all $t \geq 0$.

4.3 On Modes of Operation

The different modes proposed, Saturnin-CTR-Cascade, Saturnin-Short and Saturnin-Hash, are intended to provide quantum security against chosen message superposition attacks and superposition verification queries (IND-qCCA security), additionally to classical and quantum security against chosen message attacks and verification queries (IND-CCA). For authenticated encryption, we do not claim security in nonce-misuse scenarios or against the even stronger model of nonce-superposition attacks. To the best of our knowledge, all quantum attacks on classically unbroken MACs that have been reported (see e.g. [KLLN16a]) use chosen-plaintext queries only.

4.3.1 Generic composition.

One of the most natural way to design an authenticated encryption scheme is to combine an encryption scheme and a MAC. The seminal work of Bellare and Namprempre [BN08] studied several composition methods in the classical setting: MAC-then-encrypt (used in TLS 1.0), encrypt-and-MAC (as used in SSH), and encrypt-then-MAC (used in IPSec). They showed that in the classical context, the encrypt-then-MAC construction offers the strongest security: it is generically secure against chosen-plaintext and chosen-ciphertext attacks (IND-CCA) assuming that the encryption scheme is secure against chosen-plaintext attacks (IND-CPA), and the MAC is unforgeable (SUF-CMA). In addition, the encrypt-then-MAC composition allows to reject forgeries quickly without even decrypting the ciphertext. This also reduces the risk of inadvertently releasing an unverified plaintext.

In the quantum setting, Soukharev, Jao and Seshadri have revisited these results [SJS16], and proved that the encrypt-then-MAC composition offers IND-qCCA security, assuming that the encryption scheme is IND-qCPA, and the MAC is SUF-qCMA.

4.3.2 Quantum Security of MACs.

To the best of our knowledge, there are two main references for the quantum security of MACs. In [BZ13a], the authors adopt the following game for existential unforgeability under quantum chosen-plaintext queries: the adversary makes a certain number q of (superposition) chosen-plaintext queries, then she is required to output $q + 1$ valid tags. They argue that, provided that the MAC is a quantum-secure PRF, it is also secure in this setting. In [AMRS], the authors show some shortcomings in this previous approach and introduce the notion of Blind Unforgeability (BU), which is a quantum extension of EUF-CMA.

A "blind forgery" experiment consists in the following game: the adversary selects a parameter ε. The challenger generates a key k and a random "blinding" of the MAC, B_ε, which is an ε-fraction of the message space on which the adversary can query the MAC, while the rest is forbidden to her. The adversary, having only access to the blinded version of the MAC, must then produce a forgery of a message $m \in B_\varepsilon$. This game essentially reduces to EUF-CMA when the adversary makes only classical queries.

The authors of [AMRS] show that a quantum-secure pseudorandom function is a BU-secure MAC. This is why our quantum security arguments below are focused on the security as a quantum PRF. Unfortunately, there does not seem to exist in the literature any provable security claims against quantum superposition verification queries.

4.3.3 SATURNIN-CTR-Cascade

Choice of the Counter Mode. Classically, if the underlying block cipher is a PRP, the Counter mode is secure up to the birthday bound (2^{128} in our case), provided that, for any fixed key, the inputs of block cipher are never reused [Nat01]. This last condition is obviously guaranteed by choosing as input to the block cipher the concatenation of the 128-bit nonce and of a 128-bit counter.

In [ATTU16], a generic proof of quantum security is provided for primitives that XOR the message to a pseudo-random sequence generated from the length of the message and a random secret key. If the construction is IND-CPA, i.e. indistinguishable under (classical) chosen-plaintext attacks, then it becomes IND-qCPA, i.e. indistinguishable under quantum (superposition) chosen-plaintext attacks. The proof is actually very simple. If we take the Counter mode of Figure 6, we observe that, the messages being only XORed to the keystream, a superposition query oracle can be easily emulated using a classical one (by querying the all-zero sequence and XORing the inputs), hence any superposition attack translates to a classical one. As a consequence, the Counter mode remains quantumly secure up to the *classical* birthday bound, which is not the case for other modes of operation

(e.g. the CBC mode of encryption, which seems to achieve only a quantum security up to the *quantum* birthday bound). This brings us to the quantum complexity of a full quantum key-recovery on SATURNIN, showing that this mode suits perfectly our needs.

Choice of the Cascade Construction. We define the compression function $h(k, m) = \text{SATURNIN}_k(m) \oplus m$, where SATURNIN is used with a domain separator equal to 2,3, 4 or 5 (but for simplicity, we do not consider the different domain separators, nor associated data). The Cascade construction builds a function H from h by:

$$H(k, m_0, \dots x_\ell) = h(\dots h(h(k, m_0), m_1) \dots m_\ell)) \ .$$

Classically, the Cascade construction, and NMAC, enjoy proofs of security in [BCK96] and [Bel15].

Quantum Security of Cascade. In [SY17], the NMAC (and HMAC) constructions are shown to be quantum-secure if the underlying compression function $h(k, m)$ is a quantum-secure PRF. The precise definition is Definition 2.5 in [SY17]. When k is chosen uniformly at random, h should be indistinguishable from a random function to a quantum adversary making superposition queries (in x).

Assume that SATURNIN is a quantum-secure PRP; in other words, that given a fixed secret key k, an adversary has a negligible advantage in distinguishing SATURNIN under key k from a random permutation (even with quantum superposition encryption and decryption queries). Then it is difficult to distinguish $h(k, m)$ from a random function. Indeed, let A be a distinguisher for $h(k, m)$, we can transform it into a distinguisher B for $E_k(m)$ from random as follows: B calls A. Whenever A queries its oracle, we XOR m to the result and return. If the oracle is a random function, feedforwarding keeps the output random.

In [Zha15], it is shown that quantumly, random functions are indistinguishable from random permutations up to the quantum birthday bound (in other words, a distinguisher actually outputs a collision), which is $2^{n/3}$ for n-bit to n-bit functions, and 2^{85} queries in our case, as well as 2^{85} quantum memory. The advantage of an adversary of solving this problem with q queries is $C \times \frac{q^3}{2^n}$ for some constant C.

More precisely, Theorem 5.1 in [SY17] shows that the Cascade construction is a quantum secure PRF (if we fix the number ℓ of message blocks as a constant). The advantage of a quantum adversary in breaking it is $34\ell q^{3/2}\sqrt{A}$, where A is the advantage of an adversary making at most $4q$ queries to break the PRF h. Hence in our case: $34\ell q^{3/2}\sqrt{A} = C\ell q^{3/2}\sqrt{\frac{q^3}{2^n}} = C\ell\frac{q^3}{2^{n/2}}$, where C is a constant.

This proof seems not tight, and no quantum attack on NMAC or Cascade achieving better advantage than $\frac{q^3}{2^n}$ is currently known. Furthermore, the exponential number of queries forbids a direct application of attacks based on Simon's algorithm.

4.3.4 SATURNIN-Short

In [BR00], the authors prove the security (assuming a strong PRP) of an encode-then-encrypt construction. SATURNIN-Short can be seen as such, where the encoding corresponds to appending the nonce N, which is transmitted with the message. With small modifications, confidentiality and authenticity come from Theorems 4.1 and 4.2 in [BR00].

Quantumly, thanks to [AMRS], the security as a PRF implies the security against forgeries in the BU game (with quantum superposition chosen-plaintext queries).

4.3.5 SATURNIN-Hash

Classically, the security of Merkle-Damgård is related to that of the compression function. Ours uses the Matyas–Meyer–Oseas (MMO) mode, similar to the Cascade: $h_{i+1} = \text{SATURNIN}_{h_i}(m_i) \oplus m_i$. It is dual to the Davies–Meyer construction, which injects the message block m_i into the key, and would give rather $h_{i+1} = \text{SATURNIN}_{m_i}(h_i) \oplus h_i$.

In [Zha18], Zhandry proves the quantum indifferentiability of the Merkle-Damgård construction, provided that the underlying compression function is a random function and that a prefix-free encoding (hence a good padding) is used. We saw above that SATURNIN with feedforward, under the assumption that SATURNIN is a random permutation, is quantumly indistinguishable from a random function (up to the quantum collision bound), hence this proof applies. Theorem 5.2 in [Zha18] gives a probability of success $O(q^4 2^{-n/2})$ for any adversary making at most q queries. As above, there is currently no quantum attack on the Merkle-Damgård construction and the exponential term forbids a direct application of quantum period-finding.

There is further work on the quantum security of iterated hash constructions, for example Merkle-Damgård with Davies-Meyer in [HY18]. We reckon that a proof such as [HY18] could be done also with the MMO mode.

Finding a (second) preimage on a hash function generically can be done using Grover's algorithm, halving the level of quantum security with respect to the classical one. In this setting, there is no notion of query limitation, as the adversary has all the power to implement the hash function and query it in an off-line manner. For collision search, the power of a quantum adversary depends on the amount of quantum memory it can use (assuming a single processor model). We obtain in Section 5 the tradeoff curve $\mathcal{T}^5 \times \mathcal{M}_q = 2^{512}$, where $\mathcal{M}_q$ denotes quantum memory.

5 Security Analysis

5.1 Security of the Block Cipher against Classical Attacks

As explained in Section 4.1, the structure of the SATURNIN block cipher is very similar to the structure of an AES operating on 16-bit words. The SATURNIN block cipher then benefits from the 20-year cryptanalytic effort against the AES. Most notably, the main families of attacks against the AES can be directly transposed to SATURNIN, by replacing the AES 8-bit Sbox by the SATURNIN 16-bit Super-Sbox. Due mainly to SATURNIN's simplified key-schedule, we have been able to improve the highest number of attacked rounds with respect to AES, from 7 AES-rounds to 7.5 super-rounds from SATURNIN. Improved best attacks exploiting further this key-schedule might reach up to 8 super-rounds (which would already be an impressive result), but more than that seems extremely unlikely from our preliminary analysis, when considering the known cryptanalysis tools.

Differential cryptanalysis. The Super-Sbox S_{16} has differential uniformity 80, or equivalently the highest probability for a non-trivial differential is $80 \times 2^{-16} = 2^{-9.68}$. The AES structure guarantees that any four consecutive rounds have at least 25 active Super-Sboxes. This implies that the best differential characteristics over 4 super-rounds and 8 super-rounds have probability at most

$$\left(\frac{80}{2^{16}}\right)^{25} = 2^{-241.9} \text{ and } \left(\frac{80}{2^{16}}\right)^{50} = 2^{-483.9}$$

respectively. Moreover, it is worth noticing that the proportion of differentials for the Super-Sbox with probability higher than 2^{-10} is very small since there are only 110 such differentials (among 2^{32}); moreover, all these 110 differentials have exactly five active

nibbles. More precisely, the differential spectrum of the super-Sbox (i.e., the number of entries in its DDT with their multiplicities) is given in Table 4.

Table 4: Differential spectrum of the Super-Sbox. This corresponds to the list of all nonzero values in the DDT of the Super-Sbox, and the number in bracket is the number of occurrences (excluding the value corresponding to the trivial differential (0,0)).

2 [1267010382]	4 [329513419]	6 [56414502]	8 [10192982]	10 [1143106]
12 [283372]	14 [20740]	16 [347147]	18 [22466]	20 [24586]
22 [927]	24 [6530]	26 [164]	28 [276]	32 [25871]
34 [1331]	36 [2018]	38 [40]	40 [476]	42 [4]
44 [26]	46 [2]	48 [84]	50 [8]	52 [8]
64 [858]	66 [32]	68 [68]	72 [4]	80 [6]

Linear cryptanalysis. The *linearity* of the Super-Sbox is defined as the highest magnitude taken by its Walsh transform:

$$\widehat{\mathsf{S}_{16}}(\alpha,\beta) = \sum_{x\in\mathbb{F}_2^{16}} (-1)^{\beta\cdot\mathsf{S}_{16}+\alpha\cdot x}, \quad \alpha,\beta\in\mathbb{F}_2^{16}, \beta\neq 0\ .$$

The *linearity* of the Super-Sbox is equal to 3072. By the same arguments as previously, we deduce that the highest squared correlation for a linear trail over 4 rounds and over 8 rounds is at most

$$\left(\frac{3072^2}{2^{32}}\right)^{25} = 2^{-220.7} \text{ and } \left(\frac{3072^2}{2^{32}}\right)^{50} = 2^{-441.5}$$

respectively. The magnitudes of the entries in the LAT of the Super-Sbox are given in Table 5.

Table 5: Walsh spectrum of the Super-Sbox. This corresponds to the list of all nonzero magnitudes of the Walsh transform of the Super-Sbox (or equivalently of all nonzero entries in its LAT), and the number in bracket is the number of occurrences (excluding the value corresponding to the trivial value (0,0)).

64 [832807463]	128 [758861540]	192 [644238519]	256 [517925648]	320 [388234114]
384 [275518253]	448 [182759798]	512 [115136927]	576 [67670502]	640 [37783009]
704 [19670427]	768 [9849332]	832 [4623640]	896 [2118213]	960 [971052]
1024 [503300]	1088 [255426]	1152 [131566]	1216 [73760]	1280 [46457]
1344 [22626]	1408 [11984]	1472 [6569]	1536 [4611]	1600 [2829]
1664 [1459]	1728 [563]	1792 [2043]	1856 [99]	1920 [5482]
1984 [127]	2048 [10869]	2112 [138]	2176 [5606]	2240 [12]
2304 [1673]	2432 [470]	2560 [298]	2688 [84]	2816 [16]
2944 [2]	3072 [12]			

Algebraic degree. The choice of the 4-bit Sboxes σ_0 and σ_1 guarantees that all components of the Super-Sbox (i.e., all non-trivial linear combinations of its coordinates) have degree 9. The same property also holds for the inverse function, S_{16}^{-1}, whose components have degree 9. It has been shown in [BCD11] that the degree of an SPN does not increase

as fast as expected because of the structure of its Sbox layer, which is composed of several transformations operating on a smaller number of variables. More precisely, when composing an Sbox layer $F = (S, \ldots, S)$ with another function, an upper bound on the resulting degree can be derived from the following quantity, which characterizes the Sbox:

$$\gamma(S) = \max_{1 \leq i < m} \frac{m-i}{m-\delta_i}$$

where m is the size of the Sbox S and δ_i is the maximum degree of the product of i coordinates of S. When considering the SATURNIN Super-Sbox S_{16}, this quantity $\gamma(\mathsf{S}_{16})$ can be derived from the degree of S_{16}^{-1}. Indeed, it is known from [BC13] that, for any m-bit permutation S, the smallest i such that $\delta_i \geq m-1$ equals $(m - \deg S^{-1})$. Here, we deduce that, for S_{16}, we have $\delta_7 = 15$ and $\delta_6 \leq 14$. It follows that, for all $i \leq 6$,

$$\frac{m-i}{m-\delta_i} \leq \frac{15}{2}$$

and for all $i \geq 8$,

$$\frac{m-i}{m-\delta_i} \leq \frac{8}{1} .$$

We obtain that

$$\gamma(\mathsf{S}_{16}) = \frac{16-7}{16-15} = 9 .$$

We then use Theorem 2 in [BCD11] for upper-bounding the degree of two super-rounds (without the last linear layer which does not influence the degree), considered as the composition of four similar permutations S_{64} operating on 64 bits. Each of these permutations can be decomposed as a Super-Sbox layer and a transformation of degree 9. We then deduce that S_{64} (and then two super-rounds) has degree at most

$$64 - \frac{64-9}{\gamma(\mathsf{S}_{16})} = 57 .$$

Provided that this upper bound is tight both for S_{64} and its inverse, we can recursively apply the same arguments for proving that $\gamma(\mathsf{S}_{64}) = 57$ and that four super-rounds have degree at most

$$256 - \frac{256-57}{\gamma(\mathsf{S}_{64})} = 252 .$$

It follows that the full degree in SATURNIN is reached after at least five super-rounds. These results are summed up on Table 6.

Table 6: Upper-bound on the algebraic degree of SATURNIN when the number of super-round varies.

r (super-rounds)	1	2	3	4	5
degree	9	57	233	252	255

It is worth noticing that this general upper-bound is known to provide a good estimate of the exact degree of the primitive, as shown by the experiments on Keccak for instance.

Bicliques. The exhaustive search attack with bicliques [BKR11] always allows to gain a small factor against the baseline exhaustive key search, by testing all the keys faster than an evaluation of the cipher. It is applicable to SATURNIN, as to any cipher. However, with this biclique exhaustive search, it is still impossible to retrieve the correct key in less than 2^{224} *operations*, not contradicting our security claims.

Impossible Differential Attacks. Impossible differential attacks were introduced by Knudsen and by Biham, Biryukov and Shamir [BBS99]. They form one of the cryptanalysis families that provide some of the best known attacks on reduced-round AES [BLNS18], together with the Demirci-Selcuk MITM attacks. They are designed by first writing an impossible differential transition, which yields a distinguisher for some middle rounds. With partial key guesses, the differential path can be extended forwards and backwards. Good guesses of the keys are such that this path cannot occur. Bad guesses make this path occur as soon as we try enough plaintext-ciphertext pairs, as the middle rounds are replaced by a random permutation. Hence, the attacker first retrieves a set of sufficiently many plaintext-ciphertext pairs with good input and output differences, and he uses these pairs to sieve the subkey space, by trying partial encryptions and decryptions, and removing the pairs that yield the impossible differential. To date, the best impossible differential attack on AES-128 [BLNS18] targets 7 rounds and it requires, with 2^{105} chosen plaintexts, a time of $2^{106.88}$ and memory of 2^{74}. We estimate that a similar impossible differential attack can be applied to 7 super-rounds of Saturnin, with twice these complexity exponents. As the key-schedule of Saturnin is simpler than that of the AES, it may be possible to extend this attack to 7.5 super-rounds, as it is the case for DS-MITM attacks, or even 8 super-rounds, which would be a very impressive result.

Figure 24 in Appendix C presents some of the impossible differential distinguishers that we have studied. When trying to improve the number of rounds with respect to AES, we considered extending the path A two super-rounds backwards and 1.5 forward, but we did not manage to find any configuration that did not involve the whole key, while keeping the number of needed pairs to test lower than 2^{256}. Please note that other configurations with 3 to 1 states in the middle of the impossible would rapidly lead to the full key being involved in the part that activates the whole state. The path B considers a part of the cipher shifted of one super-round with respect to path A. Though we have not managed yet, regarding the key schedule relations, this second path might be more promising to increase the number of attacked super-rounds by 0.5 or 1.

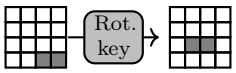
Demirci-Selçuk Meet-in-the-Middle Attack. The DS-MITM attack [DS08, DFJ13] yields the best reduced-round single-key recoveries on AES to date. There exists a variant for each key length of AES. Indeed, the attack works by guessing parts of the internal state; hence the ratio between the state size and the key size is important, and also, the key-schedule relations in AES depend on the key size. In our case, the key size is equal to the block size, so Saturnin is more analogous to AES-128 than the others. The best DS-MITM attack on AES-128 reaches 7 rounds, runs in data 2^{105}, time 2^{105} and memory 2^{81}. It works using a distinguisher on the middle rounds with the following property: given a constrained input and output differential, the values of the internal states can only take few possibilities; if we make the input vary, the sequence of outputs stays in a limited subspace. It is possible to tabulate all these possibilities (in the simple AES-128 attack, there are 2^{80} of them). The adversary finds pairs with some input-output difference. She guesses part of the key, allowing a pair to satisfy the whole inner differential path, and checks whether the middle property is satisfied for this pair. This fails for all subkey guesses, except the right one, since the middle property is very constrained. The 7-round attack works against Saturnin, with twice the complexity exponents. In the next section we present an improved DS-MITM on **7.5 super-rounds**. It is, as could be expected, the best known cryptanalysis on a reduced-round version of Saturnin.

Subspace trails. Until recently, all known distinguishers on the AES in the single-key model could reach at most 4 rounds. However, since 2016, the first 5-round AES-distinguishers appeared [SLG+16, GRR17, RBH17, Gra18]. Most notably, these distinguishers led to improved attacks on reduced-round versions of the cipher, like the attack

on 5 rounds described in [BDK+18] based on the distinguisher exhibited in [Gra18]. The main ingredient of these distinguishers is the existence of subspace trails, i.e., of two linear subspaces U and V of states such that the image by the round function of any coset of U is included in a coset of V. The previously mentioned results on the AES exploit such subspace trails over two rounds of the cipher. The length of the longest subspace trail for a cipher is therefore an essential quantity in these attacks. However, it has been shown in [LTW18] that, if the Sbox does not have any linear structure (i.e. any component with a constant differential), then all subspace trails are the direct product of subspace trails of the single Sbox, i.e. in our case, the direct product of 16 subspaces, each of them being either $\{0\}$ or $\mathbb{F}_2^{16}$. It can easily be checked that the Super-Sbox S_{16} does not have any linear structure, implying that the previous property is valid. Then, SATURNIN behaves exactly as the AES with respect to subspace trails: the fact that the linear layer in the Super-Sbox representation consists of the multiplication of each column in Matrix C by an MDS matrix over $\mathbb{F}_2^{16}$ implies that the longest subspace trail has a length corresponding to two super-rounds and is obtained by considering as input linear space a collection of columns of the matrix (i.e., a collection of sheets in the cube representation). It follows that any distinguisher or attack exploiting this property cannot reach more than five super-rounds.

5.2 DS-MITM attack on 7.5-round SATURNIN

As previously said, due to its simpler key-schedule, the DS-MITM attack can be slightly improved on SATURNIN compared to the AES with transposition. We rewrite SATURNIN as a 4×4 square of 16-bit "supernibbles" which correspond to columns in the cube. We write the Super-Sbox as S, which is actually an Sbox, followed by MixColumns, followed by another Sbox. In even rounds, the key is XORed to the internal state; in odd rounds, it is rotated by 5 supernibble-positions (they correspond to the 20 nibble-positions by which the key is rotated in the cube representation of SATURNIN).



In the differential path that we use for 7.5-round SATURNIN, the plaintexts are active in a single line, while the ciphertexts are active in three lines. We will in total guess 15 supernibbles of the key, corresponding to the intersection of three lines and three columns. In the following, for more clarity, we number the supernibbles with the standard AES byte numbering. The sequence of states of round i is denoted x_i (after adding the key), y_i (after the Super-Sbox), z_i (after transposition) and w_i (after Super-Mixcolumns).

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

Figure 19: AES byte ordering

The attack uses the following property (see Figure 20), which is a variant of the usual middle-rounds property for the DS-MITM attacks on AES: if we are given a plaintext-ciphertext pair such that in y_1, only the nibble 0 is active and, in w_4, only the nibbles $0, 4$ are active, then if we make $y_1[0]$ take a sequence of 2^4 arbitrary differences and obtain the corresponding 32-bit differences in $w_4[0, 4]$, they can only take up to $2^{16\times(4+8+2+1)} = 2^{240}$ values among $2^{32\times 2^4}$. This allows to efficiently distinguish between 4 AES rounds and a random path.

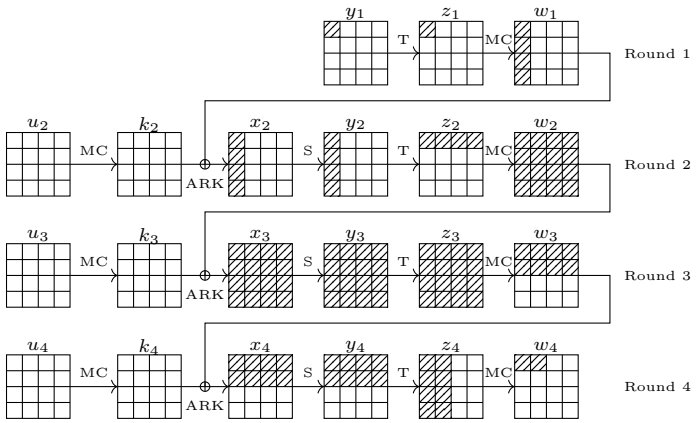


Figure 20: Differential path used in the "internal" property of our DS-MITM attack on 7.5-round Saturnin.

The proof of this property uses arguments inherited from rebound attacks. We first guess the value of $\Delta y_1[0]$, the difference in $y_1[0]$, for the given pair. We obtain the difference in x_2. We guess the four nibbles $x_2[0,1,2,3]$ and obtain the differences in y_2, hence x_3. On the other side, we guess $\Delta w_4[0,4]$, hence obtaining Δy_4; we guess 8 more state nibbles to obtain Δx_4, hence Δy_3. It remains to match between the two differences. There is on average one solution (we assume that the solutions to the differential equation of the Super-Sbox are tabulated). At this point, we know the whole sequence of states $x_2[0,1,2,3]$, $x_3[0\dots15]$, $x_4[0,1,4,5,8,9,12,13]$ for the given pair. So we can propagate the arbitrary sequence of differences in $y_1[0]$ up to $w_4[0,4]$, whose values can be computed and stored.

Attack. We use the usual DS-MITM attack principle, but reorder the steps as done in the last section of [BNPS19]. First of all, using 2^{225} encryption queries (grouped into structures that make the first line of the input take all 2^{64} possibilities), we build $2^{14\times16} = 2^{224}$ plaintext-ciphertext pairs verifying the input-output differential. We then build a table of size $2^{15\times16} = 2^{240}$, indexed by all 2^{240} guesses of K (the whole key, except the nibble number 3) and associated to good pairs of plaintext. The rest of the computation consists in finding the good key guess in this table:

- For each of the 2^{224} pairs, we find the 2^{16} key guesses such that the whole differential path is satisfied: we have 3 super-nibbles conditions in round 0 in order to reach the input of the internal property, while we have $(9+2)$ super-nibble conditions in rounds 6 and 5 respectively for reaching the output of the internal property. As we have 15 supper-nibbles of the key involved and a total of 14 supernibble conditions, we obtain around 2^{16} posibilities for the 15 words of the key that lead to the internal property. In less than 2^{240} time, thanks to early-abort techniques, we associate each key guess, with the good corresponding pairs leading to the correct path.

- Consider a key guess and its corresponding pair. Let $P, x_0 \dots z_7, C$ and $P', x'_0 \dots z'_7, C'$ be the respective sequences of internal states for this pair. We create a set of 2^4 plaintexts by making the difference in $y_1[0]$ assume the 2^4 arbitrary values chosen before. Due to the knowledge of the key supernibbles, we know the value of $y_1[0]$, so we can propagate its difference to $w_0[0]$, and know the necessary state nibbles to complete our plaintext sequence. We encrypt this sequence with the secret-key oracle, obtaining 2^4 ciphertexts. We then partially decrypt the corresponding ciphertexts and obtain the sequence of differences in $w_4[0,4]$, since we know the necessary key

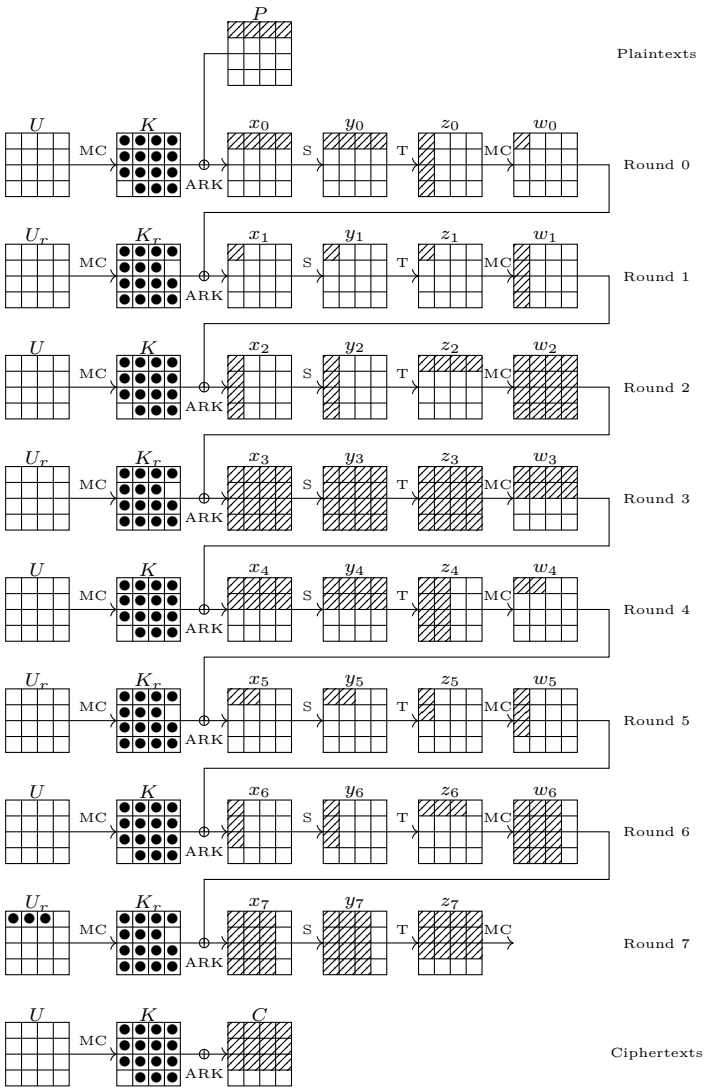


Figure 21: Full differential path used in the DS-MITM attack on 7.5-round SATURNIN. We denote the master key K and its rotated version (for odd rounds) K_r. U and U_r are respectively $MC^{-1}(K)$ and $MC^{-1}(K_r)$.

supernibbles. In total, this requires 2^{244} encryptions, after which we have stored 2^{244} 256-bit blocks with the associated keys.

- Then, we perform an exhaustive search in the middle. We compute all possible sequences of 2^4 differences in $w_4[0,4]$, for all choices of internal state supernibbles, and we search a collision with some key guess in the table. There are 2^{240} sequences to compute, each costs 2^4 partial SATURNIN encryptions.

In the end, we expect only one collision to occur, which gives the good key guess (that we can complete easily after). This slightly improved attack mainly works because of the simpler key-schedule of SATURNIN, which allows to reuse multiple times the key guesses, contrary to AES-128. The attack complexities are 2^{244} chosen-plaintext queries, an equivalent time, and 2^{244} memory (counted in 256-bit blocks).

5.3 Security of the Block Cipher Against Quantum Attacks

Given a few plaintext-ciphertext pairs, a quantum adversary can perform a naive exhaustive search for the key using Grover's algorithm [Gro96]. It requires approximately $2^{256/2} = 2^{128}$ iterations (we claim that it requires more than 2^{112} encryptions), each of which contains basic operations and an evaluation of an implementation of Saturnin as a quantum circuit (see [GLRS16] for the AES). We do not go into the details of such an implementation, but we point out that it would require at least as much quantum gates as classical gates (and possibly more, due to the necessity of reversibility), with interleaved layers of quantum error correction. Error-correcting operations are quantum, but classically controlled, and they provoke an overhead with respect to classical computations.

In its super-sbox representation, Saturnin has a similar shape as AES, with a 256-bit state. To date, there does not exist much literature on quantum attacks on reduced-round AES in the secret-key setting, *i.e.* procedures that retrieve the secret key faster than Grover's algorithm. A first analysis was made in [BNPS19]. The authors show that quantum versions of the Square attack can target up to 6 rounds of AES-128 and 7 rounds of AES-192 and AES-256 (as it is the case classically), and they construct a quantum DS-MITM attack on 8-round AES-256. All these procedures only require classical plaintext queries to a secret-key oracle. No attack was found with a better speed-up than quadratic, so for now it seems safe to claim, at least, the same quantum security margin as the classical one.

We should remark here that the DS-MITM attack of [BNPS19] reaches 8 rounds in a case where the key length is the double of the block length. On the contrary, the quantum security margin of Saturnin, the highest number of rounds attacked with a procedure faster than Grover, would likely be similar to that of AES-128. Indeed, we do not know of any better *quantum* attack than an adaptation of the Square for 6-round AES (its time, data and memory complexity being roughly the square of the original ones, given in Table 3 of [BNPS19]). The quantum security margin seems therefore bigger, with 6 reached super-rounds versus 7.5 classically.

We also remark that computing input-output pairs with target differences is the bottleneck of classical impossible differential and DS-MITM attacks against AES-128. This operation seems to enjoy less than a quadratic quantum speedup, as this requires finding partial collisions [BHT98, Amb07], and in some cases, it may require high amounts of quantum memory. Hence it may be difficult to make quantum versions of these attacks competitive against an exhaustive search for the key. The attack of [BNPS19] actually overcomes this issue by using only a low amount of classically computed pairs.

5.4 Security of the Modes of Operation

5.4.1 Against Classical Adversaries

All our security estimates rely on the assumption that Saturnin is an ideal block cipher, classically and quantumly, meaning that distinguishing it from a random PRP requires time 2^{256} classically and 2^{128} quantumly; and more generally, the probability of success of a distinguisher running in time less than $\mathcal{T}$ (but with little data) is $\frac{\mathcal{T}}{2^{256}}$ classically and $\frac{\mathcal{T}^2}{2^{256}}$ quantumly. There are additional constant factors that these bounds do not take into account, which is why our final security claims are reduced.

In general, we recall that classical key recovery requires 2^{256} evaluations of Saturnin or a little less if an optimized exhaustive search is used. The birthday bound for random values of 256 bits is 2^{128}. We remark that the use of different domain separators prevents most length-extension attacks on our modes.

Advantage of a Classical Adversary. Let t be the tag length. In general, the expected advantage (probability of success p) of an adversary against our AE schemes (in breaking authenticity, confidentiality or integrity) is:

$$p < \frac{\mathcal{T}}{2^{256}} + \frac{\mathcal{D}^2}{2^{256}} + \mathcal{D}2^{-t}$$

where the adversary is entitled to encryption or verification queries of a total of $\mathcal{D}$ blocks, and $\mathcal{T}$ computation time. Indeed, the adversary succeeds in breaking the PRP security of SATURNIN, *i.e.* in finding the key, with probability $\frac{\mathcal{T}}{2^{256}}$ if he has time $\mathcal{T}$: this is exhaustive search (a negligible amount of data, say two encryption queries, is necessary). He can output a collision after $\mathcal{D}$ encryption queries with probability $\frac{\mathcal{D}^2}{2^{256}}$, breaking the AE scheme as a PRF. Finally, he can simply ask verification queries of random ciphertexts and tags; each of them has a probability 2^{-t} of being accepted (since the verifier will decipher and recompute the corresponding tag, ending up with a random string that must match the adversary's tag). Hence the probability of success after $\mathcal{D}$ verification queries is $\mathcal{D}2^{-t}$.

SATURNIN-CTR-Cascade. Our CTR usage makes sure that the value $N||$counter is different for any two blocks in two or the same message. Hence CTR remains secure up to (a little less than) the birthday bound 2^{128}. An attack can be mounted when the number of queries reaches 2^{128} [LS18].

SATURNIN-Short. In SATURNIN-Short, we reuse the same bound as SATURNIN-CTR-Cascade, except that, although the tag is not truncated to 128 bits, the attack using verification queries applies as if it were the case. Indeed, a forgery using verification queries amounts to finding c, N such that $E_k^{-1}(c)$ contains N on its 128 right bits. N is not queried in superposition, and the adversary has only access to the verification result, not the value. So this amounts to looking for 2^{256} good elements (sound pairs c, N) among $2^{256+128}$ (all choices) with the verifier as oracle. Classically, the probability of success after $\mathcal{D}$ trials is $\mathcal{D}2^{-128}$.

SATURNIN-Hash. If the underlying compression function is a pseudo-random function, then SATURNIN-Hash is also one. Finding a collision costs 2^{128} time with little memory using Floyd's cycle-finding algorithm and this can be parallelized [vW99].

5.4.2 Against Quantum Adversaries

All our generic attacks count a quantum evaluation of SATURNIN as a single quantum time unit. We note that a quantum oracle for SATURNIN would likely yield a significant overhead in practical time complexities.

Quantum Exhaustive Search with Limited Time. Suppose that we use Grover's algorithm to search for some preimage of 1 for a function $f : \{0,1\}^n \to \{0,1\}$. We start in a superposition over all $\{0,1\}^n$. Then we iterate an operator which, thanks to a call to f in superposition, increases the amplitude of all preimages of 1 ("good elements") and decreases that of the others ("bad elements"). The amount of amplitude "moved" at each iteration depends only on the ratio between the number of good and bad elements. After some time, this process is stopped and the result is measured, hoping that a good element will be obtained. But the probability of measuring a good element depends on the squared amplitude of the good subspace, which is why Grover's probability of success does not increase linearly with the time, but quadratically. When queries to f and / or the total time are limited to $\mathcal{D}$ and / or $\mathcal{T}$, assuming *e.g.* that there is one good element only, the success probability is $\frac{\mathcal{D}^2}{2^n}$ or $\frac{\mathcal{T}^2}{2^n}$ respectively, instead of $\frac{\mathcal{T}}{2^n}$ classically.

This is also why quantum exhaustive search has a worse parallelization speedup than its classical counterpart: Grover's algorithm excels as a sequential computation, but stopping it after some time makes the success probability decrease quadratically, contrary to classical exhaustive search.

Quantum Collision Search. Suppose given a superposition oracle access to a random function on n bits (*e.g.* our hash function Saturnin-Hash, which can be implemented by a quantum adversary as a quantum circuit). An algorithm by Brassard, Høyer and Tapp [BHT98] allows to retrieve a collision of this function in approximately $2^{n/3}$ superposition queries, using $2^{n/3}$ quantum random-access memory. Quantum random-access memory or quantum RAM (in that case, memory with quantum access) is a superior model of quantum computation which seems even further away from practical realizations. Some authors [GR04, Ber09, JS19] argue that quantum memory would be realized using qubits, and that maintaining $\mathcal{M}_q$ quantum memory would require $O(\mathcal{M}_q)$ classical computations at each step, due to quantum error correction.

If quantum RAM is considered not available, an alternative collision search can be done using [CNS17] in $2^{2n/5}$ superposition queries, using only as many qubits as required by Grover iterations and oracle evaluations (in our case, we can lower bound this cost by $2^8 = 256$ qubits, since this is the state size of Saturnin). It is worth noticing that, while the query lower bound for random functions is known to be $2^{n/3}$, there exists no proof that better memory usage than [BHT98] or better time complexity than [CNS17] is impossible, but no such algorithm is known.

In general, we can subsume [CNS17] and [BHT98] with a single (maybe non tight) bound of $\mathcal{T}^5 \times \mathcal{M}_q = 2^{2n}$ where $\mathcal{M}_q$ is the quantum memory available and $\mathcal{T}$ is the quantum time complexity (or the number of queries).

When the queries are limited to a value $\mathcal{D}$, quantum collision search succeeds with probability $\frac{\mathcal{D}^3}{2^n}$ [Zha15]. The best method consists in applying a truncated version of [BHT98]: we first query $\mathcal{D}/2$ elements, store them in quantum RAM and run $\mathcal{D}/2$ iterates of Grover, searching for a collision on this intermediate table.

Advantage of a Quantum Adversary. Let t be the tag length. In general, the expected advantage (probability of success p) of a quantum adversary against our AE schemes (in breaking authenticity, confidentiality or integrity) is:

$$p < \frac{\mathcal{T}^2}{2^{256}} + \frac{\mathcal{D}^3}{2^{256}} + \mathcal{D}^2 2^{-t}$$

where the adversary is entitled to encryption or verification queries of a total of $\mathcal{D}$ blocks (in superposition), and $\mathcal{T}$ computation time.

Indeed, the adversary succeeds in breaking the PRP security of Saturnin, *i.e.* in finding the key, with probability $\frac{\mathcal{T}^2}{2^n}$ if he has time $\mathcal{T}$: this is generically optimal, and corresponds to running Grover's algorithm, stopping at time $\mathcal{T}$ (after $\mathcal{T}$ iterations) and measuring the result. Furthermore, he succeeds in breaking the AE scheme as a PRF with probability $\frac{\mathcal{D}^3}{2^n}$, since this is the advantage of outputting a collision after $\mathcal{D}$ queries. Finally, he succeeds with probability $\mathcal{D}^2 2^{-t}$ in making the verifier accept a random tag, using Grover's algorithm with the verifier as oracle.

Notice that this bound is information-theoretic and considers the best time complexity, without notions of quantum memory requirement (the quantum query lower bound for collisions is applied).

Saturnin-Short. Quantumly, Grover's algorithm speeds up the search for a random ciphertext and nonce giving a good verification result. With $\mathcal{D}$ superposition verification

queries, the success probability is $\mathcal{D}^2 2^{-128}$, so everything happens as if the tag was actually truncated to 128 bits.

SATURNIN-Hash. Using the quantum time - quantum memory tradeoff $\mathcal{T}^5 \times \mathcal{M}_q = 2^{2n}$, we find that the best time is 2^{85} with unrestricted quantum random access memory. If qRAM is forbidden (and only "plain" quantum circuits are allowed), the algorithm of [CNS17] gives 2^{102} quantum time.

References

[ADK+14] Martin R. Albrecht, Benedikt Driessen, Elif Bilge Kavun, Gregor Leander, Christof Paar, and Tolga Yalçin. Block ciphers - focus on the linear layer (feat. PRIDE). In Juan A. Garay and Rosario Gennaro, editors, *CRYPTO 2014, Part I*, volume 8616 of *LNCS*, pages 57–76. Springer, Heidelberg, August 2014.

[Amb07] Andris Ambainis. Quantum walk algorithm for element distinctness. *SIAM J. Comput.*, 37(1):210–239, 2007.

[AMRS] Gorjan Alagic, Christian Majenz, Alexander Russell, and Fang Song. Quantum-secure message authentication via blind-unforgeability. IACR Cryptology ePrint Archive, Report 2018/1150. https://eprint.iacr.org/2018/1150.

[ATTU16] Mayuresh Vivekanand Anand, Ehsan Ebrahimi Targhi, Gelo Noel Tabia, and Dominique Unruh. Post-quantum security of the CBC, CFB, OFB, CTR, and XTS modes of operation. In Tsuyoshi Takagi, editor, *Post-Quantum Cryptography - 7th International Workshop, PQCrypto 2016*, pages 44–63. Springer, Heidelberg, 2016.

[BBS99] Eli Biham, Alex Biryukov, and Adi Shamir. Cryptanalysis of Skipjack reduced to 31 rounds using impossible differentials. In Jacques Stern, editor, *EUROCRYPT'99*, volume 1592 of *LNCS*, pages 12–23. Springer, Heidelberg, May 1999.

[BC13] Christina Boura and Anne Canteaut. On the influence of the algebraic degree of F^{-1} on the algebraic degree of $G \circ F$. *IEEE Trans. Information Theory*, 59(1):691–702, 2013.

[BCD11] Christina Boura, Anne Canteaut, and Christophe De Cannière. Higher-order differential properties of Keccak and Luffa. In Antoine Joux, editor, *FSE 2011*, volume 6733 of *LNCS*, pages 252–269. Springer, Heidelberg, February 2011.

[BCK96] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. Keying hash functions for message authentication. In Neal Koblitz, editor, *CRYPTO'96*, volume 1109 of *LNCS*, pages 1–15. Springer, Heidelberg, August 1996.

[BDK+18] Achiya Bar-On, Orr Dunkelman, Nathan Keller, Eyal Ronen, and Adi Shamir. Improved key recovery attacks on reduced-round AES with practical data and memory complexities. In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part II*, volume 10992 of *LNCS*, pages 185–212. Springer, Heidelberg, August 2018.

[Bel15] Mihir Bellare. New proofs for NMAC and HMAC: Security without collision resistance. *Journal of Cryptology*, 28(4):844–878, October 2015.

[Ber09] Daniel J. Bernstein. Cost analysis of hash collisions: Will quantum computers make SHARCS obsolete. In *SHARCS 2009*, pages 105–116, 2009. Available at http://skew2011.mat.dtu.dk/proceedings/.

[BHT98] Gilles Brassard, Peter Høyer, and Alain Tapp. Quantum cryptanalysis of hash and claw-free functions. In Claudio L. Lucchesi and Arnaldo V. Moura, editors, *LATIN 1998*, volume 1380 of *LNCS*, pages 163–169. Springer, Heidelberg, April 1998.

[BJK+16] Christof Beierle, Jérémy Jean, Stefan Kölbl, Gregor Leander, Amir Moradi, Thomas Peyrin, Yu Sasaki, Pascal Sasdrich, and Siang Meng Sim. The SKINNY family of block ciphers and its low-latency variant MANTIS. In Matthew Robshaw and Jonathan Katz, editors, *CRYPTO 2016, Part II*, volume 9815 of *LNCS*, pages 123–153. Springer, Heidelberg, August 2016.

[BK03] Mihir Bellare and Tadayoshi Kohno. A theoretical treatment of related-key attacks: RKA-PRPs, RKA-PRFs, and applications. In Eli Biham, editor, *EUROCRYPT 2003*, volume 2656 of *LNCS*, pages 491–506. Springer, Heidelberg, May 2003.

[BK09] Alex Biryukov and Dmitry Khovratovich. Related-key cryptanalysis of the full AES-192 and AES-256. In Mitsuru Matsui, editor, *ASIACRYPT 2009*, volume 5912 of *LNCS*, pages 1–18. Springer, Heidelberg, December 2009.

[BKN09] Alex Biryukov, Dmitry Khovratovich, and Ivica Nikolic. Distinguisher and related-key attack on the full AES-256. In Shai Halevi, editor, *CRYPTO 2009*, volume 5677 of *LNCS*, pages 231–249. Springer, Heidelberg, August 2009.

[BKR11] Andrey Bogdanov, Dmitry Khovratovich, and Christian Rechberger. Biclique cryptanalysis of the full AES. In Dong Hoon Lee and Xiaoyun Wang, editors, *ASIACRYPT 2011*, volume 7073 of *LNCS*, pages 344–371. Springer, Heidelberg, December 2011.

[BLNS18] Christina Boura, Virginie Lallemand, María Naya-Plasencia, and Valentin Suder. Making the impossible possible. *Journal of Cryptology*, 31(1):101–133, January 2018.

[BN08] Mihir Bellare and Chanathip Namprempre. Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. *Journal of Cryptology*, 21(4):469–491, October 2008.

[BNPS19] Xavier Bonnetain, María Naya-Plasencia, and André Schrottenloher. Quantum security analysis of AES. IACR Cryptology ePrint Archive, Report 2019/272, 2019. https://eprint.iacr.org/2019/272.

[BR00] Mihir Bellare and Phillip Rogaway. Encode-then-encipher encryption: How to exploit nonces or redundancy in plaintexts for efficient cryptography. In Tatsuaki Okamoto, editor, *ASIACRYPT 2000*, volume 1976 of *LNCS*, pages 317–330. Springer, Heidelberg, December 2000.

[BZ13a] Dan Boneh and Mark Zhandry. Quantum-secure message authentication codes. In Thomas Johansson and Phong Q. Nguyen, editors, *EUROCRYPT 2013*, volume 7881 of *LNCS*, pages 592–608. Springer, Heidelberg, May 2013.

[BZ13b] Dan Boneh and Mark Zhandry. Secure signatures and chosen ciphertext security in a quantum computing world. In Ran Canetti and Juan A. Garay, editors, *CRYPTO 2013, Part II*, volume 8043 of *LNCS*, pages 361–379. Springer, Heidelberg, August 2013.

[CNS17] André Chailloux, María Naya-Plasencia, and André Schrottenloher. An efficient quantum collision search algorithm and implications on symmetric cryptography. In Tsuyoshi Takagi and Thomas Peyrin, editors, *ASIACRYPT 2017, Part II*, volume 10625 of *LNCS*, pages 211–240. Springer, Heidelberg, December 2017.

[DFJ13] Patrick Derbez, Pierre-Alain Fouque, and Jérémy Jean. Improved key recovery attacks on reduced-round AES in the single-key setting. In Thomas Johansson and Phong Q. Nguyen, editors, *EUROCRYPT 2013*, volume 7881 of *LNCS*, pages 371–387. Springer, Heidelberg, May 2013.

[DKR97] Joan Daemen, Lars R. Knudsen, and Vincent Rijmen. The block cipher Square. In Eli Biham, editor, *FSE'97*, volume 1267 of *LNCS*, pages 149–165. Springer, Heidelberg, January 1997.

[DL18] Sébastien Duval and Gaëtan Leurent. MDS matrices with lightweight circuits. *IACR Trans. Symm. Cryptol.*, 2018(2):48–78, 2018.

[DR99] Joan Daemen and Vincent Rijmen. AES Proposal: Rijndael. Submission to the NIST AES competition, 1999.

[DR02a] Joan Daemen and Vincent Rijmen. AES and the wide trail design strategy (invited talk). In Lars R. Knudsen, editor, *EUROCRYPT 2002*, volume 2332 of *LNCS*, pages 108–109. Springer, Heidelberg, April / May 2002.

[DR02b] Joan Daemen and Vincent Rijmen. *The Design of Rijndael: AES - The Advanced Encryption Standard*. Information Security and Cryptography. Springer, 2002.

[DS08] Hüseyin Demirci and Ali Aydin Selçuk. A meet-in-the-middle attack on 8-round AES. In Kaisa Nyberg, editor, *FSE 2008*, volume 5086 of *LNCS*, pages 116–126. Springer, Heidelberg, February 2008.

[Gag17] Tommaso Gagliardoni. *Quantum Security of Cryptographic Primitives*. PhD thesis, Darmstadt University of Technology, Germany, 2017.

[GBB+08] Henri Gilbert, Ryad Benadjila, Olivier Billet, Gilles Macario-Rat, Thomas Peyrin, Matt Robshaw, and Yannick Seurin. SHA-3 proposal: ECHO. Submission to NIST, 2008. https://ehash.iaik.tugraz.at/uploads/9/91/Echo.pdf.

[GHS16] Tommaso Gagliardoni, Andreas Hülsing, and Christian Schaffner. Semantic security and indistinguishability in the quantum world. In Matthew Robshaw and Jonathan Katz, editors, *CRYPTO 2016, Part III*, volume 9816 of *LNCS*, pages 60–89. Springer, Heidelberg, August 2016.

[GJ75] Terry Gilliam and Terry Jones. Monty Python and the Holy Grail. Distributed by EMI Films, 1975.

[GKM+08] Praveen Gauravaram, Lars R. Knudsen, Krystian Matusiewicz, Florian Mendel, Christian Rechberger, Martin Schläffer, and Søren S. Thomsen. Grøstl – a SHA-3 candidate. Submission to NIST, 2008. http://www.groestl.info/Groestl.pdf.

[GLRS16] Markus Grassl, Brandon Langenberg, Martin Roetteler, and Rainer Steinwandt. Applying Grover's algorithm to AES: Quantum resource estimates. In Tsuyoshi Takagi, editor, *Post-Quantum Cryptography - 7th International Workshop, PQCrypto 2016*, pages 29–43. Springer, Heidelberg, 2016.

[GM08] Samuel Galice and Marine Minier. Improving integral attacks against Rijndael-256 up to 9 rounds. In Serge Vaudenay, editor, *AFRICACRYPT 08*, volume 5023 of *LNCS*, pages 1–15. Springer, Heidelberg, June 2008.

[GR04] Lov K. Grover and Terry Rudolph. How significant are the known collision and element distinctness quantum algorithms? *Quantum Information & Computation*, 4(3):201–206, 2004.

[Gra18] Lorenzo Grassi. Mixture differential cryptanalysis: a new approach to distinguishers and attacks on round-reduced AES. *IACR Trans. Symm. Cryptol.*, 2018(2):133–160, 2018.

[Gro96] Lov K. Grover. A fast quantum mechanical algorithm for database search. In *28th ACM STOC*, pages 212–219. ACM Press, May 1996.

[GRR17] Lorenzo Grassi, Christian Rechberger, and Sondre Rønjom. A new structural-differential property of 5-round AES. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *EUROCRYPT 2017, Part II*, volume 10211 of *LNCS*, pages 289–317. Springer, Heidelberg, April / May 2017.

[HY18] Akinori Hosoyamada and Kan Yasuda. Building quantum-one-way functions from block ciphers: Davies-Meyer and Merkle-Damgård constructions. In Thomas Peyrin and Steven Galbraith, editors, *ASIACRYPT 2018, Part I*, volume 11272 of *LNCS*, pages 275–304. Springer, Heidelberg, December 2018.

[IAC+08] Sebastiaan Indesteege, Elena Andreeva, Christophe De Cannière, Orr Dunkelman, Emilia Käsper, Svetla Nikova, Bart Preneel, and Elmar Tischhauser. The LANE hash function. Submission to NIST, 2008. https://www.esat.kuleuven.be/cosic/publications/article-1181.pdf.

[JS19] Samuel Jaques and John M. Schanck. Quantum cryptanalysis in the RAM model: Claw-finding attacks on SIKE. IACR Cryptology ePrint Archive, Report 2019/103, 2019. https://eprint.iacr.org/2019/103.

[KLLN16a] Marc Kaplan, Gaëtan Leurent, Anthony Leverrier, and María Naya-Plasencia. Breaking symmetric cryptosystems using quantum period finding. In Matthew Robshaw and Jonathan Katz, editors, *CRYPTO 2016, Part II*, volume 9815 of *LNCS*, pages 207–237. Springer, Heidelberg, August 2016.

[KLLN16b] Marc Kaplan, Gaëtan Leurent, Anthony Leverrier, and María Naya-Plasencia. Quantum differential and linear cryptanalysis. *IACR Trans. Symm. Cryptol.*, 2016(1):71–94, 2016. http://tosc.iacr.org/index.php/ToSC/article/view/536.

[KM10] Hidenori Kuwakado and Masakatu Morii. Quantum distinguisher between the 3-round Feistel cipher and the random permutation. In *ISIT 2010*, pages 2682–2685. IEEE, 2010.

[KM12] Hidenori Kuwakado and Masakatu Morii. Security on the quantum-type even-mansour cipher. In *ISITA 2012*, pages 312–316. IEEE, 2012.

[LP07] Gregor Leander and Axel Poschmann. On the Classification of 4 Bit S-Boxes. In Claude Carlet and Berk Sunar, editors, *WAIFI 2007*, volume 4547 of *LNCS*, pages 159–176. Springer, Heidelberg, June 2007.

[LS18] Gaëtan Leurent and Ferdinand Sibleyras. The missing difference problem, and its applications to counter mode encryption. In Jesper Buus Nielsen and Vincent Rijmen, editors, *EUROCRYPT 2018, Part II*, volume 10821 of *LNCS*, pages 745–770. Springer, Heidelberg, April / May 2018.

[LTW18] Gregor Leander, Cihangir Tezcan, and Friedrich Wiemer. Searching for subspace trails and truncated differentials. *IACR Trans. Symm. Cryptol.*, 2018(1):74–100, 2018.

[MPP09] Marine Minier, Raphael C.-W. Phan, and Benjamin Pousse. Distinguishers for ciphers and known key attack against Rijndael with large blocks. In Bart Preneel, editor, *AFRICACRYPT 09*, volume 5580 of *LNCS*, pages 60–76. Springer, Heidelberg, June 2009.

[Nak08] Jorge Nakahara Jr. 3D: A three-dimensional block cipher. In Matthew K. Franklin, Lucas Chi Kwong Hui, and Duncan S. Wong, editors, *CANS 08*, volume 5339 of *LNCS*, pages 252–267. Springer, Heidelberg, December 2008.

[Nat01] National Institute of Standards and Technology. SP 800-38A: Recommendation for Block Cipher Modes of Operation: Methods and Techniques, December 2001.

[Nat15] National Institute of Standards and Technology. SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions. NIST FIPS PUB 202, U.S. Department of Commerce, August 2015.

[Nat16] National Institute of Standards and Technlology. Submission requirements and evaluation criteria for the post-quantum cryptography standardization process, 2016. https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/call-for-proposals-final-dec-2016.pdf.

[Nat18] National Academies of Sciences, Engineering, and Medicine. *Quantum Computing: Progress and Prospects*. The National Academies Press, Washington, DC, 2018.

[NC02] Michael A Nielsen and Isaac Chuang. *Quantum computation and quantum information*. AAPT, 2002.

[NP07] Jorge Nakahara Jr. and Ivan Carlos Pavão. Impossible-differential attacks on large-block Rijndael. In Juan A. Garay, Arjen K. Lenstra, Masahiro Mambo, and René Peralta, editors, *ISC 2007*, volume 4779 of *LNCS*, pages 104–117. Springer, Heidelberg, October 2007.

[RBH17] Sondre Rønjom, Navid Ghaedi Bardeh, and Tor Helleseth. Yoyo tricks with AES. In Tsuyoshi Takagi and Thomas Peyrin, editors, *ASIACRYPT 2017, Part I*, volume 10624 of *LNCS*, pages 217–243. Springer, Heidelberg, December 2017.

[Sas10] Yu Sasaki. Known-key attacks on Rijndael with large blocks and strengthening ShiftRow parameter. In Isao Echizen, Noboru Kunihiro, and Ryôichi Sasaki, editors, *IWSEC 10*, volume 6434 of *LNCS*, pages 301–315. Springer, Heidelberg, November 2010.

[Sho94] Peter W. Shor. Algorithms for quantum computation: Discrete logarithms and factoring. In *35th FOCS*, pages 124–134. IEEE Computer Society Press, November 1994.

[Sim94] Daniel R. Simon. On the power of quantum computation. In *35th FOCS*, pages 116–123. IEEE Computer Society Press, November 1994.

[SJS16] Vladimir Soukharev, David Jao, and Srinath Seshadri. Post-quantum security models for authenticated encryption. In Tsuyoshi Takagi, editor, *Post-Quantum Cryptography - 7th International Workshop, PQCrypto 2016*, pages 64–78. Springer, Heidelberg, 2016.

[SLG+16] Bing Sun, Meicheng Liu, Jian Guo, Longjiang Qu, and Vincent Rijmen. New insights on AES-like SPN ciphers. In Matthew Robshaw and Jonathan Katz, editors, *CRYPTO 2016, Part I*, volume 9814 of *LNCS*, pages 605–624. Springer, Heidelberg, August 2016.

[SY17] Fang Song and Aaram Yun. Quantum security of NMAC and related constructions - PRF domain extension against quantum attacks. In Jonathan Katz and Hovav Shacham, editors, *CRYPTO 2017, Part II*, volume 10402 of *LNCS*, pages 283–309. Springer, Heidelberg, August 2017.

[UDI+11] Markus Ullrich, Christophe De Cannière, Sebastian Indesteege, Özgül Küçük, Nicky Mouha, and Bart Preneel. Finding optimal bitsliced implementations of 4x4-bit s-boxes. In *SKEW 2011*, 2011. Available at http://skew2011.mat.dtu.dk/proceedings/.

[vW99] Paul C. van Oorschot and Michael J. Wiener. Parallel collision search with cryptanalytic applications. *Journal of Cryptology*, 12(1):1–28, January 1999.

[WGR+13] Qingju Wang, Dawu Gu, Vincent Rijmen, Ya Liu, Jiazhe Chen, and Andrey Bogdanov. Improved impossible differential attacks on large-block Rijndael. In Taekyoung Kwon, Mun-Kyu Lee, and Daesung Kwon, editors, *ICISC 12*, volume 7839 of *LNCS*, pages 126–140. Springer, Heidelberg, November 2013.

[Zha15] Mark Zhandry. A note on the quantum collision and set equality problems. *Quantum Information & Computation*, 15(7&8):557–567, 2015.

[Zha18] Mark Zhandry. How to record quantum queries, and applications to quantum indifferentiability. Cryptology ePrint Archive, Report 2018/276, 2018. https://eprint.iacr.org/2018/276.

[ZWP+08] Lei Zhang, Wenling Wu, Je Hong Park, Bon Wook Koo, and Yongjin Yeom. Improved impossible differential attacks on large-block Rijndael. In Tzong-Chen Wu, Chin-Laung Lei, Vincent Rijmen, and Der-Tsai Lee, editors, *ISC 2008*, volume 5222 of *LNCS*, pages 298–315. Springer, Heidelberg, September 2008.

A Implementations of Inverse Components

```
#define S_LAYER_INV(a, b, c, d) { \
        a ^= b | d;               \
        b ^= a | c;               \
        c ^= b & d;               \
        d ^= b | c;               \
        b ^= a | d;               \
        a ^= b & c;               \
    }
```

(a) σ^{-1}.

```
#define PI_0_INV(a, b, c, d, tmp) { \
        tmp = a; a = d; d = c;      \
        c = b; b = tmp;             \
    }

#define PI_1_INV(a, b, c, d, tmp) { \
        tmp = a; a = c;             \
        c = d; d = tmp;             \
    }
```

(b) π_0^{-1} and π_1^{-1}.

Figure 22: C code needed to implement the inverse S-boxes in a bitsliced fashion

```
#define MUL_INV(x0, x1, x2, x3, tmp) {                                  \
        x3 ^= x0;                                       /* XOR */       \
        tmp = x0; x0 = x3; x3 = x2; x2 = x1; x1 = tmp; /* rotation */   \
}

#define MDS_INV(x0, x1, x2, x3, x4, x5, x6, x7,                         \
                x8, x9, xa, xb, xc, xd, xe, xf, tmp) {                  \
        xc ^= x0; xd ^= x1; xe ^= x2; xf ^= x3; /* D ^= A */            \
        x4 ^= x8; x5 ^= x9; x6 ^= xa; x7 ^= xb; /* B ^= C */            \
        x0 ^= x4; x1 ^= x5; x2 ^= x6; x3 ^= x7; /* A ^= B */            \
        x8 ^= xc; x9 ^= xd; xa ^= xe; xb ^= xf; /* C ^= D */            \
        MUL_INV(x0, x1, x2, x3, tmp);           /* A = MUL_INV(A) */    \
        MUL_INV(x0, x1, x2, x3, tmp);           /* A = MUL_INV(A) */    \
        MUL_INV(x8, x9, xa, xb, tmp);           /* C = MUL_INV(C) */    \
        MUL_INV(x8, x9, xa, xb, tmp);           /* C = MUL_INV(C) */    \
        xc ^= x0; xd ^= x1; xe ^= x2; xf ^= x3; /* D ^= A */            \
        x4 ^= x8; x5 ^= x9; x6 ^= xa; x7 ^= xb; /* B ^= C */            \
        MUL_INV(x4, x5, x6, x7, tmp);           /* B = MUL_INV(B) */    \
        MUL_INV(xc, xd, xe, xf, tmp);           /* D = MUL_inv(D) */    \
        x0 ^= x4; x1 ^= x5; x2 ^= x6; x3 ^= x7; /* A ^= B */            \
        x8 ^= xc; x9 ^= xd; xa ^= xe; xb ^= xf; /* C ^= D */            \
    }
```

Figure 23: The implementation of the inverse MDS matrix as a C macro.

B Some Quantum Computing Notions

Quantum Computing and Oracles We refer to [NC02] for a comprehensible introduction to quantum computing. While it is certainly difficult to assert the *effective* time of yet-to-be-implemented quantum operations, we use as common ground the quantum circuit model. Quantum and classical time complexities can only be compared roughly; but two quantum circuits, as they are written in the same abstract manner, can be compared precisely. By replacing classical exhaustive search by quantum exhaustive search, *i.e.* Grover's search, the notions of *quantum level of security*, *quantum security margin* and *quantum attack* become immediately available, by analogy with their classical counterparts.

A quantum circuit consists of a sequence of operations (or quantum gates) applied to a pool of qubits. Qubits are the analogue of classical bits. Classically, the state of a bit is either 0 or 1. Quantumly, the state of a qubit is a vector in a two-dimensional Hilbert space,

with a canonical basis $\{|0\rangle, |1\rangle\}$. The qubits are first prepared in an arbitrary state, say $|0\rangle$. The operations, quantum gates and oracle queries, are all unitary operators; and they are all reversible. Via *entanglement*, a general n-qubit system can only be described on a basis of 2^n vectors. But all information on the system is not accessible. The different basis vectors $|i\rangle$ have complex amplitudes α_i, such that $|\alpha_i|^2$ is the probability to obtain i upon *measurement* of the system. A measurement destroys the state (the superposition *collapses*) and replaces it with $|i\rangle$ for the obtained i. Consequently, only n bits of information can be extracted from an n-qubit quantum system. The sequence of quantum gates causes constructive and destructive interferences between the states, which reduce the amplitude of "bad states" and increase that of "good states", so that, upon measurement, we expect a meaningful result.

Throughout this document, we use the circuit model. We do not need to specify exactly the universal gate set used, as regarding quantum attacks, our complexities will be given in multiples of a quantum circuit for SATURNIN.

Oracles. Classically, an oracle (for encryption, decryption, verification...) can be seen as a black-box function f which takes an adversary-controlled input x and returns $f(x)$. Quantumly, a *superposition* oracle is a unitary O_f which takes an input $|x\rangle|b\rangle$ and returns $|x\rangle|b \oplus f(x)\rangle$. The output $f(x)$ is only written on the additional register, ensuring reversibility. The input state can be in any superposition, but this does not make the adversary in power to compute "all possibilities at once", for example to query a secret-key oracle on all the codebook in one query, since the information can only obtained by measurement, and a measurement makes the superposition collapse.

Simon's algorithm. Given superposition query access to a function $f : \{0,1\}^n \to \{0,1\}^n$ which "hides" a period s, *i.e.* $f(x \oplus s) = f(x)$ for all s, Simon's algorithm [Sim94] recovers s using $O(n)$ queries to O_f and little (polynomial in n) computation overhead.

Grover's algorithm. Given superposition query access to a function $f : \{0,1\}^n \to \{0,1\}$, such that $f^{-1}(1)$ contains 2^t elements, Grover's algorithm [Gro96] recovers a preimage of 1 in $O(2^{(n-t)/2})$ time and queries to O_f instead of $O(2^{n-t})$ classical time and queries to f. This algorithm consists in iterating $O(2^{(n-t)/2})$ times a unitary operator which uses a query to O_f to move some amplitude towards the elements of $f^{-1}(1)$.

BHT Collision Search Algorithm. The algorithm of [BHT98] uses Grover's search as a subroutine. It makes $2^{n/3}$ queries, stores them in quantum hardware, and looks for a collision on one of the queried elements. The probability for a random element to collide on this table is $\frac{2^{n/3}}{2^n}$, so this Grover's search step requires time $2^{n/3}$. This algorithm is optimal for a random function.

C Impossible Differential Distinguishers

We reproduce here the two impossible differential distinguishers that we have studied for Saturnin. The whole cubic state of the cipher is represented as its 4 slices. We denote by S the S-Box layer, MD the operation $\mathsf{SR}_{\text{slice}} \circ \mathsf{MC} \circ \mathsf{SR}_{\text{slice}}^{-1}$ and by MS the operation $\mathsf{SR}_{\text{sheet}} \circ \mathsf{MC} \circ \mathsf{SR}_{\text{sheet}}^{-1}$.

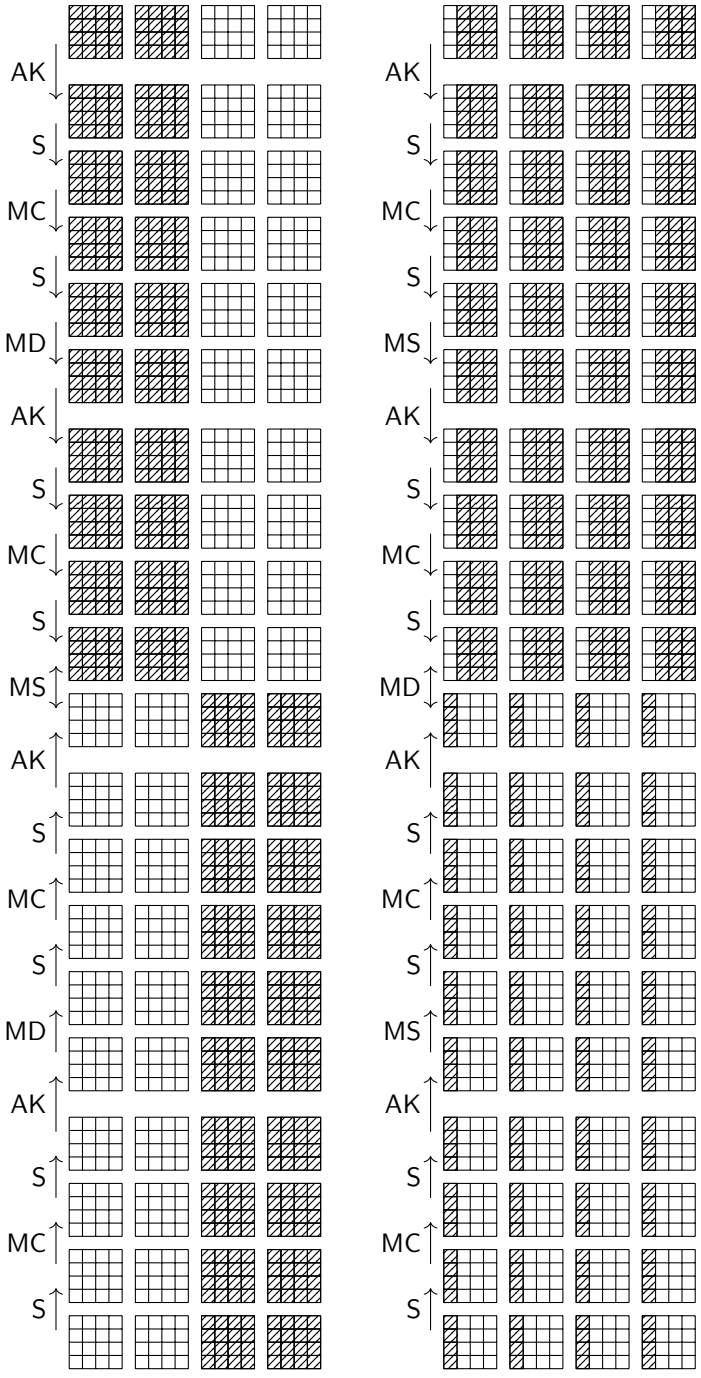


Figure 24: Paths A (left) and B (right).

D On the Name

The transcription of the French pronounciation of "SATURNIN" is given in Figure 25.

satyʁnɛ̃

Figure 25: How to pronounce SATURNIN using the International Phonetic Alphabet.

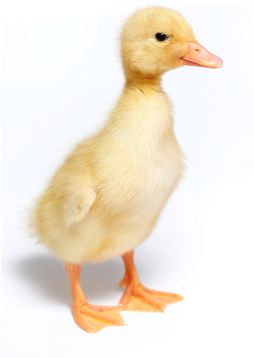
There are multiple motivations for this name.

Saturnin the Duck. The duck is undeniably a symbol of lightness because it floats. It has been famously used as the reference for lightness throughout the ages, for instance by Sir Bedevere [GJ75]. The bantamweight weight class in boxing is also named after a small duck, and corresponds to lightweight fighters. As it turns out, Saturnin is the most famous duck in France: it was the hero of a well-known TV show[5]. A Saturnin-like[6] yellow duck is shown in Figure 26a.

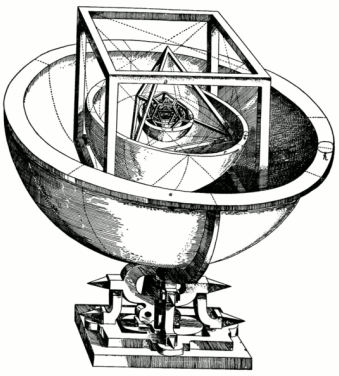
Kepler's *Mysterium Cosmographicum*. The astronomer made the following observation in his 1596 opus, as described on his wikipedia page:

> [Kepler] found that each of the five Platonic solids could be inscribed and circumscribed by spherical orbs; nesting these solids, each encased in a sphere, within one another would produce six layers, corresponding to the six known planets—Mercury, Venus, Earth, Mars, Jupiter, and Saturn. By ordering the solids selectively—octahedron, icosahedron, dodecahedron, tetrahedron, cube—Kepler found that the spheres could be placed at intervals corresponding to the relative sizes of each planet's path, assuming the planets circle the Sun.

This system is summarized in Figure 26b. As we can see, the planet Saturn is associated with the cube—the exact shape of our cipher.



(a) A Saturnin-like duck (credit: Fir0002/Flagstaffotos).



(b) From Kepler's *Mysterium Cosmographicum*, via Wikipedia.

Wisdom. The planet Saturn is a symbol of the wisdom coming with age, a fitting metaphor for our reliance on the knowledge accumulated since the publication of the AES.

[5] https://fr.wikipedia.org/wiki/Les_Aventures_de_Saturnin.
[6] https://commons.wikimedia.org/wiki/File:Duckling_-_domestic_duck.jpg