IACR Transactions on Symmetric Cryptology
ISSN 2519-173X, Vol. 2020, No. S1, pp. 295–349. DOI:10.13154/tosc.v2020.iS1.295-349

Spook: Sponge-Based Leakage-Resistant Authenticated Encryption with a Masked Tweakable Block Cipher

Davide Bellizia[1], Francesco Berti[1], Olivier Bronchain[1], Gaëtan Cassiers[1], Sébastien Duval[1], Chun Guo[2], Gregor Leander[3], Gaëtan Leurent[4], Itamar Levi[1], Charles Momin[1], Olivier Pereira[1], Thomas Peters[1], François-Xavier Standaert[1], Balazs Udvarhelyi[1] and Friedrich Wiemer[3]

[1] ICTEAM Institute, Université catholique de Louvain, Louvain-la-Neuve, Belgium
[2] School of Cyber Science and Technology and Key Laboratory of Cryptologic Technology and Information Security, Ministry of Education, Shandong University, Jinan, China
[3] Ruhr-Universität Bochum, Bochum, Germany
[4] Team SECRET, Inria Paris Research Center, Paris, France
team@spook.dev

Abstract. This paper defines Spook: a sponge-based authenticated encryption with associated data algorithm. It is primarily designed to provide security against side-channel attacks at a low energy cost. For this purpose, Spook is mixing a leakage-resistant mode of operation with bitslice ciphers enabling efficient and low latency implementations. The leakage-resistant mode of operation leverages a re-keying function to prevent differential side-channel analysis, a duplex sponge construction to efficiently process the data, and a tag verification based on a Tweakable Block Cipher (TBC) providing strong data integrity guarantees in the presence of leakages. The underlying bitslice ciphers are optimized for the masking countermeasures against side-channel attacks. Spook is an efficient single-pass algorithm. It ensures state-of-the-art black box security with several prominent features: (*i*) nonce misuse-resilience, (*ii*) beyond-birthday security with respect to the TBC block size, and (*iii*) multi-user security at minimum cost with a public tweak. Besides the specifications and design rationale, we provide first software and hardware implementation results of (unprotected) Spook which confirm the limited overheads that the use of two primitives sharing internal components imply. We also show that the integrity of Spook with leakage, so far analyzed with unbounded leakages for the duplex sponge and a strongly protected TBC modeled as leak-free, can be proven with a much weaker unpredictability assumption for the TBC. We finally discuss external cryptanalysis results and tweaks to improve both the security margins and efficiency of Spook.

Keywords: Authenticated encryption · NIST lightweight cryptography standardization effort · leakage-resistance · bitslice ciphers · masking countermeasure · low energy

1 Introduction: design rationale and motivation

Spook is an Authenticated Encryption scheme with Associated Data (AEAD). Its primary design goals are *resistance against side-channel analysis* and *low-energy implementations* (jointly). The motivation for the first goal stems from the observation that lightweight devices may be deployed in environments where they can be under physical control of an adversary, yet be responsible for sensitive tasks, or be the root of critical distributed

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

attacks starting from seemingly non-critical connected objects [RSWO18]. As a result, the ability to provide side-channel resistance (and possibly resistance against fault attacks) easily and at low cost was identified by the NIST as a desirable feature for lightweight cryptography.[1] The motivation for the second goal stems from the observation that energy is a suitable metric to compare the performances of cryptographic algorithms [KDH+12], and a relevant one from the application viewpoint. It is in particular increasingly needed for battery-operated / energy harvesting devices, for example in the IoT [MMGD17].

In order to reach these goals, Spook builds on and specializes two main ingredients.

The first ingredient is a leakage-resistant mode of operation that enables efficient side-channel secure implementations. We use the TETSponge mode of operation for this purpose [GPPS19b], which is the lightweight variation of a sequence of works aiming at high-physical security guarantees for (authenticated) encryption [PSV15, BKP+18, BPPS17, BGP+20]. For integrity, TETSponge reaches the top of the definitions' hierarchy established in [GPPS19a], namely Ciphertext Integrity with Misuse and Leakage in encryption and decryption (CIML2), in a liberal model where all the intermediate computations are leaked to the adversary, except for a long-term secret key that is only used twice per encrypted or decrypted message. For confidentiality, TETSponge reaches security against Chosen Ciphertext Adversaries with misuse-resilience and Leakage in encryption (CCAmL1). Compared to related works with constructions additionally achieving CCA security with decryption leakages (i.e., CCAmL2 [GPPS19a]), the TETSponge mode has the significant advantage of being *single-pass* in encryption and in decryption, which we believe is essential for lightweight implementations.[2] Concretely, TETSponge encourages so-called *leveled implementations*, where (expensive) protections against side-channel attacks are used in a limited way and independent of the message size, while the bulk of the computation is executed by cheap and weakly protected components. As exhibited in [BGP+20] and [BBC+20] for software (resp., hardware) implementations, leveled implementations allow significant energy gains when side-channel protections must be activated.

The second ingredient is the adoption of regular symmetric primitives to operate the TETSponge mode of operation, namely the Clyde-128 Tweakable Block Cipher (TBC) and the Shadow-512 permutation, both based on simple extensions of the LS-design framework, which aims at efficient bitslice implementations [GLSV14]. In order to facilitate leveled implementations, those primitives use components that can be efficiently masked against side-channel attacks for the TBC (e.g., with [CGLS20] in hardware or [GR17] in software), and enable fast implementations for the permutation. They bring two main improvements compared to earlier proposals of LS-designs. On the one hand, they leverage the tools introduced by Beierle et al. [BCLR17] in order to prevent the invariant attacks that put several earlier LS-designs at risk [LMR15, TLS16]. On the other hand, they replace the table-based L-boxes used in previous LS-designs by word-level L-boxes that can be efficiently implemented as a sequence of rotation and XOR operations, which is beneficial to prevent cache attacks [TOS10]. As a result, both Clyde-128 and Shadow-512 enable efficient bitslicing and side-channel resistant implementations on a wide range of platforms, (e.g., 32-bit microprocessors such as increasingly used in mobile applications and dedicated hardware or FPGAs). Both primitives also share the same S-box and L-box in order to limit the implementation overheads in case of unprotected implementations.

The motivations for using two symmetric primitives in TETSponge are twofold. First, an invertible (tweakable) block cipher is instrumental to reach CIML2 security in the unbounded leakage model [BPPS17]. Second, duplex sponge constructions are in general attractive for efficient AE: they can achieve this functionality in a single pass, are highly

[1] https://csrc.nist.gov/projects/lightweight-cryptography.

[2] We use the definition of misuse-resilience of Ashur et al. [ADL17] rather than the definition of misuse-resistance of Rogaway and Schrimpton [RS06] for a similar reason (i.e., to avoid the need of two passes).

flexible and ensure nice security bounds in the multi-user setting [BDPA11, DMA17]. Sponge constructions are also believed to provide some leakage-resistance (or resilience) by design [DEM+17]. Spook combines the advantages of both. Eventually, and besides these main features, Spook inherits other interesting properties from the TETSponge mode of operation: (*i*) it is secure beyond the birthday bound (with respect to the size n of the TBC), and (*ii*) it can provide n-bit multi-user security at low cost with a public tweak. We additionally note that an important aspect of our security claims is that we consider security definitions that allow all the computations (including the computation of the "challenge ciphertext") to leak, which we denote as *leakage-resistance* (following the terminology in [GPPS19a, Sta19]). This is in contrast with alternative definitions of *leakage-resilience* excluding the leakage of the challenge ciphertext (e.g., [BMOS17]).

Besides specifications and design rationale for the mode and primitives, we provide first software and hardware implementation results of (unprotected) Spook and confirm the limited overheads that our use of two primitives with shared S-boxes and L-boxes imply. We also show that the integrity of Spook with leakage, so far analyzed with unbounded leakages for the duplex sponge and a strongly protected TBC modeled as leak-free, can be proven with a much weaker unpredictability assumption for the TBC, extending a recent result of Berti et al. to Spook [BGP+19]. We finally discuss external cryptanalysis results and tweaks in order to improve both the security margins and efficiency of Spook.

2 Specifications

2.1 The TETSponge mode of operation

Notations. We denote the plaintext as $\boldsymbol{M}$. It is parsed into ℓ blocks $M[0], M[1], \ldots, M[\ell-1]$, where the size of blocks 0 to $\ell-2$ is r and the size of the last block is $1 \leq |M[\ell-1]| \leq r$. We denote the associated data as $\boldsymbol{A}$. It is parsed into λ blocks $A[0], A[1], \ldots, A[\lambda-1]$ in the same way as the plaintext. We denote the τ-bit nonce as N and the key as $K||P$, where K is a long-term secret key of n bits, and P is a public tweak of $n-1$ bits (one bit is used to separate key and tag generations with the TBC).[3] The secret key K has to be selected uniformly at random in $\{0,1\}^n$. The public tweak P is set to an $(n-1)$-bit zero vector in case only single-user security is requested. In case multi-user security is requested, a long-term "public key" p of $n-2$ bits must be selected uniformly at random, since the instance of TBC we propose next is not designed to resist related-tweak attacks, and P is set to $p||1$ (so one bit is used to separate the single-user and multi-user security variants). As a result, the $\mathsf{TETSponge}[\mathsf{E}, \pi](\boldsymbol{A}, \boldsymbol{M}, N, K||P)$ mode of operation relies on a TBC with n-bit blocks, tweaks and keys, denoted as E, and an $(r+c)$-bit permutation denoted as π. Our primary parameters are $n = 128$, $r = 256$, $c = 256$ and $\tau = 128$.

Conventions. TETSponge operates over bitstrings (i.e., each of the manipulated data – the plaintext, associated data, ciphertext, keys and nonce – is a sequence of bits). The Spook cipher is however defined for bytestrings (i.e., each of the manipulated data is a sequence of bytes). For encryption, input data (i.e., plaintext, associated data, keys, nonce) bytestrings are first mapped to bitstrings using the BMAP function defined next, and the ciphertext is converted back to a bytestring using the inverse of the BMAP function. The operations are the same for decryption, except that the plaintext and ciphertext are swapped. BMAP maps bytes to bits in little-endian order. More precisely, it takes as input a sequence of bytes of length q: $(X[0], \ldots, X[q-1])$ and outputs a sequence of bits $(Y[0], \ldots, Y[8q-1])$, where $Y[8i+j] = (X[i]/2^j) \mod 2$ for $0 \leq i < q$ and $0 \leq j < 8$. As a result, the nonce N, the private part of the key K and (when applicable) its public part p are all 16 bytes long. In order to get the bitstring p (which has a length of 126 bits) from the corresponding bytestring, the last two bits are discarded after application of BMAP.

[3] So in our reference implementations, $K||P$ is the key input string required by the NIST API.

The encryption. The encryption of the 4-string input $(\boldsymbol{A}, \boldsymbol{M}, N, K||P)$, illustrated in Figure 1, first derives an n-bit initial seed B by using a TBC call $\mathsf{E}_K^{P||0}(N||0^*)$. The initial seed B is used as a fresh key for an inner keyed duplex sponge construction, to process $\boldsymbol{A}$ and $\boldsymbol{M}$ and produce $\boldsymbol{C}$. Two bits are used for domain separation, in order to distinguish $\boldsymbol{M}$ from $\boldsymbol{A}$ and mark if the last blocks of $\boldsymbol{A}$ and $\boldsymbol{M}$ are of full r bits or not. Let $U||V$ be the first $2n-1$ bits of the final state, with $|U| = n$. The tag Z is produced by using another TBC call $\mathsf{E}_K^{V||1}(U)$, where the 1 concatenated with V guarantees that this tweak is different from the one used to generate B. The ciphertext is made of $\ell - 1$ blocks of r bits, a final block of length $1 \leq |C[\ell-1]| \leq r$ and an n-bit tag. We next denote it as $\boldsymbol{C} := \boldsymbol{c}||Z := C[0]||\ldots||C[\ell-1]||Z$ (i.e., $\boldsymbol{c}$ is the ciphertext excluding the tag).

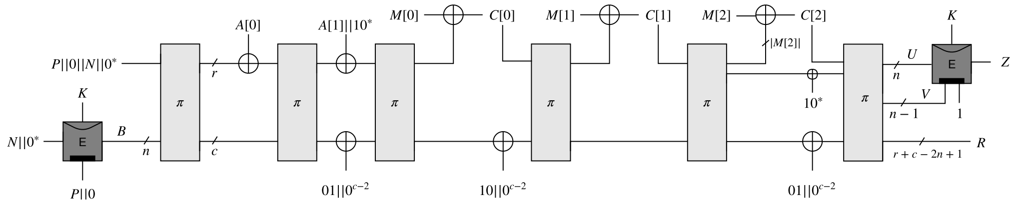

Figure 1: TETsponge mode with TBC E and permutation π, applied to a 2-block $\boldsymbol{A}$ and a 3-block $\boldsymbol{M}$. The value $01||0^{c-2}$ is inserted only if $|A[\lambda-1]| < r$ (resp., $|M[\ell-1]| < r$)).

The decryption. In order to decrypt the 4-string input $(\boldsymbol{A}, \boldsymbol{C}, N, K||P)$, the mode first derives the initial seed B via $\mathsf{E}_K^{P||0}(N||0^*)$, as when encrypting. It then runs the inner keyed duplex sponge construction on $\boldsymbol{A}$ and $\boldsymbol{c}$ to derive $\boldsymbol{M}$ and the $(2n-1)$-bit truncated state $U||V$. Finally, it makes an inverse TBC call $U^* = \left(\mathsf{E}_K^{V||1}\right)^{-1}(Z)$, and outputs $\boldsymbol{M}$ if and only if there is a match $U^* = U$. In this way, invalid decryption only leaks meaningless random values U^*, instead of the correct tags (so cannot be used for forgeries).

More precisely, the specification of TETSponge$[\mathsf{E}, \pi]$.Enc and TETSponge$[\mathsf{E}, \pi]$.Dec are given in Appendix A, Algorithms 3 and 4. The different cases that the TETSponge mode can encounter are additionally illustrated in Appendix B, Figure 3.

2.2 Clyde-128, a Tweakable LS-Design

The TETSponge mode of operation requires a TBC. We use the Tweakable LS-Design (TLS-design) framework introduced as part of the SCREAM authenticated encryption candidate to the CAESAR competition for this purpose [GLS+14]. TLS-designs are tweakable variants of the LS-designs which specify a family of bitslice ciphers aimed at efficient masked implementations [GLSV14]. Such ciphers work on $n = (s \cdot l)$-bit states, where the size of the S-box is s and the size of the L-box is $2l$. We denote the full cipher state as x, a state row as $x[i, \star]$ $(0 \leq i < s)$ and a state column as $x[\star, j]$ $(0 \leq j < l)$. Concretely, we consider $s = 4$ and $l = 32$. Although the internal representation of the data is a $(s \cdot l)$-bit matrix, the cipher operates over bitstring inputs and outputs. The mapping between a bitstring B and the corresponding bit matrix x is $x[i, j] = B[i \cdot l + j]$.

From an implementation viewpoint, the S-boxes and L-boxes are defined such that they can always be executed thanks to simple operations on the rows (typically corresponding to processor words). The $2l$-bit L-boxes are slightly different from (l-bit) L-boxes that were used in the original LS-designs. As will be clear in Section 2.4, they enable a better branch number at limited cost. In summary, Clyde-128 (illustrated in Appendix C) updates the n-bit state x by iterating N_s steps, each of them made of two rounds (so $N_r = 2N_s$).

One significant advantage of these designs is their inherent simplicity: they can be described in few lines, as illustrated in Algorithm 1, where μ denotes the plaintext, TK a

Algorithm 1 TLS-design with $2l$-bit L-box and s-bit S-box ($n = s \cdot l$)

$x \leftarrow \mu \oplus TK(0)$; ▷ x is a $s \times l$ bits matrix
for $0 \leq \sigma < N_s$ **do**
  for $0 \leq \rho < 2$ **do**
    $r = 2 \cdot \sigma + \rho$; ▷ Round index
    for $0 \leq j < l$ **do**
      $x[\star, j] = \mathsf{S}(x[\star, j])$; ▷ S-box Layer
    for $0 \leq i < s/2$ **do**
      $(x[2i, \star], x[2i+1, \star]) = \mathsf{L}(x[2i, \star], x[2i+1, \star])$; ▷ L-box Layer
    $x \leftarrow x \oplus W(r)$; ▷ Constant addition
  $x \leftarrow x \oplus TK(\sigma + 1)$; ▷ Tweakey addition
return x

combination of the master key K and tweak T that we call tweakey [JNP14], $W(r)$ are round constants, and S and L are an s-bit S-box and a $2l$-bit L-box.[4]

We use SCREAM's lightweight tweakey scheduling algorithm [GLS+14]. It takes the n-bit key K and the n-bit tweak T as input. The tweak is divided into $n/2$-bit halves: $T = t_0 \| t_1$. Then, three different tweakeys are used every three steps as follows:

$$\begin{aligned} TK(3i) &= K \oplus (t_0 \| t_1), \\ TK(3i+1) &= K \oplus (t_0 \oplus t_1 \| t_0), \\ TK(3i+2) &= K \oplus (t_1 \| t_0 \oplus t_1). \end{aligned}$$

The tweakeys can be computed on-the-fly using a linear function ϕ, corresponding to multiplication by a primitive element in $GF(4)$ (with $\phi^2(x) = \phi(x) \oplus x$, and $\phi^3(x) = x$):

$$\begin{aligned} \phi :\ x_0 \| x_1 \ &\mapsto\ (x_0 \oplus x_1) \| x_0, \\ \delta_0 &= T, \\ \delta_{i+1} &= \phi(\delta_i), \\ TK(i) &= K \oplus \delta_i. \end{aligned}$$

2.3 Shadow-512, a Multiple LS-Design

The $\mathsf{TETSponge}$ mode of operation also requires a (larger) permutation. We use a simple variant of the LS-designs that we denote as mLS-designs (standing for multiple LS-designs) for this purpose. In summary, mLS-designs mix multiple LS-designs thanks to an additional diffusion layer. Such ciphers work on $n = (m \cdot s \cdot l)$-bit states, where m is the number of LS-designs considered, the size of the S-box is s and the size of the L-box is $2l$. Taking similar notations as for TLS-designs, we denote the full cipher state as x, each $(s \cdot l)$-bit substate corresponding to an LS-design as a bundle $x[b, \star, \star]$ $(0 \leq b < m)$, a bundle row as $x[b, i, \star]$ $(0 \leq i < s)$ and a bundle column as $x[b, \star, j]$ $(0 \leq j < l)$. Concretely, we will consider $m = 4$, $s = 4$ and $l = 32$. Again, the internal representation of the data is an $(m \cdot s \cdot l)$-bit state but the cipher operates over bitstring inputs and outputs. The mapping between a bitstring B and a state x is $x[b, i, j] = B[b \cdot l \cdot s + i \cdot l + j]$.

In summary, $\mathsf{Shadow\text{-}512}$ (illustrated in Appendix C) updates the n-bit state x by iterating N_s steps, each of them made of two different rounds (denoted as A and B): they respectively apply an L-box to the rows of each bundle independently, and a diffusion layer mixing the rows of different bundles (on top of the S-box layer). An accurate description is given in Algorithm 2, where μ denotes the input, $W(r)$ are round constants, S and L are an s-bit S-box and a $2l$-bit L-box and D is the m-bit diffusion layer.

[4] For regularity (in hardware implementations), we keep the linear L-box in the last round.

Algorithm 2 mLS-design with $2l$-bit L-boxes and s-bit S-boxes ($n = m \cdot s \cdot l$)

```
x ← μ;                                                    ▷ x is a m × s × l bits matrix
for 0 ≤ σ < N_s do
    for 0 ≤ b < m do                                      ▷ Round A
        for 0 ≤ j < l do
            x[b,⋆,j] = S(x[b,⋆,j]);                       ▷ S-box Layer
        for 0 ≤ i < s/2 do
            (x[2i,⋆], x[2i+1,⋆]) = L(x[2i,⋆], x[2i+1,⋆]); ▷ L-box Layer
    x ← x ⊕ W(2·σ);                                       ▷ Constant addition
    for 0 ≤ b < m do                                      ▷ Round B
        for 0 ≤ j < l do
            x[b,⋆,j] = S(x[b,⋆,j]);                       ▷ S-box Layer
    for 0 ≤ i < s do
        for 0 ≤ j < l do
            x[⋆,i,j] = D(x[⋆,i,j]);                       ▷ Diffusion Layer
    x ← x ⊕ W(2·σ+1);                                     ▷ Constant addition
return x
```

2.4 Clyde-128 and Shadow-512 components

We now describe the components S, L and D and the round constants used in Clyde-128 and Shadow-512. Both ciphers are designed to enable simple (software and hardware) implementations based on 32-bit word-level operations. For the S-box, we provide its circuit representation (which can be applied in parallel to the 32 bits of a word). For the L-box and diffusion layer, we provide a sequence of 32-bit operations. We denote the bitwise AND as $\odot$ and the left rotation of a word x by α bits as $\mathsf{rot}(x, \alpha)$.

S-box. We use a variant of the 4-bit S-box used in Skinny [BJK+16], modified by replacing the NOR gates by AND gates. It is given in Table 1, with numbers representing bitstrings encoded in little-endian. That is, $x = \sum_{i=0}^{3} 2^i \cdot x[i]$ and $S(x) = \sum_{i=0}^{3} 2^i \cdot y[i]$. It has linear square correlation and differential probabilities 2^{-2} and algebraic degree 3.

Table 1: S-box in table representation.

x	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\mathsf{S}(x)$	0	8	1	15	2	10	7	9	4	13	5	6	14	3	11	12

Concretely, $y = \mathsf{S}(x)$ can be implemented serially with 4 AND gates and 4 XOR gates in the direct and inverse directions. In the direct sense, it has an AND depth of two and allows computing the two first (and two last) AND gates in parallel:

- $y[1] = (x[0] \odot x[1]) \oplus x[2]$,
- $y[0] = (x[3] \odot x[0]) \oplus x[1]$,
- $y[3] = (y[1] \odot x[3]) \oplus x[0]$,
- $y[2] = (y[0] \odot y[1]) \oplus x[3]$.

The S-box is illustrated in Appendix D and its inverse is given in Appendix E.

L-box. We use an interleaved L-box that applies jointly to pairs of 32-bit words and has branch number 16 over those pairs. Denoting the words on which it applies as x and y:

$$(a, b) = \mathsf{L}'(x, y) = \begin{pmatrix} \mathsf{circ}(\mathtt{0xec045008}) \cdot x^\mathsf{T} \oplus \mathsf{circ}(\mathtt{0x36000f60}) \cdot y^\mathsf{T} \\ \mathsf{circ}(\mathtt{0x1b0007b0}) \cdot x^\mathsf{T} \oplus \mathsf{circ}(\mathtt{0xec045008}) \cdot y^\mathsf{T} \end{pmatrix},$$

where circ denotes the circulant matrix whose first line is given in hexadecimal notation, so that the number $b = \sum_{i=0}^{31} 2^i b_i$ corresponds to the row vector $(b_0, \ldots, b_{31})$.

Concretely, this L-box can be efficiently implemented (in the direct and inverse directions) thanks to six word-level (left) rotations and six 32-bit XORs per word as follows:

- $a = x \oplus \mathsf{rot}(x, 12)$,
- $a = a \oplus \mathsf{rot}(a, 3)$,
- $a = a \oplus \mathsf{rot}(x, 17)$,
- $c = a \oplus \mathsf{rot}(a, 31)$,
- $a = a \oplus \mathsf{rot}(d, 26)$,
- $a = a \oplus \mathsf{rot}(c, 15)$,
- $b = y \oplus \mathsf{rot}(y, 12)$,
- $b = b \oplus \mathsf{rot}(b, 3)$,
- $b = b \oplus \mathsf{rot}(y, 17)$,
- $d = b \oplus \mathsf{rot}(b, 31)$,
- $b = b \oplus \mathsf{rot}(c, 25)$,
- $b = b \oplus \mathsf{rot}(d, 15)$.

The L-box is illustrated in Appendix D and its inverse is given in Appendix F. As previously mentioned, such an interleaved L-box differs from the one used in the original LS-designs (which works on l bits rather than $2l$). The motivation for this choice is a better branch number at limited implementation cost. Precisely, the best known non-interleaved 32-bit L-box has branch number 12 and we reach 16 with this new solution. Exploiting such an interleaved L-box implies that the S-boxes must have an even number of bits.

Diffusion layer (for Shadow-512 only). We use the diffusion layer of the low-energy cipher Midori [BBI+15], which is based on a near-MDS 4×4 matrix defined as follows:

$$(a, b, c, d) = \mathsf{D}(w, x, y, z) = \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix} \cdot \begin{pmatrix} w \\ x \\ y \\ z \end{pmatrix} .$$

It has branch number 4 (an MDS diffusion would provide 5), is illustrated in Appendix D, and can be implemented with six 32-bit XORs as a circuit with gate depth 2 as follows:

- $u = w \oplus x$,
- $a = x \oplus v$,
- $c = u \oplus z$,
- $v = y \oplus z$,
- $b = w \oplus v$,
- $d = u \oplus y$.

Round constants. Round constants for Clyde-128 are generated from a 4-bit LFSR. Each state of the LFSR is used as the constant for a single round. The four bits are XORed with the first bit of the four state rows. Precisely, the round constants are:

• Round 0: (1,0,0,0), • Round 1: (0,1,0,0), • Round 2: (0,0,1,0), • Round 3: (0,0,0,1),
• Round 4: (1,1,0,0), • Round 5: (0,1,1,0), • Round 6: (0,0,1,1), • Round 7: (1,1,0,1),
• Round 8: (1,0,1,0), • Round 9: (0,1,0,1), • Round 10: (1,1,1,0), • Round 11: (0,1,1,1).

For Shadow-512, we take exactly the same constants, but for each bundle $b = 0, \ldots, m-1$, we add the round constant on the bth bit of the four bundle rows, that is $x[b, \star, b]$.

3 Security analysis and claims

We assume that all keys (so both the secret K and public p when applicable) are selected uniformly at random, and that related keys are prevented at the protocol level.

3.1 The TETSponge mode of operation

The black box security analysis of TETSponge is proven in the ideal TBC and permutation models.[5] CIML2 is proven under the additional assumption that the long-term key of the TBC cannot be leaked (but all other intermediate values can be leaked in full). The original proofs are made by modeling the strongly protected TBC as a leak-free component. We show in Section 8 that this assumption can be relaxed to a falsifiable unpredictability assumption. CCAmL1 security is proven under an oracle-free and bounded leakage assumption. We refer to [GPPS19b] for details on these assumptions.

Based on the above, the single-user security claims of the mode are summarized in Table 2. The bounds are close to 2^n security and it is expected that the best adversarial strategy is to try breaking the physical assumptions. A detailed discussion of how such physical assumptions translate into heuristic security requirements can be found in [BBC+20]. Informally, the CIML2 bound guarantees that message integrity reduces to the security of the Clyde-128 implementation against Differential Power Analysis (DPA). The CCAmL1 analysis is more subtle, but essentially guarantees that the confidentiality of long messages reduces to the security of single-block messages against Simple Power Analysis (SPA) and the security of the Clyde-128 implementation against DPA.[6]

Table 2: Single-user security claims.

Security model	security (bits)
Plaintext confidentiality with nonce misuse-resilience (mR)	$n - \log n$
Ciphertext integrity with misuse-resistance (MR) & no leakage	$n - \log n$
Plaintext confidentiality with encryption leakages and mR	$\approx n/2$
Ciphertext integrity with full leakages and MR	$\approx n - \log n$

The security claims for the multi-user variant of TETSponge are summarized in Table 3.

Table 3: Multi-user security claims.

Security model	security (bits)	# of users
Plaintext conf. with nonce misuse-resilience (mR)	$n - 2\log n$	$\approx 2^{n-2}$
Ciphertext int. with misuse-resistance (MR) & no leak.	$n - 2\log n$	$\approx 2^{n-2}$
Plaintext conf. with encryption leakages and mR	$\approx n/2$	$\approx 2^{n-2}$
Ciphertext int. with full leakages and MR	$\approx n - 2\log n$	$\approx 2^{n-2}$

No additional restrictions are imposed on the message length. The security bounds in both tables are for the total number of message and associated data blocks to encrypt.

3.2 From mode assumptions to primitives requirements

As always in cryptography, reductions in idealized models do not imply security when the idealized components (e.g., permutations) are instantiated with practical primitives (e.g., Shadow-512). Therefore, it is important to consider the security of the resulting schemes as a whole. The rationale behind the Spook design can be outlined as follows.

For integrity against leakage, the minimum requirements are that (*i*) the Clyde-128 TBC is a strongly unpredictable block cipher, as discussed in Section 8, and (*ii*) the hash function corresponding to Figure 1 where B is public and with output $U||V$ is collision

[5] Modeling permutations as ideal ones is necessary for the leakage analyzes in [GPPS19b] but black box proofs under a strong pseudo-random permutation assumption should be feasible as in [BGP+20].

[6] The birthday bound for CCAmL1 could be improved. We are not aware if matching attacks.

resistant: this is because the integrity guarantees of Spook are in the unbounded leakage model (where all intermediate values are leaked in full) with nonce misuse, so that the difficulty to forge valid tags Z depends on the difficulty to find collisions for $U||V$.

For the confidentiality of Spook against leakage, the minimum requirement is that the combination of the Clyde-128 TBC with iterations of the Shadow-512 permutation put in a duplex sponge construction yields pseudo-random outputs for the rate part of the construction. This is because Spook only guarantees confidentiality in the nonce misuse-resilience setting, so that for the challenge plaintext, the ephemeral key B is always fresh.[7] We note that we do not make any claims in line with the hermetic sponge strategy. That is, we do not consider distinguishers for Shadow-512 as valid attacks, as long as they do not directly break actual security requirements of the resulting mode.

3.3 The Clyde-128 (tweakable) block cipher

We next discuss the resistance of Clyde-128 with regards to several attack vectors that could break its unpredictability (when taken as a stand-alone primitive) or pseudo-randomness (in combination with the Shadow-512 permutation).[8] We recall that the tweak of Clyde-128 is either constant (as a zero vector or a public value) or pseudo-random and out of adversarial control (for the tag generation). So while a standard TBC requires security against chosen-tweak attacks, the number of rounds selected for Clyde-128 only corresponds to single-key and random-tweak security. Chosen-tweak security for Clyde-128 could be obtained by doubling the number of rounds, following the approach in [GPPR11].

Differential and linear attacks. The security of Clyde-128 against linear and differential attacks can be analyzed thanks to the wide-trail strategy [DR01]. As usually, we restrict to analyze average probabilities/square correlations of characteristics and for this assume independent round keys. As mentioned above, the L-box has differential branch number 16 over pairs of bits entering the same S-box. This implies that any characteristic over any step (two consecutive rounds) has at least 16 active S-boxes. Recall that the maximal differential probability and maximal square linear correlation for our S-box is 2^{-2}. As a result, eight rounds (four steps) lead to an upper bound on the expected probability of any differential characteristics of $(2^{-2})^{4 \cdot 16} = 2^{-128}$. Similarly, the average square correlation of any linear characteristic over the same number of steps is bounded to $(2^{-2})^{4 \cdot 16} = 2^{-128}$ as well. Our recommended parameters add four rounds (two steps) to prevent improvements of these standard attacks (e.g., multiple approximations, improved guessing).

There exist several advanced variants of differential and linear attacks. We next briefly explain why we think that they do not pose a threat for the security of Clyde-128.

Boomerang and differential-linear attacks. Those variants are in particular promising if the probability of differentials and the correlations of linear approximations are high for a small number of rounds but decrease very fast when increasing the number of rounds. As we use the wide trail strategy, this is not the situation for our construction. For Clyde-128 the number of active S-boxes increases rather linearly in the number of rounds.

Truncated differentials. Truncated differentials aim at predicting not the exact difference, but only a pattern in the output difference. One interesting special case is the one of a single bit difference and predicting a single bit in the output difference for a few rounds with high probability. This can be seen as a simple diffusion test, but it also gives insights on truncated differentials and, as it is the same in this extreme case, experimental

[7] This is in the single-user setting. Collisions on B can happen in the multi-user setting but are not security-damaging since the nonce N and public key p are put as inputs of the permutation.

[8] In practice, unpredictability is believed to be more relaxed than pseudo-randomness [DS09b].

Table 4: Division Properties for Clyde-128 over r rounds, for $r \in \{1, \ldots, 8\}$ out of 12. The input division property is `0x7fffffffffffffffffffffffffffffff`.

r	Output Division Property
1	`0xfffffffffffffffffffefffffffeffff`
2	`0xf7ffffffffffbfffeefffddfffffffff`
3	`0x5ffffef7ffbfbfff2fffff65cdfbfffd`
4	`0x1befaa64f3fbefffc9024f49f5301e15`
5	`0x04180406d83e8e9f0000000000000000`
6	`0x410280002c0401010000000000000000`
7	`0x20001000200020000000000000000000`
8	`0x00000020000100000000000000000000`

differential-linear distinguishers. For Clyde-128 we ran a limited experiment using 2^{30} plaintext pairs for each of the 128 bit input differences, and estimated the bias of any bit in the output difference. For a single round there are truncated differentials with probability one. Stated equivalently, not every output bit depends on every input bit after one round. For 2 rounds, using 2^{30} data, we estimated the maximal bias to be 2^{-3}. For 3 and more rounds, the available data was not enough to detect statistically significant biases.

Algebraic degree. As for almost any modern block cipher, we do not expect that algebraic cryptanalysis [CP02], that is breaking the cipher by solving non-linear equations, poses any threat to Clyde-128. Thus, here we focus on attacks that in particular exploit a limited algebraic degree. Those attacks include classical integral attacks, cube attacks [DS09a] and, more recently and fine-grained, attacks based on division property [Tod15].

For the algebraic degree, it is so far out of reach to give meaningful lower bounds on the degree. However, there are quite advanced, and usually rather precise, upper bounds known. The best general upper bound for an SPN cipher is given in [BCC11]. According to those bounds, taking into account that the algebraic degree of our S-box is maximal (i.e., 3), at least five rounds of Clyde-128 are necessary to reach the maximum algebraic degree (i.e., degree 127). Thus, we expect that the recommended twelve rounds (six steps) provide more than sufficient margin to avoid attacks based on low-degree.

Division property. The division property, as introduced by Todo, captures fine grained algebraic structures in ciphers. There are several variants by now, but all of them can be seen as an intermediate step between bounds on the degree on the one hand and computing the entire algebraic normal form (i.e., the exact polynomial representation of the cipher) on the other hand. Ensuring resistance against all possible variations is out of reach today. To estimate Clyde-128 resistance against attacks based on division properties, we used the tool based approach proposed in [XZBL16]. The idea is to build a MILP model for the division property and solve the resulting optimization problem using an off-the-shelf solver. For Clyde-128, this resulted in a distinguisher for eight rounds (four steps). In particular, Table 4 lists one division property for every number of rounds $1 \leqslant r \leqslant 8$. For these, the starting point is always `0x7fffffffffffffffffffffffffffffff`, implying that the plaintext set contains all 2^{127} plaintexts, where the MSB is not set. The final division properties from the table denote the balanced bits (all one bits) in the output after r rounds. Again, we assume that the four additional rounds ensure a sufficient security margin.

Invariant subspace attacks. A successful cryptanalysis method for previous LS-designs are *invariant subspace attacks* [LMR15]. Here, an adversary tries to identify a coset of a linear subspace $U + a$ which gets mapped to another coset $U + b$ by the round function.

Table 5: Dimensions of W_L for Clyde-128's round constants and different No. steps/rounds.

No. steps/rounds	3/6	4/8	5/10	6/10
$\dim W_L$	96	128	128	128

If such cosets exist, the interleaved key addition can translate the coset $U + b$ back to $U + a$, if round keys from $U + (a + b)$ are used. These round keys, leading to an iterative application of the invariant subspace property, are thus called *weak keys* – and the overall invariant subspace attack is a weak key attack. A generalization of this attack is the *nonlinear invariant* or *invariant set* attack [TLS16]. It generalizes invariant subspaces by tracing a set which is invariant under the round transformation rather than a subspace.

Later, in [BCLR17], it was shown that both variants (invariant subspace and invariant set attacks) can be partly thwarted with the right choice of round constants. In particular, any invariant for the linear layer and the round key addition has the linear structures $W_L(c_i)$. By $W_L(c_i)$, we denote the smallest L-invariant subspace that contains all c_i. The c_i are round constant differences for rounds in which the same tweakey is added.

We computed the dimension of W_L for our chosen round constants and different number of rounds. Table 5 lists the corresponding dimensions. Having $\dim W_L = n$ (as is the case from eight rounds / four steps on) implies that any invariant is trivial, namely constant. With [BCLR17, Proposition 2], we can conclude that no non-trivial invariant exists which is at the same time invariant for Clyde-128's S-box layer and its linear layer.

Subspace trails. Subspace trails are an alternative generalization of invariant subspace attacks [GRR16]. They differ in two points to invariant subspaces. First, the subspace U may now vary: in the next round it might be translated to a different subspace V. Second, not only one coset but all $U + a_i$ are mapped to cosets $V + b_i$. While the first property generalizes invariant subspaces, the second restricts the attack. Indeed, [LTW18] showed that subspace trails are a special case of truncated differentials. They further developed an algorithm to bound the length of any probability one subspace trail. Using this algorithmic approach, we bound any subspace trail length for Clyde-128 by three rounds.

3.4 The Shadow-512 permutation

As mentioned in Section 3.2, the confidentiality requirements for the Shadow-512 permutation are difficult to specify exactly since we do not require an ideal permutation and only the combination of Clyde-128 and Shadow-512 must lead to pseudo-random outputs. For performance purposes, we aimed for minimum security requirements. First, we targeted 128-bit security against linear cryptanalysis. This can be analyzed by considering the super S-box structure of Shadow-512. Two rounds activate 16 S-boxes and four rounds activate 16×4 S-boxes thanks to the branch number of the diffusion layer. Hence, a probability bound of 2^{-128} for the best linear characteristic is reached after four rounds.

We ran the same experiment to estimate truncated differentials for a low number of steps as for Clyde-128, using 2^{30} plaintext pairs. For one step there are, as expected due to the super S-box construction, still probability one truncated differentials. For 2 and more steps, the available data was not enough to detect statistically significant biases.

Next, and similar to Clyde-128 as well, we searched for the longest subspace trails through Shadow-512. Here, the algorithm bounds the length of any probability-one subspace trail by five rounds. Finally, we required an algebraic degree 128 which, according to the upper bound in [BCC11], can be reached after at least five rounds of Shadow-512.

Integrity requirements are simpler to state: we require that when included in the TETSponge mode of operation, the Shadow-512 permutation ensures collision resistance for the 255 bits that are used to generate the tag. For this purpose, a minimum requirement is to prevent truncated differentials with probability larger than 2^{-127} for those 255 bits. A simple heuristic for this purpose is to require that no differential characteristic has probability better than 2^{-384}, which happens after twelve rounds (six steps).[9]

3.5 External cryptanalysis

A recent work by Derbez, Huynh, Lallemand, Naya-Plasencia, Perrin and Schrottenloher analyzes Shadow-512 and its integration in Spook. They demonstrate a distinguisher on the full Shadow-512 permutation, and a practical forgery attack against fours steps of Spook in the nonce-misuse setting [DHL+20]. We explain their main results below.

3.5.1 Analysis of Shadow-512

We first define modified round constants $\tilde{\mathsf{W}} = \mathsf{D}^{-1}(\mathsf{W})$, so that one step is composed of:

- operations that operate independently on each bundle: $\mathsf{S}, \mathsf{L}, \mathsf{W}, \mathsf{S}, \tilde{\mathsf{W}}$,
- the inter-bundle mixing D defined in Section 2.4,

where the first part can be seen as four 128-bit boxes σ_0, σ_1, σ_2, σ_3 (we omit the dependency on the round number for simplicity). The following analysis is based on this representation, with the state considered as four 128-bit words (i.e., the bundles of Spook).

Exploiting D. The first result of Derbez et al. is a 5-step distinguisher on Shadow-512 that can be described as a rebound attack [MRST09], using the following probability-1 differential characteristics that exploit the branch number 4 of D:

$$\begin{aligned}
&\text{backwards:} && \begin{bmatrix} * & * & * & 0 \end{bmatrix} \xleftarrow{\sigma} \begin{bmatrix} * & * & * & 0 \end{bmatrix} \xleftarrow{\mathsf{D}} \begin{bmatrix} 0 & 0 & 0 & * \end{bmatrix} \xleftarrow{\sigma} \begin{bmatrix} 0 & 0 & 0 & \alpha \end{bmatrix} \xleftarrow{\mathsf{D}} \begin{bmatrix} \alpha & \alpha & \alpha & 0 \end{bmatrix}, \\
&\text{forwards:} && \begin{bmatrix} \beta & \beta & \beta & 0 \end{bmatrix} \xrightarrow{\mathsf{D}} \begin{bmatrix} 0 & 0 & 0 & \beta \end{bmatrix} \xrightarrow{\sigma} \begin{bmatrix} 0 & 0 & 0 & * \end{bmatrix} \xrightarrow{\mathsf{D}} \begin{bmatrix} * & * & * & 0 \end{bmatrix} \xrightarrow{\sigma} \begin{bmatrix} * & * & * & 0 \end{bmatrix}.
\end{aligned}$$

Starting from the middle, the inbound phase of the rebound attack builds a pair of values corresponding to the transition $\begin{bmatrix} \alpha & \alpha & \alpha & 0 \end{bmatrix} \to \begin{bmatrix} \beta & \beta & \beta & 0 \end{bmatrix}$ over σ. Going though the outbound phase, this defines a pair with input and output difference in a dimension-128 subspace. This result can be extended to 6 steps, using sparse values α, β so that the trail can cover one more S layer with high probability, and doing the inbound phase over SDS.

Exploiting sparse constants. The analysis of Derbez et al. also shows that the sparse constants in Shadow-512 have an unfortunate interaction with the D operation. More precisely, there is a high probability that $\sigma_i(x) = \sigma_j(x)$ with $i \neq j$. Indeed W only affects the bth bit of the rows of bundle b, so that after the W operation, the state in σ_i and σ_j only differs in the ith and jth bits. Moreover, $\tilde{\mathsf{W}}$ affects bits $\{0, 1, 2, 3\} \setminus \{b\}$ of bundle b, so that the difference between σ_i and σ_j is also only in the ith and jth bits. With some probability the difference introduced by W is corrected by the difference in $\tilde{\mathsf{W}}$ (with only the S operation in between), and we obtain $\sigma_i(x) = \sigma_j(x)$. This type of property exploiting sparse round constants is related to previous works such as internal differential attacks [Pey10], rotational cryptanalysis [KN10], self-similarity [BDLF10], or invariant subspaces [LAAZ11, LMR15]. In the case of Shadow-512, the probability of

[9] The rationale behind this heuristic is that if there exist a truncated differential characteristic with a zero difference on the 255 bits of output, it corresponds to the sum of 2^{257} fully specified differential characteristics, and at least one them must have probability higher than $2^{-127}/2^{257} = 2^{-384}$.

having $\sigma_i(x) = \sigma_j(x)$ depends on the step number and the corresponding constants: as shown in [DHL+20], the probability is 2^{-4} at step 3, 2^{-6} at step 4, but zero for other steps. When $\sigma_i(x) = \sigma_j(x)$, various types of symmetry (or subspaces) can propagate through the round function with high probability, because D also preserves equality of bundles. Using symmetric states, it is possible to extend this distinguisher to 7 steps of Shadow-512.

3.5.2 Analysis of reduced Spook

The same properties can be used to attack the integrity of a reduced version of Spook with repeated nonces. In this setting, an attacker knows the value of the outer part of the sponge state, and he tries to construct two messages leading to a collision. After querying the oracle on one of the colliding messages, he can generate a forgery for the second one. Derbez et al. assume a reduced-round version keeping rounds 2 to 5, because the probability that $\sigma_0(x) = \sigma_1(x)$ is high at rounds 3 and 4. Since the outer part of the sponge is known, they start with a pair of messages leading to states:

$$\begin{bmatrix}\sigma_0^{-1}(x) & \sigma_1^{-1}(x) & u & v\end{bmatrix}, \quad \begin{bmatrix}\sigma_0^{-1}(y) & \sigma_1^{-1}(y) & u & v\end{bmatrix},$$

where u, v are unknown values in the inner part of sponge state, and x, y are chosen arbitrarily. After the σ layer, they obtain:

$$\begin{bmatrix}x & x & \sigma_2(u) & \sigma_3(v)\end{bmatrix}, \quad \begin{bmatrix}y & y & \sigma_2(u) & \sigma_3(v)\end{bmatrix}.$$

After the D operation, this leads to:

$$\begin{bmatrix}x' & x' & u' & v'\end{bmatrix}, \quad \begin{bmatrix}y' & y' & u' & v'\end{bmatrix}.$$

With high probability, we then have $\sigma_0(x') = \sigma_1(x')$ and $\sigma_0(y') = \sigma_1(y')$, so that the equality between the first two bundles is preserved through round 3, and similarly through round 4, leading to states:

$$\begin{bmatrix}x'' & x'' & u'' & v''\end{bmatrix}, \quad \begin{bmatrix}y'' & y'' & u'' & v''\end{bmatrix}.$$

Finally, there is a high probability that $\sigma_0(x'') \oplus \sigma_1(x'') = \sigma_0(y'') \oplus \sigma_1(y'')$, so that there is no difference in the inner part of the output. According to the analysis of [DHL+20], the total probability to find the required collision on the targeted rounds is $2^{-24.8}$. This analysis resembles a truncated differential attack, but it also differs significantly. Indeed, the probability of the characteristic is only high when two bundles have the same value (i.e., when two σ boxes share the same input), which does not happen with random pairs of inputs. In particular, it does not contradict our analysis of differentials and truncated differentials, because our bound on the probability of differentials assumes random input pairs.

3.5.3 Impact

As mentioned by the authors of [DHL+20], neither the Shadow-512 distinguisher of Section 3.5.1 nor the collision attack of Section 3.5.2 threaten the confidentiality or integrity of the full Spook. However, the collision attack highlights that the heuristic used to select the number of rounds of Shadow-512 in Section 3.4 is not conservative. We discuss tweaks in order to improve security margins against this attack in Section 7.

4 Primary candidate and variants

Underlying primitives. We consider two sets of parameters for the Clyde-128 TBC and Shadow-512 permutation. The *recommended parameters* are 12 rounds for Clyde-128 and

12 rounds for Shadow-512. We also provide *aggressive parameters*, with 12 rounds for Clyde-128 and 8 rounds for Shadow-512, as a cryptanalysis target (not recommended for practical use). Our reference implementations and test vectors are based on recommended parameters. We note that the collision attack of [DHL+20] (cf. Section 3.5.2) nearly breaks the aggressive parameters (it breaks 4 steps but does not start from the 1st round).

Full algorithm. We denote as Spook[128, 512, su] the AEAD algorithm operating TET-Sponge in the single user setting with Clyde-128 as TBC and Shadow-512 as permutation, and as Spook[128, 512, mu] its multi-user version. Based on these notations, we define a:

- **Primary candidate** as Spook[128, 512, su] with recommended parameters.
- **First variant** as Spook[128, 512, mu] with recommended parameters.

We recall that the only difference between the single-user and multi-user versions of Spook is that the public tweak p is stuck at zero in the first case (i.e., the key is limited to 128 secret bits), and picked up at random in the second one (i.e., the key is made of 128 secret bits and 126 public bits). We additionally define two smaller versions of Spook with a 384-bit state. They are obtained by turning the 512-bit permutation into a 384-bit one. We do so by defining Shadow-384 as a 3LS-design (rather than a 4LS-design) where the diffusion layer $(a, b, c) = \mathsf{D}(x, y, z)$ is specified as:

- $a = x \oplus y \oplus z$,
- $b = x \oplus z$,
- $c = x \oplus y$.

The rest of the permutation and the other elements of the mode are adapted so that $r = 128$, with the same number of rounds for the parameters, leading to our:

- **Second variant** as Spook[128, 384, su] with recommended parameters.
- **Third variant** as Spook[128, 384, mu] with recommended parameters.

5 Rationale: design trade-offs, advantages & limitations

Spook is an AEAD algorithm with state-of-the-art guarantees in the black box setting. Namely, it ensures beyond-birthday security with respect to the block size of its underlying TBC, can be extended to multi-user security with a public tweak, and provides nonce misuse-resilience in the sense of Ashur et al [ADL17]. Thanks to its one-pass structure, Spook should allow efficient implementations on a wide range of platforms. Its design is in particular well-suited to 32-bit software implementations (thanks to an intensive exploitation of 32-bit word-level operations), and to dedicated hardware and FPGA implementations (thanks to the low gate complexity and limited depth of its different components).

Spook provides excellent opportunities to mitigate physical attacks efficiently thanks to its leakage-resistant features. In particular, the general rationale behind its design enables leveled implementations, where the Clyde-128 TBC is well protected against side-channel attacks and the Shadow-512 (or Shadow-384) permutation is implemented with cheaper protections (or even no protections at all). It is in the specific contexts where physical attacks are an important concern that Spook is expected to exhibit significant performance (e.g., energy) gains compared to modes without leakage-resistant (or resilient) features.

Concretely, protecting the TBC can be achieved thanks to the masking countermeasure, both in hardware [CGLS20] and in software [GR17]. For this purpose, Clyde-128 is designed both with low AND complexity (as previous LS-designs) and low AND depth (which is important to limit the latency of so-called glitch-resistant implementations [NRS11, FGP+18]). As for the permutation, low-latency / low-energy implementations in the sense of [KDH+12] are natural candidates in hardware, while some minimum countermeasures to prevent SPA (e.g., low-order masking, or time randomization [VMKS12]) should be sufficient in software. For this purpose, the Shadow-512 (or Shadow-384) permutation is designed with low-latency components. Leveled implementations of Spook can also benefit from pre-computing the (expensive) generation of fresh seeds if needed.

The main price to pay for the leakage-resistant features of Spook is that it suffers from some overheads in case of short messages. This seems unavoidable in any mode leveraging a re-keying process. However, and as evaluated in [BGP+20, BBC+20], these overheads are amortized as soon as the data to process is a few blocks long, and the gains of leveled implementations can reach factors 10 to 100 (e.g., in energy) if a high physical security level is required by an application. A secondary drawback is the need of two primitives (a TBC and a permutation), which implies a larger cost (i.e., area) in hardware. However, this drawback vanishes for the intended performance metric (i.e., the energy per bit) and case studies, since (*i*) the Clyde TBC is only used for initialization and finalization and can be switched off for the rest of the computations, and (*ii*) leveled implementations require implementations with different physical security levels anyway. Furthermore, in case uniformly (un)protected implementations are considered, the use of the same S-box and L-box in Clyde-128 and Shadow-512 (or Shadow-384) should allow resource sharing. We show in the next section that even in this disadvantageous context, Spook reaches excellent performance levels and the overheads due to the two primitives are small.

Eventually, we list a couple of additional interesting features of Spook.

First, the TETSponge mode is compatible with solutions for the encryption of long messages segmented into several smaller packets, as for example proposed by Bertoni et al. [BDPA11] and formalized by Hoang et al. [HRRV15]. Such a "session feature" can be used as a partial tagging mechanism which allows the decryption of long messages when only a limited memory is available (i.e., smaller than the size of the message), and saves the execution of one TBC per segment (i.e., the highly protected part and therefore more expensive part in a leveled implementation of TETSponge). These modes are not directly compatible with the NIST API. We discuss them (and their adaptation to Spook) in a separate publication [CGP+19]. Second, since leveraging a re-keying process, the Spook algorithm inherently provides good resistance against some Differential Fault Attacks, as discussed in [MSGR10, DEM+17]). Finally, an inverse-free variant of Spook can be obtained by performing the tag verification in the direct sense. It can only satisfy CIML1 in the unbounded leakage model, yet can provide good concrete security against bounded leakages if the tag verification is sufficiently protected (e.g., masked). It is also the natural way to implement Spook if side-channel attacks are not a concern. When such an inverse-free variant is considered, the tag of Spook can be truncated, leading to a standard tradeoff between the mode's integrity guarantees and performances.

More discussions about the high-level design choices and security claims of the Spook authenticated encryption scheme, together with news, updated (unprotected and masked) implementations, mathematical and side-channel cryptanalysis challenges and other additional resources, can be found on the algorithm website https://www.spook.dev/.

6 Unprotected implementation results

In this section, we provide first results of optimized (unprotected) implementations of Spook. Our main objective is to demonstrate that even in this disadvantageous context (which does not take advantage of leveled implementations), Spook has excellent performance figures on a wide range of platforms. In particular, the overheads (in hardware area and embedded software code size) due to the use of two different primitives are shown to be limited thanks to the possibility to share resources (i.e., S-boxes and L-boxes).

6.1 Software implementations

The structure of the (m)LS-design primitives with 32-bit row length makes them very easy to implement on 32-bit platforms. The naive implementation provided in the reference implementation of Spook (see https://www.spook.dev/) is already quite efficient, and the optimized implementations keep the same structure. This implementation is based on storing each row of the (m)LS-designs as a machine word. The L-box is then implemented using rotations and XORs, the S-box in a bitslice fashion (4 ANDs and 4 XORs for a full LS state), and the D-box takes another 24 XORs for the full mLS state.

We focus on two kinds of platforms: high-end (x86_64 with SIMD instructions) and embedded targets (ARM Cortex-M/RISC-V). For portability, we use as much as possible standard C code (with a few common compiler extensions). Common optimizations for both kinds of targets were ensuring properly aligned data layout. We also ensured that relevant function calls and loop unrolling could be inlined by the compiler.

6.1.1 High-end platforms

The Clyde-128 implementation for high-end platforms comes in two flavors: 32-bit (reference implementation with generic optimizations applied) and 64-bit, where a pair of rows is stored in a 64-bit word (interleaving bits of both rows). This implementation can thus perform the L-box using rotations and XORs on a single 64-bit word. The S-box is performed on four 64-bit words, where one out of two bits in each word is not used. This implementation requires to switch representation to and from interleaved-rows registers, which is performed by the `_pdep_u64` and `_pext_u64` instructions.[10] Overall, the 64-bit implementation of Clyde-128 gains about 4 % performance over the 32-bit one.

The Shadow-512 primitive is more interesting: the mLS design gives more opportunities to exploit the parallelism of SIMD instructions. First, we explored the use of 128-bit words, used as 4 times 32 bits, where each 32-bit sub-word is associated to one bundle. The full Shadow-512 state is thus four 128-bit words, one for each row. S-boxes and L-boxes are then easily implemented (using bitwise XORs and ANDs, and 32-bit rotations). The D-box is more challenging to implement, since it mixes sub-words from the same 128-bit word. We do it by using the shuffle primitive, requiring 12 shuffles and 8 XORs for the full 512-bit D-box. The implementation is written using only C code without platform-specific instrinsics, thanks to the `vector` compiler extension supported by GCC and Clang.

We explored implementations of Shadow-512 with larger word sizes. We considered the use of 256-bit AVX2 and 512-bit AVX512 instructions, but this does not translate into significant practical gains. We report performance numbers in Table 6. We observe that performance is significantly better for the Skylake-AVX512 target. This is due to the presence of rotate instructions for 128 bits and 256 bits in its instruction set.

6.1.2 Embedded Software (ARM Cortex/ RISC-V)

The code of our embedded software implementations is written in C with minor changes compared to the reference implementation, chosen to optimize the assembly code generated by the compiler. The round constants are not stored in memory but derived from an LFSR to reduce the code size. The three tweakeys are pre-computed: at each round, the correct one is loaded according to the round index modulo three. In order to avoid (sometimes costly) arithmetic operations, the later is performed by hard-coding a value `0x924` at the beginning of each encryption, which is then right-shifted by two at the end of each round so that its lower bits always correspond to the round index modulo 3.

The performances obtained with optimization flags set for reduced code size and maximum speed are given in Tables 7 and 8. The total code size is reported with the

[10] From the BMI2 instruction set, available first on the Haswell Intel micro-architecture.

Table 6: High-end software performance results. Number of cycles compiled for various micro-architectures, and throughput (cycles per byte) for a message of 2048 bytes.

	x86-64 (SSE2)	Haswell (AVX2)	Skylake-AVX512
Clyde-128 (32-bit)	317	283	283
Clyde-128 (64-bit)		271	271
Shadow-512 (32-bit)	904	457	342
Shadow-512 (128-bit)	409	397	304
Shadow-512 (256-bit)		432	312
Shadow-512 (512-bit)			454
Spook (C32bit-S128bit)	13.3 (per byte)	13.3 (per byte)	10.1 (per byte)

number of cycles to evaluate each primitives. The number of cycles per byte is given for a complete run of Spook on a 2048-byte message. Overall, the Cortex-M3 has better performances than the other MCU's. This comes from the barrel shifter that allows performing rotations and XORs in a single cycle, while the others MCU's need three cycles for it. For the RISC-V implementation, we use the RI5CY core.[11] It leads to faster results than the Cortex-M0 because of its 27 data registers (while the Cortex has 12 data registers). It can therefore hold the whole Shadow-512 in the registers, while the Cortex spends time (and code size) storing and loading that state into memory.

Table 7: Size-optimized performances on embedded platforms (-Os).

	Size [Bytes]	Clyde-128 [Cycles]	Shadow-512 [Cycles]	Spook [Cycles/byte]
Cortex-M0	1936	3274	8626	299
Cortex-M3	1878	1764	5496	187
RI5CY	2138	1853	4731	161

Table 8: Speed-optimized performances on embedded platforms (-O3).

	Size [Bytes]	Clyde-128 [Cycles]	Shadow-512 [Cycles]	Spook [Cycles/byte]
Cortex-M0	4628	2450	6288	205
Cortex-M3	3822	802	2340	77
RI5CY	4618	1259	4062	132

6.2 Hardware implementations

We now present an optimized hardware architecture for the unprotected implementation of Spook. The tag verification is therefore based on the inverse-free variant.

Our main optimization goal is to minimize the amount of logic needed in order to implement Spook, mixing Shadow-512 and Clyde-128 while keeping high performance levels. For this purpose, we perform Shadow-512's Round A and Round B each in multiple clock cycles that operate over a part of the state. We observe that the Round A logic can be re-used to implement the round function of Clyde-128. Therefore, the same logic core can be used for both primitives and the practical impact of Clyde-128 on the overall cost boils down to the one of the logic performing the tweakey update and its addition.

[11] https://github.com/pulp-platform/riscv.

Our architecture is depicted in Figure 2. Each bus is 128-bit long unless indicated otherwise. The IOs (in red) include the nonce, the key, the tag, the bytes to digest (denoted as `Din`) and the digested bytes (denoted as `Dout`). The `Din` bytes can be associated data, plaintext or ciphertext and come from an external block that pads them with 10* when needed. It follows that `Dout` contains bytes either from the ciphertext or the plaintext. The blocks SLW (i.e., S-box then L-box then `W` addition) and SDW (i.e., S-box then D-box then `W` addition) contain the combinatorial logic to perform a round of Clyde-128 and Shadow-512's Round B, respectively. The `pad ciphertext` module is only used during a decryption process to pad the input ciphertext before the latter is used during the following execution of Shadow-512. The values α_0 and α_1 are for the domain separation bits.

To process a call of Clyde-128, the initial plaintext and tweak are respectively stored in the registers $R0$ and $R1$. The control signal `mode_RB` is unset (i.e., equals 0) in contrast with `mode_clyde` that is set in such a way that the state of the TBC cycles through the `R0` register and the SLW logic (and the tweakey addition) at the rate of one round per clock cycle. Therefore, the full Clyde computation takes 12 cycles. The tweak flows through registers `R1`, `R2` and through the ϕ logic, producing a valid updated tweak every two clock cycles. For Shadow-512, the four 128-bit bundles $b_0, \ldots, b_3$ are stored in the registers `R0` to `R3`. Those registers act as circular shift registers with two shifting modes. For Round A, the data cycles from `R3` to `R0`, then through the SLW unit back to `R3`, computing a full round in four cycles (all mux controls are unset). For Round B, data is cycling inside the same `Ri` register: the signal `mode_RB` is set, forwarding the data to the SDW unit (that updates a part of it input and shifts the other part) for $32/N_u$ cycles.

When starting the Spook operation, a clock cycle is required in order to load the values N and 0* needed to initiate the first call of Clyde-128 that computes the fresh seed B. Next, for the first call of Shadow-512, the initial state (i.e., $0^*||N||0^*||B$) is loaded sequentially per bundle at the end of the seed computation (excepted for B), using again the control signals `feed0` and `feedN`. Shadow is then executed, and the digest unit is used a the beginning of each execution when AD/P/C needs to be fed. Finally, the tag computation is initiated by waiting a clock cycle (in Round A mode shifting), which is required in order to have the first two bundles used as plaintext and tweak, respectively. Additionally, the signal `t` is set in order to ensure that the MSB of the tweak is high.

6.2.1 FPGAs

We synthetized the previous architecture on an Artix-7 FPGA (xc7a100tcsg324-3) with Xilinx ISE14.7 toolset. The interface used is a variant of the CAESAR API: the only difference is that the core has a single input channel of 32 bit instead of two.

The main parameter of our investigations is the number of units implemented, next denoted as N_u. Considering the state as four 128-bit bundles, a unit is computing the S-box on the columns with the same index in each bundle (i.e., 4 parallel S-boxes) and applies a D-box over 16 bits to the outcome. By using multiple instance of such units in parallel (N_u can vary from 1 to 32) and combining them with a shift register strategy, computing Round B lasts more or less clock cycles for a variable logic cost.

Implementation results for different values of N_u and optimization goals are shown in Table 9. A run of Shadow-512 is performed in $6\,(4 + 32/N_u)$ cycles. The table includes standard post place-and-route metrics, namely the amount of slices, of registers and of look-up tables required, the clock frequency, the latency, the throughput (for long messages, denoted as TP) and the throughput over area ratio (denoted as TPA). Note that the TPA metric for $N_u = 8$ (which is a natural number of units to balance the cost of Round A and Round B) improves the preliminary results reported in [Beh19] by a factor 10, and does it by improving both the area and throughput, reflecting better architectural choices.

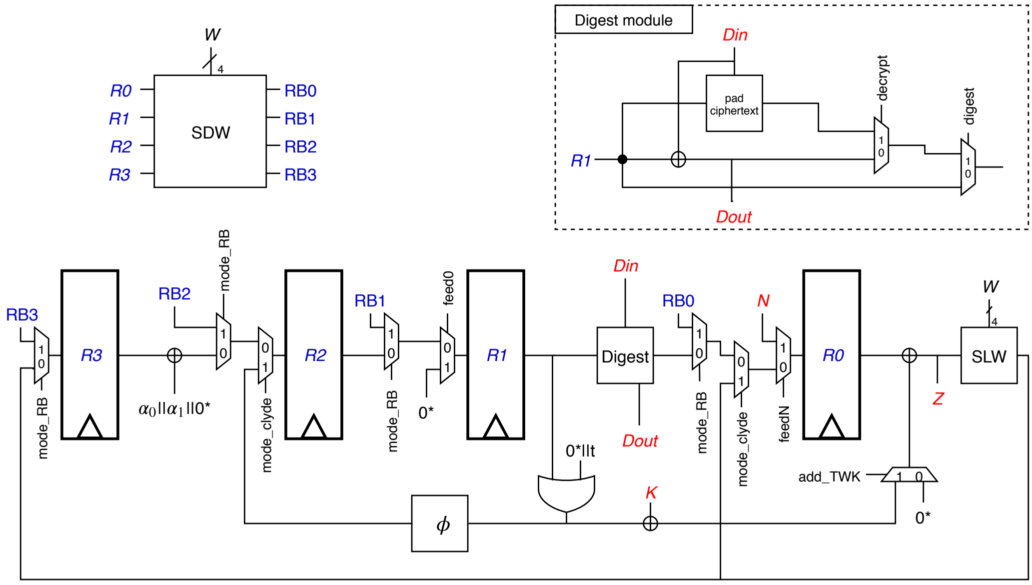

Figure 2: Architecture of Spook[128,512,su] (unprotected, inverse-free variant).

Table 9: Artix-7 implementations results (post place-and-route).

N_u	Opt. Strat.	Slices	Regs	LUTs	Freq. [MHz]	Lat. [Cycles]	TP [Mbps]	TPA [Mbps/LUT]
1	Area	519	1452	1941	138	216	163	0.084
1	Speed	546	1452	2008	171	216	202	0.1
8	Area	549	1449	2039	140	48	746	0.366
8	Speed	568	1449	2112	184	48	981	0.464
32	Area	642	1447	2383	137	30	1169	0.490
32	Speed	660	1447	2441	188	30	1604	0.657

The practical impact of Clyde-128 is assessed by running a synthesis using the same optimisation parameters with the logic exclusively related to Clyde-128 removed. As shown in Table 10, it appears that the implementation results obtained with and without Clyde-128 for our architecture with $N_u = 8$ only differ by 261 LUTs for both optimization strategies. As for cycles overheads, Clyde-128 is implemented in 12 cycles in our architecture, so we need (24+1) cycles corresponding to the initial/final Clyde and one cycle for the interface. This could be further reduced by using exploiting more parallelism if needed. Overall, these results confirm Spook's excellent opportunities of resource sharing.

6.2.2 ASICs

In this subsection, we finally present a first optimized ASIC implementation of unprotected Spook in a 65nm technology, adopting the $N_u = 8$ level of parallelization. The numbers we provide are based on a classic design flow, performed with the TSCM-N65LP (low-power) design kit, adopting *Cadence Genus 16.12-s027* for the synthesis and *Cadence Innovus 16.10* for the place-and-route steps. We have used the clock gating option for the synthesis in order to reduce the dynamic power consumption when the datapath of the Spook processor is in idle. The design reached a density of 93%. In Table 11, the overall cost of the ASIC implementation is reported. In Table 12, the impact of the clock

Table 10: Practical cost of Clyde-128 on top of Shadow-512 ($N_u = 8$).

Opt. Strat.	Slices w/o Clyde-128	Δ Slices	Regs w/o Clyde-128	Δ Regs	LUTs w/o Clyde-128	Δ LUTs
Area	524	25 (4.7%)	1447	2 (≈ 0%)	1862	177 (9.5%)
Speed	518	50 (9.6%)	1447	2 (≈ 0%)	1948	164 (8.4%)

Table 11: ASIC implementation results (post place-and-route) with $N_u = 8$.

Instance	Area [kGE]	Max. Freq [MHz]	Power [mW]	Throughput [Mbps]	Energy [pJ/bit]
Spook[128,512,su]	17.5	416	8.27	2218.6	3.72

frequency on the efficiency and power consumption of our architecture is detailed. It can be seen from Table 12 that our proposed implementation is remarkably energy-efficient in the 10MHz to 100MHz range, while the energy penalty for using the maximum clock frequency is very limited. Note that this range of frequencies typically covers the ones used in most IoT applications and RFID devices. At lower frequency, the static power consumption is dominant, and the energy per bit increases significanlty as well (although further optimizations could be investigated for this specific context).

7 Tweak proposals

While the analysis of [DHL+20] does not target the full Spook AEAD, it exploits two design choices in Shadow that may be improved with simple changes: (*i*) the round constants are sparse and affect only one S-Box per bundle, and (*ii*) the branch number of D is only 4. In Section 7.1, we discuss tweaks that strengthen Shadow. Our rationale is that they should increase the security more efficiently than a direct increase of the number of rounds. Next, in Section 7.2, we discuss additional tweaks to improve performances, based on the finer-grain understanding of software and hardware implementations that the previous section enables. We conclude this section by proposing Spook v2, a variation of Spook improving its security margins at the cost of minimum performance overheads.

7.1 Improving security margins

Based on the analysis of [DHL+20], two natural approaches to improve the security margins of Spook are to use denser round constants in Shadow and to improve the D transform. Changing the round constants is a more ad hoc change that primarily affects the collision attack while improving the D transform is a more general improvement that also mitigates the distinguisher. We therefore propose to replace the binary D by an efficient MDS matrix proposed in [DL18]. Precisely, we propose to use the $M^{8,3}_{4,6}$ matrix for Shadow-512 and $M^{5,1'}_{3,4}$ for Shadow-384. In both cases, we work in the 32-bit ring of polynomials modulo $x^{32} + x^8 + 1$ with the constant factor x. The coefficients of the polynomials are the bits of an LS row, the low-degree coefficient corresponding to the column $i = 0$. This change weakens the cryptanalysis of [DHL+20] for two reasons: equality of two bundles is no longer preserved through D and diffusion is improved. This limits the symmetry properties in Shadow and increases the bound on the number of active S-Boxes for differential & linear characteristics: we now have 80 active S-Boxes (rather than 64) every four rounds for Shadow-512, and 64 ones (rather than 48) every four rounds for Shadow-384. We evaluated that the cost of this change is minimal in hardware and implies roughly 15% of cycles

Table 12: Impact of clock frequency on the ASIC results with $N_u = 8$.

Frequency [MHz]	Power [mW]	Throughput [Mbps]	Energy [pJ/bit]
416	8.27	2218.6	3.72
400	8.04	2133.3	3.76
333	6.66	1776	3.75
100	1.96	533.3	3.675
10	0.166	53.3	3.675
0.1	0.004	0.53	7.5

overheads in embedded software implementations (e.g., Cortex-M3, RI5CY). Note that we still do not claim that the Shadow permutation is hermetic. In particular, we expect that rebound distinguishers based on the diffusion properties of L-box could reach up to 10 or 12 rounds, but this type of inside-out property in not applicable to the mode.[12]

7.2 Improving efficiency

The original constant addition of Shadow adds a single-bit constant to the four 32-bit words of each bundle. This limits the efficiency for software implementations, as each touched word requires a fixed amount of operations independently of the actual number of bits of the constant. By grouping constant additions to fewer state words, the execution time of Spook on microprocessors can decrease by more than 10 % while at the same time enabling denser constants. New constants for Shadow taking advantage of this observation are as follows: for the round A, a constant word is added to the second row of each bundle (so that we need four 32-bit constants for Shadow-512 and three ones are for Shadow-384); for the round B, a constant word is added to all the rows of the first bundle. The 32-bit constants are obtained from the state of a 32-bit LFSR. Using the same representation as for the MDS D-box, the shifting of the LFSR is the multiplication by x modulo $x^{32} + x^7 + x^6 + x^2 + 1$. The first round constant is obtained by shifting this LFSR 1024 times.

Besides, we also change the input of the first Shadow-512 call to $B||P||0||N||0^*$, which improves the efficiency (both in cycles and area) of the hardware implementation.

7.3 Spook v2 and performance overheads

The Spook v2 proposal combines the two improvements presented above for Shadow (Clyde-128 is left unchanged). It reaches similar performances as Spook v1: the cost increase due to the MDS D-box is compensated by the cost reduction due to the change of constants. For completeness, we refer to Appendix G for updated figures and details about the Spook v2 design, and to Appendix H for software and hardware implementation figures.

8 Relaxing the leak-free assumption

Leveled implementations aim to exploit the possibility that different components of an encryption process may need different levels of protection against physical attacks that come with different costs. Based on this observation, the security analysis of Spook in the presence of leakage, as offered in [GPPS19b], is based on the assumption that the TBC is strongly protected against side-channel analysis, while the permutation is only weakly protected. This previous work, following others [PSV15, BKP+18, BPPS17, BGP+20],

[12] Rebound distinguishers based on properties of L are less powerful than distinguishers based on properties of D, because input/output differences do not match the bundle limits, and the distinguisher would typically start from a B round in order to reach the maximum number of rounds.

modeled this strongly protected TBC as leak-free. Yet, the use of such an idealized assumption is not desirable from a practical standpoint. We next show that it is sufficient to assume that the TBC provides strong unpredictability in the presence of leakage, which is a weaker and empirically falsifiable assumption. For this purpose, we adapt a recent result from Berti et al. on leakage-resilient MACs to TETSponge and Spook [BGP+19].

8.1 Strong Unpredictability with Leakage

We denote by $\mathsf{L} = (\mathsf{L}_{\mathsf{Eval}}, \mathsf{L}_{\mathsf{Inv}})$ the leakage function pair associated to an implementation of the TBC, $\mathsf{E} : \mathcal{K} \times \mathcal{T} \times \mathcal{X} \longmapsto \mathcal{Z}$, where $\mathsf{L}_{\mathsf{Eval}}(k, tw, x)$ (resp., $\mathsf{L}_{\mathsf{Inv}}(k, tw, z)$) is the leakage resulting from the computation of $\mathsf{E}_K(T, X)$ (resp., $\mathsf{E}^{-1}_K(T, Z)$). We also allow the adversary $\mathcal{A}$ to profile the leaking device, which we write as $\mathcal{A}^{\mathsf{L}}$, following [PSV15]: when querying this oracle, $\mathcal{A}$ must provide all three inputs of E and E^{-1}, including the key. Note that this is necessary since the adversary does not know the leakage function and can only access to the leakage function through the device, in an oracle manner [SPY13]. Despite the leakage, it should be hard to guess a fresh TBC triple (X, T, Z). For this purpose, we extend the definition of Berti et al. [BGP+19] in the multi-user setting.

Definition 1 (muSUL2)**.** Given the implementation of a tweakable block cipher $\mathsf{E} : \mathcal{K} \times \mathcal{T} \times \mathcal{X} \longmapsto \mathcal{Z}$ with leakage function pair $\mathsf{L} = (\mathsf{L}_{\mathsf{Eval}}, \mathsf{L}_{\mathsf{Inv}})$, the *multi-user strong unpredictability advantage with leakage* of an adversary $\mathcal{A}$ against E with u users is:

$$\mathbf{Adv}^{\mathsf{muSUL2}}_{\mathcal{A},\mathsf{E},\mathsf{L},u} := \Pr\big[\mathsf{muSUL2}_{\mathcal{A},\mathsf{E},\mathsf{L},u} \Rightarrow 1\big],$$

where the security game muSUL2 is defined in Table 13.

Table 13: Strong unpredictability with leakage in evaluation and inversion experiment.

$\mathsf{muSUL2}_{\mathcal{A},\mathsf{E},\mathsf{L},u}$ experiment.	
Initialization:	Oracle $\mathsf{LEval}(i, T, X)$:
$K_1, \ldots, K_u \xleftarrow{\$} \mathcal{K}$	$Z = \mathsf{E}_{K_i}(T, X)$, $\ell_{\mathsf{ev}} = \mathsf{L}_{\mathsf{Eval}}(K_i, T, X)$
$\mathcal{L} \leftarrow \emptyset$	$\mathcal{L} \leftarrow \mathcal{L} \cup \{(i, X, T, Z)\}$
	Return (Z, ℓ_{ev})
Finalization:	
$(i, X, T, Z) \leftarrow \mathcal{A}^{\mathsf{LEval},\mathsf{LInv},\mathsf{L}}$	Oracle $\mathsf{LInv}(i, T, Z)$:
If $(i, X, T, Z) \in \mathcal{L}$, Return 0	$X = \mathsf{E}^{-1}_{K_i}(T, Z)$, $\ell_{\mathsf{i}} = \mathsf{L}_{\mathsf{Inv}}(K_i, T, Z)$
If $Z == \mathsf{E}_{K_i}(T, X)$, Return 1	$\mathcal{L} \leftarrow \mathcal{L} \cup \{(i, X, T, Z)\}$
Return 0	Return (X, ℓ_{i})

We say that an implementation of E with leakage $\mathsf{L} = (\mathsf{L}_{\mathsf{Eval}}, \mathsf{L}_{\mathsf{Inv}})$ is a $(u, q_t, q_v, q_{\mathsf{L}}, t, \epsilon)$-strongly unpredictable tweakable block cipher if, for all adversaries $\mathcal{A}$ making at most q_t tag queries, q_v verification queries, q_{L} offline profiling queries on the implementation, and running in time less than t, the above advantage is upper bounded by ϵ.

8.2 New CIML2 Analysis of TETSponge/Spook

First, we recall the definition of the CIML2 advantage in the muti-user setting.

Definition 2 (muCIML2)**.** Given the implementation of an authenticated encryption $\mathsf{AE} = (\mathsf{Enc}, \mathsf{Dec})$ with leakage function pair $\mathsf{L} = (\mathsf{L}_{\mathsf{Enc}}, \mathsf{L}_{\mathsf{Dec}})$, the *multi-user ciphertext integrity advantage with misuse-resistance and leakage* of an adversary $\mathcal{A}$ against AE with u users is:

$$\mathbf{Adv}^{\mathsf{muCIML2}}_{\mathcal{A},\mathsf{AE},\mathsf{L},u} := \Pr\big[\mathsf{muCIML2}_{\mathcal{A},\mathsf{E},\mathsf{L},u} \Rightarrow 1\big],$$

where the security game muCIML2 is defined in Table 14.

Table 14: Ciphertext integrity with nonce misuse and leakage in enc. & dec. experiment.

$\mathsf{muCIML2}_{\mathcal{A},\mathsf{AE},\mathsf{L},u}$ experiment.	
Initialization: $K_1,\ldots,K_u \xleftarrow{\$} \mathcal{K},\;\; \mathcal{L} \leftarrow \emptyset$ Finalization: $(i,N,A,C) \leftarrow \mathcal{A}^{\mathsf{LEnc},\mathsf{LDec},\mathsf{L}}$ If $(i,N,A,C) \in \mathcal{L}$, Return 0 If $\perp \neq \mathsf{Dec}_{K_i}(N,A,C)$, Return 1 Return 0	Oracle $\mathsf{LEnc}(i,N,A,M)$: $C = \mathsf{Enc}_{K_i}(N,A,M)$, $\ell_{\mathsf{e}} = \mathsf{L}_{\mathsf{Enc}}(K_i,N,A,M)$ $\mathcal{L} \leftarrow \mathcal{L} \cup \{(i,N,A,C)\}$ Return (C,ℓ_{e}) Oracle $\mathsf{LDec}(i,N,A,C)$: $M = \mathsf{Dec}_{K_i}(N,A,C)$, $\ell_{\mathsf{d}} = \mathsf{L}_{\mathsf{Dec}}(K_i,N,A,C)$ Return (M,ℓ_{d})

Following Spook in Figure 1, we call S_0 the state of the first output of the permutation π, both in encryption and decryption. Then, iteratively, we get a chain of states $S = (S_0,\ldots,S_\lambda,\ldots,S_{\lambda+\ell})$ that are the consecutive outputs of π for a λ-block associated data A and ℓ-block message M (resp., c, for ciphertext $C = c\|Z$ with tag Z) in encryption (resp., decryption) as also detailed in the full specification of Appendix A. Then, the $2n-1$ most significant bits of $S_{\lambda+\ell}$, denoted $S_{\lambda+\ell}[:2n-1]$, give $U\|V$, with $|U| = n$. To simplify the notation, we write this process by $H(S_0,A,M) = U\|V$ in encryption and by $H^{-1}(S_0,A,c) = U\|V$ in decryption. We will prove muCIML2 security in a model where all the intermediate values computed by E and π are leaked in full – which we call the unbounded leakage model. This is to ensure integrity in a very robust and simple model. This model is also quite conservative in terms of security, as it considers that the information leaked comes at no cost for $\mathcal{A}$, while it is expected to require a non negligible amount of work in reality (for measurement and information extraction). The unbounded leakage function pair $\mathsf{L}^* = (\mathsf{L}^*_{\mathsf{Enc}}, \mathsf{L}^*_{\mathsf{Dec}})$ of Spook is defined as:

$\mathsf{L}^*_{\mathsf{Enc}}(K_i\|P_i,N,A,M)$: return $B = \mathsf{E}^{T_i}_{K_i}(N)$ and $\mathsf{L}_{\mathsf{Eval}}(K_i,T_i,N)$, where $T_i := P_i\|0$, as well as S to get $H(S_0,A,M) = U\|V$ and finally $\mathsf{L}_{\mathsf{Eval}}(K_i,V\|1,U)$;

$\mathsf{L}^*_{\mathsf{Dec}}(K_i\|P_i,N,A,C)$: return $B = \mathsf{E}^{T_i}_{K_i}(N)$ and $\mathsf{L}_{\mathsf{Eval}}(K_i,T_i,N)$, where $T_i := P_i\|0$, as well as S to get $H^{-1}(S_0,A,c) = U\|V$, $U^* = \mathsf{E}^{-1}_{K_i}(V\|1,Z)$ and $\mathsf{L}_{\mathsf{Inv}}(K_i,V\|1,Z)$.

Since we model π as a random oracle, we explicitly include the chain of state S in the leakage to capture the fact that π is a public function. Given B, we can in fact compute S from the known input of the query. We add S in order to avoid any confusion in our argument and to highlight that an adversary does not have to query π "offline" to get S. We never use the programmability of the random oracle as we modeled π as an ideal permutation where we only keep track of all the records of the forward (π) and backward (π^{-1}) evaluations. We recall that the ciphertext $C = c\|Z$ above is valid if $U = U^*$.

Theorem 1. *Let $\pi : \{0,1\}^{r+c} \longmapsto \{0,1\}^{r+c}$ be a random permutation with $r = c = 2n$, modeled as a random oracle, and $\mathsf{E} : \{0,1\}^n \times \{0,1\}^n \times \{0,1\}^n \longmapsto \{0,1\}^n$ be a $(u,q_t,q_v,q_{\mathsf{L}},t,\epsilon)$-strongly unpredictable tweakable block cipher with leakage $\mathsf{L} = (\mathsf{L}_{\mathsf{Eval}},\mathsf{L}_{\mathsf{Inv}})$. Then, in the unbounded model $\mathsf{L}^* = (\mathsf{L}^*_{\mathsf{Enc}},\mathsf{L}^*_{\mathsf{Dec}})$ defined above, for any adversary $\mathcal{A}$ making at most q_e leaking encryption queries, q_d leaking decryption queries, q_π offline forward or backward queries to π, q_{L} profiling queries to L^* on chosen keys, and running in time less than t, we have:*

$$\mathbf{Adv}^{\mathsf{muCIML2}}_{\mathcal{A},\mathsf{Spook},\mathsf{L}^*,u} \leq \frac{Q^2}{2^{2n-3}} + \frac{q_\pi}{2^n} + (q_d+1)\cdot\epsilon + \frac{q_d Q^2}{2^{n-1}}\cdot\epsilon + \frac{q_d Q}{2^{2n-1}},$$

assuming that $Q = \sigma + q_e + q_d + q_\pi + 1$, where σ is the total number of blocks (of r bits) in all the queried plaintexts and ciphertexts including associated data (as well as those of the *potential forgery ciphertext), that* $t_\pi(Q-q)+t_{\pi^{-1}}q+(2q_e+q_d+q_L-q')t_{\mathsf{E}}+(q_d+q')t_{\mathsf{E}^{-1}}+t' \le t$ *for any* $0 \le q \le q_\pi$ *and* $q' \le q_L$*, where* t' *is the time to manage the chain of states from the* π*-history, and where we assume that all the* π *evaluations involved in the* q_L *queries are already among the* q_π *queries, and as long as* $4 \le Q \le 2^{n-4}$ *and* $4q_d \le Q$*.*

The leading term of this security bound is $\frac{q_d Q^2}{2^{n-1}\epsilon^{-1}}$. If we assume the block size to be $n = 128$, the unpredictability bound to be $\epsilon = 2^{-96}$ and the number of online leaking decryption queries to be as high as 2^{-64}, we can still handle Q, which contains the offline computation factors, growing up to $\approx 2^{80}$. The bound, while remaining beyond birthday, is weaker than the one obtained when the TBC is modeled as leak-free (i.e., 2^{114}, as per Table 3). This is because we now enable the TBC to leak information. However, we note that we did not find any attack strategy matching our bounds, and it is likely that a more detailed analysis could lead to further improvements (see the discussion in [BGP+19]). We offer a proof sketch below, and defer the full proof to Appendix I.

Idea of the proof. Without loss of generality, we assume that all the inputs of any query can be parsed correctly and that all the fixed-length nonces are already padded with 0's, then $|N| = n$ in the following. Let (i, N, A, C) be a forgery ciphertext with $C = c\|Z$. Let $M = \mathsf{Dec}_{K_i\|P_i}(N, A, C)$. From $B = \mathsf{E}^{T_i}_{K_i}(N)$, we write $S_0 = \pi(T_i\|N\|0^n\|B)$ and $H^{-1}(S_0, A, c) = U\|V$ (which is also $H(S_0, A, M)$). If no quadruple of the form $(i, \star, V\|1, Z)$ appears during the computation of all the evaluations and inversions of E, $(i, U, V\|1, Z)$ is a valid fresh quadruple for E which breaks the unpredictability of the TBC. However, if it is not the case, a quadruple $(i, \star, V\|1, Z)$ appears either in the evaluation of E during an LEnc query or *only* in the inversion of E in an LDec query. (Note: since the last bit of the tweak is 1 we do not need to consider the first evaluation of E in those queries, where the last bit of the tweak is always 0). In the former case, as the answer to an LEnc query is necessarily valid, the quadruple $(i, \star, V\|1, Z)$ must actually be $(i, \mathsf{E}^{-1}_{K_i}(V\|1, Z), V\|1, Z)$, i.e. $(i, U, V\|1, Z)$. Of course, if the adversary has made an LEnc query on (i, N, A, M), it cannot win. If the adversary is successful, it means that it managed to request an LEnc query on some (i, N', A', M') such that $(N, A, M) \neq (N', A', M')$. Since $\mathsf{E}^{T_i}_{K_i}$ is a permutation it can only occur if $H(S_0, A, M) = H(S_0', A', M')$ while $(S_0, A, M) \neq (S_0', A', M')$, which implies a π-collision either on the last $c - 2 = 2n - 2$ bits of some states that are not the last in the chain or on the first $2n - 1$ bits of the final state of the chain. We can cover this case by removing once and for all any π collisions of both kinds. This results in the first term of the bound. We can thus focus on the latter case where a quadruple $(i, \star, V\|1, Z)$ *only* appears when answering an LDec query, i.e. in an inversion of E. Of course, if the first time $(V\|1, Z)$ appears when answering an LDec query for the user i the ciphertext is valid, we can reduce it to the unpredictability of E again. After all, the adversary might have used its forgery in an LDec query before deciding to end the $\mathsf{muCIML2}$ experiment. So, we already have the term $(q_d + 1)\epsilon$ of the bound. Now, we are left with the case where the first time $(V\|1, Z)$ appears in an LDec query for user i, the processed ciphertext is invalid. Moreover, we recall that $(V\|1, Z)$ never appears in an LEnc query for user i.

In the forgery ciphertext, we call $R_{\lambda+\ell}$ the last input of π in decryption. Therefore, $\pi(R_{\lambda+\ell}) = S_{\lambda+\ell}$ and then $\pi(R_{\lambda+\ell})[: 2n-1] = U\|V$. We now argue that the first time the input-output couple $(R_{\lambda+\ell}, S_{\lambda+\ell})$ is defined for π it was in a forward query $\pi(R_{\lambda+\ell})$. Otherwise, either the full chain of states has been computed in backward or there is a collision somewhere in the middle. In the former case, it easy to see that being able to cast the chain $(S_0, \ldots, S_\lambda, \ldots, S_{\lambda+\ell})$ into a valid ciphertert requires that $\pi^{-1}(S_0) = T_i\|\star\|0^n\|\star$, which at least implies computing a partial preimage of 0^n at the third n-bit position. In the latter case, we must have $\pi(S_{\alpha-1})[r+2:] = \pi^{-1}(S_{\alpha+1})[r+2:]$ in their first computation, i.e. a collision on the last $c-2$ bits of their outputs. But, we already dealt with such collisions as the total of the q_π queries included in Q counts for both forward and backward queries to π. So, up to the probability of $q_\pi/2^n$, only the forward evaluation matters.

We split the remaining winning conditions into: (1) $R_{\lambda+\ell}$ appears as an input of π before the first apparition of $(V\|1, Z)$ in a leaking decryption query for user i; (2) $R_{\lambda+\ell}$ appears as an input of π strictly after the first apparition of $(V\|1, Z)$ in a leaking decryption query for user i; no matter whether Z appears first in an LEnc answer or in an LDec query and no matter where $\pi(R_{\lambda+\ell})$ is computed for the first time. For instance, $\pi(R_{\lambda+\ell})$ can appear in the chain of states in the computation of the answer to some LEnc query and the adversary is trying to compute a fresh and valid ciphertext from a prefix of that chain or in an offline π-query as an attempt to extend a chain of states. The first case means the adversary chooses Z depending on the view of the output value $U\|V$ and hence it relates to the unpredictability of E. In the second case, the target $U^*\|V$ is implicitly fixed in the first leaking decryption query where $(V\|1, Z)$ is going to appear while the output of $\pi(R_{\lambda+\ell})$ remains uniformly random and independent of the view at that time.

We finally go on with the first leaking decryption query for user i where $(V\|1, Z)$ appears (and we already know that it is invalid). By convention, we consider the forgery as the $(q_d + 1)$-th LDec query, leading to the following two cases:

In case 1, $\pi(R_{\lambda+\ell})[: 2n - 1] = U\|V$ appears before the first time a leaking decryption query for user i involves $(V\|1, Z)$. Since the corresponding ciphertext is invalid we have to emulate E by calling the LEval and LInv oracles... except if we already won against the unpredictability. If we could not win at that time and if we had to simulate the LDec query properly, we would not win with the final forgery ciphertext later as the quadruple $(i, U^*, V\|1, Z)$ would have been "consumed" in the emulation and would no longer be fresh afterwards. Fortunately, in this leaking decryption query we can start to emulate the first call to E by an LEval query to get the leaking TBC output that allows computing the chain of states of the given ciphertext until $U'\|V$ with, necessarily, $U' \neq U$. At that point, instead of emulating a leaking inversion of E with $(i, V\|1, Z)$, we can guess which output state already defined from a forward query to π among those with the $2n - 1$ first bits of the form $\star\|V$ is actually the right $U\|V$. And we know that $U\|V$ is already in the π-history. Therefore, for each such output state we have to make a reduction to $\mathsf{muSUL2}$. Fortunately again, as the number of such output states implies as many multi-collisions on the V values this number remains sufficiently small. Taking into account all the possible output states with such a property and all the leaking decryption queries (as we cannot be sure which one will correspond to our $U\|V$ of the forgery ciphertext before the end of the $\mathsf{muCIML2}$ experiment) we get the term $\epsilon \cdot q_d q^2/2^{n-1}$ of the security bound.

In case 2, the adversary outputs a forgery ciphertext while $(V\|1, Z)$ appears in a leaking decryption query before the first computation of $\pi(R_{\lambda+\ell})$. Here, we simply pick the key of the TBC to simulate the $\mathsf{muCIML2}$ experiment. If $(V\|1, Z)$ already appears when answering an LDec query for user i the TBC quadruple $(i, U^*, V\|1, Z)$ is already fixed in the answer while the current ciphertext is invalid. Therefore $\pi(R_{\lambda+\ell})$ which is still uniformly random and independent of the view at that time will have to match the target $U^*\|V$. This match thus happens with probability 2^{1-2n} for each future π evaluation in a direct offline π-query or inside a next LEnc or LDec query. Of course we do not know in advance what will be the right $(V\|1, Z)$ and the right user i until the adversary output its forgery ciphertext in the finalization phase. So, if $(i_j, V_j\|1, Z_j)$ denotes the input of the leaking inversion of E in the j-th leaking decryption query, we actually defines q_d targets $(U_j^*, V^j\|1, Z_j)$, since necessarily $i < q_d + 1$ here. Therefore, the probability that this case occurs is upper-bounded by $q_d Q/2^{2n-1}$, which completes the proof.

9 Conclusion

This paper discusses the Spook AEAD, which is designed to support low-energy implementations that are side-channel resistant, and is based in a permutation and a TBC.

By reporting new implementation results on a variety of platforms, we first demonstrate that the overheads resulting from the use of two primitives is actually marginal, thanks to the adoption of a TBC and a permutation that share most of their components. Our implementation results are in the most unfavorable conditions for Spook: we consider a setting without any specific protection against side-channel attacks. We conjecture that the benefits of the Spook design would become even more visible when countermeasures against side-channel attacks need to be implemented, since the leveled design of Spook would then come into play. As SCA protection strategies can differ between schemes, it is an important open question to establish how an informative comparison could be made between various designs, while targeting a common level of security.

As a second new contribution, this paper also offers an analysis of the integrity properties of the Spook design, based on the assumption that its underlying TBC remains unpredictable despite its leakage on past queries (and that the permutation leaks its state in full). This contrasts with the previously proposed analysis that assumed the TBC to be leak-free. The resulting security bounds remain quite satisfactory, and beyond-birthday in particular. It is another open problem to explore whether similar results could be obtained for confidentiality properties, and whether the security bounds could still be improved (in particular, by making a finer-grained analysis of the multi-user setting).

Acknowledgments. The authors are grateful to Patrick Derbez, Paul Huynh, Virginie Lallemand, Léo Perrin, Maria Naya Plasencia and Andre Schrottenloher for sharing their analysis of Shadow and Spook and discussing tweaks. We specially thank Maria Naya Plasencia for numerous interactions. Gaëtan Cassiers, Thomas Peters and François-Xavier Standaert are respectively PhD Student, Post-Doctoral Researcher and Senior Research Associate of the Belgian Fund for Scientific Research (FNRS-F.R.S.). This work has been funded in part by European Union and the Walloon Region through the ERC Project 724725 (acronym SWORD), the FEDER Project USERMedia (convention 501907-379156), the H2020 project REASSURE and the Wallinov TRUSTEYE project.

References

[ADL17] Tomer Ashur, Orr Dunkelman, and Atul Luykx. Boosting Authenticated Encryption Robustness with Minimal Modifications. In Jonathan Katz and Hovav Shacham, editors, *Advances in Cryptology - CRYPTO 2017 - 37th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 20-24, 2017, Proceedings, Part III*, volume 10403 of *LNCS*, pages 3–33. Springer, 2017.

[BBC+20] Davide Bellizia, Olivier Bronchain, Gaëtan Cassiers, Vincent Grosso, Chun Guo, Charles Momin, Olivier Pereira, Thomas Peters, and François-Xavier Standaert. Mode-Level vs. Implementation-Level Physical Security in Symmetric Cryptography: A Practical Guide Through the Leakage-Resistance Jungle. Cryptology ePrint Archive, Report 2020/211, 2020. https://eprint.iacr.org/2020/211.

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

[BCC11] Christina Boura, Anne Canteaut, and Christophe De Cannière. Higher-Order Differential Properties of Keccak and *Luffa*. In Antoine Joux, editor, *Fast Software Encryption - 18th International Workshop, FSE 2011, Lyngby, Denmark, February 13-16, 2011, Revised Selected Papers*, volume 6733 of *LNCS*, pages 252–269. Springer, 2011.

[BCLR17] Christof Beierle, Anne Canteaut, Gregor Leander, and Yann Rotella. Proving Resistance Against Invariant Attacks: How to Choose the Round Constants. In Jonathan Katz and Hovav Shacham, editors, *Advances in Cryptology - CRYPTO 2017 - 37th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 20-24, 2017, Proceedings, Part II*, volume 10402 of *LNCS*, pages 647–678. Springer, 2017.

[BDLF10] Charles Bouillaguet, Orr Dunkelman, Gaëtan Leurent, and Pierre-Alain Fouque. Another Look at Complementation Properties. In Seokhie Hong and Tetsu Iwata, editors, *Fast Software Encryption, 17th International Workshop, FSE 2010, Seoul, Korea, February 7-10, 2010, Revised Selected Papers*, volume 6147 of *Lecture Notes in Computer Science*, pages 347–364. Springer, 2010.

[BDPA11] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Duplexing the Sponge: Single-Pass Authenticated Encryption and Other Applications. In Ali Miri and Serge Vaudenay, editors, *Selected Areas in Cryptography - 18th International Workshop, SAC 2011, Toronto, ON, Canada, August 11-12, 2011, Revised Selected Papers*, volume 7118 of *LNCS*, pages 320–337. Springer, 2011.

[Beh19] Behnaz Rezvani and William Diehl. Hardware implementations of NIST lightweight cryptographic candidates: A first look. Cryptology ePrint Archive, Report 2019/824, 2019. https://eprint.iacr.org/2019/824.

[BGP+19] Francesco Berti, Chun Guo, Olivier Pereira, Thomas Peters, and François-Xavier Standaert. Strong Authenticity with Leakage under Weak and Falsifiable Physical Assumptions. Cryptology ePrint Archive, Report 2019/1413, 2019. https://eprint.iacr.org/2019/1413 – Extended abstract to appear at Inscrypt 2019.

[BGP+20] Francesco Berti, Chun Guo, Olivier Pereira, Thomas Peters, and François-Xavier Standaert. TEDT, a Leakage-Resist AEAD Mode for High Physical Security Applications. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2020(1):256–320, 2020.

[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, *Advances in Cryptology - CRYPTO 2016 - 36th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2016, Proceedings, Part II*, volume 9815 of *LNCS*, pages 123–153. Springer, 2016.

[BKP+18] Francesco Berti, François Koeune, Olivier Pereira, Thomas Peters, and François-Xavier Standaert. Ciphertext Integrity with Misuse and Leakage: Definition and Efficient Constructions with Symmetric Primitives. In Jong Kim, Gail-Joon Ahn, Seungjoo Kim, Yongdae Kim, Javier López, and Taesoo Kim, editors, *Proceedings of the 2018 on Asia Conference on Computer and Communications Security, AsiaCCS 2018, Incheon, Republic of Korea, June 04-08, 2018*, pages 37–50. ACM, 2018.

[BMOS17] Guy Barwell, Daniel P. Martin, Elisabeth Oswald, and Martijn Stam. Authenticated Encryption in the Face of Protocol and Side Channel Leakage. In Tsuyoshi Takagi and Thomas Peyrin, editors, *Advances in Cryptology - ASIACRYPT 2017 - 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings, Part I*, volume 10624 of *Lecture Notes in Computer Science*, pages 693–723. Springer, 2017.

[BPPS17] Francesco Berti, Olivier Pereira, Thomas Peters, and François-Xavier Standaert. On Leakage-Resilient Authenticated Encryption with Decryption Leakages. *IACR Trans. Symmetric Cryptol.*, 2017(3):271–293, 2017.

[CGLS20] Gaëtan Cassiers, Benjamin Grégoire, Itamar Levi, and François-Xavier Standaert. Hardware Private Circuits: From Trivial Composition to Full Verification. Cryptology ePrint Archive, Report 2020/185, 2020. https://eprint.iacr.org/2020/185.

[CGP+19] Gaëtan Cassiers, Chun Guo, Olivier Pereira, Thomas Peters, and François-Xavier Standaert. SpookChain: Chaining a Sponge-Based AEAD with Beyond-Birthday Security. In Shivam Bhasin, Avi Mendelson, and Mridul Nandi, editors, *Security, Privacy, and Applied Cryptography Engineering - 9th International Conference, SPACE 2019, Gandhinagar, India, December 3-7, 2019, Proceedings*, volume 11947 of *Lecture Notes in Computer Science*, pages 67–85. Springer, 2019.

[CP02] Nicolas Courtois and Josef Pieprzyk. Cryptanalysis of Block Ciphers with Overdefined Systems of Equations. In Yuliang Zheng, editor, *Advances in Cryptology - ASIACRYPT 2002, 8th International Conference on the Theory and Application of Cryptology and Information Security, Queenstown, New Zealand, December 1-5, 2002, Proceedings*, volume 2501 of *LNCS*, pages 267–287. Springer, 2002.

[DEM+17] Christoph Dobraunig, Maria Eichlseder, Stefan Mangard, Florian Mendel, and Thomas Unterluggauer. ISAP - Towards Side-Channel Secure Authenticated Encryption. *IACR Trans. Symmetric Cryptol.*, 2017(1):80–105, 2017.

[DHL+20] Patrick Derbez, Paul Huynh, Virginie Lallemand, María Naya-Plasencia, Léo Perrin, and André Schrottenloher. Cryptanalysis Results on Spook. Cryptology ePrint Archive, Report 2020/309, 2020. https://eprint.iacr.org/2020/309.

[DL18] Sébastien Duval and Gaëtan Leurent. MDS Matrices with Lightweight Circuits. *IACR Trans. Symmetric Cryptol.*, 2018(2):48–78, 2018.

[DMA17] Joan Daemen, Bart Mennink, and Gilles Van Assche. Full-State Keyed Duplex with Built-In Multi-user Support. In Tsuyoshi Takagi and Thomas Peyrin, editors, *Advances in Cryptology - ASIACRYPT 2017 - 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings, Part II*, volume 10625 of *LNCS*, pages 606–637. Springer, 2017.

[DR01] Joan Daemen and Vincent Rijmen. The Wide Trail Design Strategy. In Bahram Honary, editor, *Cryptography and Coding, 8th IMA International Conference, Cirencester, UK, December 17-19, 2001, Proceedings*, volume 2260 of *LNCS*, pages 222–238. Springer, 2001.

[DS09a] Itai Dinur and Adi Shamir. Cube Attacks on Tweakable Black Box Polynomials. In Antoine Joux, editor, *Advances in Cryptology - EUROCRYPT 2009, 28th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Cologne, Germany, April 26-30, 2009. Proceedings*, volume 5479 of *LNCS*, pages 278–299. Springer, 2009.

[DS09b] Yevgeniy Dodis and John P. Steinberger. Message Authentication Codes from Unpredictable Block Ciphers. In Shai Halevi, editor, *Advances in Cryptology - CRYPTO 2009, 29th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2009. Proceedings*, volume 5677 of *Lecture Notes in Computer Science*, pages 267–285. Springer, 2009.

[FGP+18] Sebastian Faust, Vincent Grosso, Santos Merino Del Pozo, Clara Paglialonga, and François-Xavier Standaert. Composable Masking Schemes in the Presence of Physical Defaults & the Robust Probing Model. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2018(3):89–120, 2018.

[GLS+14] Vincent Grosso, Gaëtan Leurent, François-Xavier Standaert, Kerem Varici, rançois Durvaux, Lubos Gaspar, and Stéphanie Kerckhof. SCREAM & iSCREAM Side-Channel Resistant Authenticated Encryption with Masking, Submission to the CAESAR competition, 2014.

[GLSV14] Vincent Grosso, Gaëtan Leurent, François-Xavier Standaert, and Kerem Varici. LS-Designs: Bitslice Encryption for Efficient Masked Software Implementations. In Carlos Cid and Christian Rechberger, editors, *Fast Software Encryption - 21st International Workshop, FSE 2014, London, UK, March 3-5, 2014. Revised Selected Papers*, volume 8540 of *LNCS*, pages 18–37. Springer, 2014.

[GPPR11] Jian Guo, Thomas Peyrin, Axel Poschmann, and Matthew J. B. Robshaw. The LED Block Cipher. In Bart Preneel and Tsuyoshi Takagi, editors, *Cryptographic Hardware and Embedded Systems - CHES 2011 - 13th International Workshop, Nara, Japan, September 28 - October 1, 2011. Proceedings*, volume 6917 of *LNCS*, pages 326–341. Springer, 2011.

[GPPS19a] Chun Guo, Olivier Pereira, Thomas Peters, and François-Xavier Standaert. Authenticated Encryption with Nonce Misuse and Physical Leakage: Definitions, Separation Results and First Construction - (Extended Abstract). In Peter Schwabe and Nicolas Thériault, editors, *Progress in Cryptology - LATINCRYPT 2019 - 6th International Conference on Cryptology and Information Security in Latin America, Santiago de Chile, Chile, October 2-4, 2019, Proceedings*, volume 11774 of *Lecture Notes in Computer Science*, pages 150–172. Springer, 2019.

[GPPS19b] Chun Guo, Olivier Pereira, Thomas Peters, and François-Xavier Standaert. Towards Low-Energy Leakage-Resistant Authenticated Encryption from the Duplex Sponge Construction. Cryptology ePrint Archive, Report 2019/193, 2019. https://eprint.iacr.org/2019/193.

[GR17] Dahmun Goudarzi and Matthieu Rivain. How Fast Can Higher-Order Masking Be in Software? In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *Advances in Cryptology - EUROCRYPT 2017 - 36th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Paris, France, April 30 - May 4, 2017, Proceedings, Part I*, volume 10210 of *LNCS*, pages 567–597, 2017.

[GRR16] Lorenzo Grassi, Christian Rechberger, and Sondre Rønjom. Subspace Trail Cryptanalysis and its Applications to AES. *IACR Trans. Symmetric Cryptol.*, 2016(2):192–225, 2016.

[HRRV15] Viet Tung Hoang, Reza Reyhanitabar, Phillip Rogaway, and Damian Vizár. Online Authenticated-Encryption and its Nonce-Reuse Misuse-Resistance. In Rosario Gennaro and Matthew Robshaw, editors, *Advances in Cryptology - CRYPTO 2015 - 35th Annual Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2015, Proceedings, Part I*, volume 9215 of *LNCS*, pages 493–517. Springer, 2015.

[JNP14] Jérémy Jean, Ivica Nikolic, and Thomas Peyrin. Tweaks and Keys for Block Ciphers: The TWEAKEY Framework. In Palash Sarkar and Tetsu Iwata, editors, *Advances in Cryptology - ASIACRYPT 2014 - 20th International Conference on the Theory and Application of Cryptology and Information Security, Kaoshiung, Taiwan, R.O.C., December 7-11, 2014, Proceedings, Part II*, volume 8874 of *LNCS*, pages 274–288. Springer, 2014.

[KDH+12] Stéphanie Kerckhof, François Durvaux, Cédric Hocquet, David Bol, and François-Xavier Standaert. Towards Green Cryptography: A Comparison of Lightweight Ciphers from the Energy Viewpoint. In Emmanuel Prouff and Patrick Schaumont, editors, *Cryptographic Hardware and Embedded Systems - CHES 2012 - 14th International Workshop, Leuven, Belgium, September 9-12, 2012. Proceedings*, volume 7428 of *LNCS*, pages 390–407. Springer, 2012.

[KN10] Dmitry Khovratovich and Ivica Nikolic. Rotational Cryptanalysis of ARX. In Seokhie Hong and Tetsu Iwata, editors, *Fast Software Encryption, 17th International Workshop, FSE 2010, Seoul, Korea, February 7-10, 2010, Revised Selected Papers*, volume 6147 of *Lecture Notes in Computer Science*, pages 333–346. Springer, 2010.

[LAAZ11] Gregor Leander, Mohamed Ahmed Abdelraheem, Hoda AlKhzaimi, and Erik Zenner. A Cryptanalysis of PRINTcipher: The Invariant Subspace Attack. In Phillip Rogaway, editor, *Advances in Cryptology - CRYPTO 2011 - 31st Annual Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2011. Proceedings*, volume 6841 of *Lecture Notes in Computer Science*, pages 206–221. Springer, 2011.

[LMR15] Gregor Leander, Brice Minaud, and Sondre Rønjom. A Generic Approach to Invariant Subspace Attacks: Cryptanalysis of Robin, iSCREAM and Zorro. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part I*, volume 9056 of *LNCS*, pages 254–283. Springer, 2015.

[LTW18] Gregor Leander, Cihangir Tezcan, and Friedrich Wiemer. Searching for Subspace Trails and Truncated Differentials. *IACR Trans. Symmetric Cryptol.*, 2018(1):74–100, 2018.

[MMGD17] Elodie Morin, Mickael Maman, Roberto Guizzetti, and Andrzej Duda. Comparison of the Device Lifetime in Wireless Networks for the Internet of Things. *IEEE Access*, 5:7097–7114, 2017.

[MRST09] Florian Mendel, Christian Rechberger, Martin Schläffer, and Søren S. Thomsen. The Rebound Attack: Cryptanalysis of Reduced Whirlpool and Grøstl. In Orr

Dunkelman, editor, *Fast Software Encryption, 16th International Workshop, FSE 2009, Leuven, Belgium, February 22-25, 2009, Revised Selected Papers*, volume 5665 of *Lecture Notes in Computer Science*, pages 260–276. Springer, 2009.

[MSGR10] Marcel Medwed, François-Xavier Standaert, Johann Großschädl, and Francesco Regazzoni. Fresh Re-keying: Security against Side-Channel and Fault Attacks for Low-Cost Devices. In Daniel J. Bernstein and Tanja Lange, editors, *Progress in Cryptology - AFRICACRYPT 2010, Third International Conference on Cryptology in Africa, Stellenbosch, South Africa, May 3-6, 2010. Proceedings*, volume 6055 of *LNCS*, pages 279–296. Springer, 2010.

[NRS11] Svetla Nikova, Vincent Rijmen, and Martin Schläffer. Secure Hardware Implementation of Nonlinear Functions in the Presence of Glitches. *J. Cryptology*, 24(2):292–321, 2011.

[Pey10] Thomas Peyrin. Improved Differential Attacks for ECHO and Grøstl. In Tal Rabin, editor, *Advances in Cryptology - CRYPTO 2010, 30th Annual Cryptology Conference, Santa Barbara, CA, USA, August 15-19, 2010. Proceedings*, volume 6223 of *Lecture Notes in Computer Science*, pages 370–392. Springer, 2010.

[PSV15] Olivier Pereira, François-Xavier Standaert, and Srinivas Vivek. Leakage-Resilient Authentication and Encryption from Symmetric Cryptographic Primitives. In Indrajit Ray, Ninghui Li, and Christopher Kruegel, editors, *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-16, 2015*, pages 96–108. ACM, 2015.

[RS06] Phillip Rogaway and Thomas Shrimpton. A Provable-Security Treatment of the Key-Wrap Problem. In Serge Vaudenay, editor, *Advances in Cryptology - EUROCRYPT 2006, 25th Annual International Conference on the Theory and Applications of Cryptographic Techniques, St. Petersburg, Russia, May 28 - June 1, 2006, Proceedings*, volume 4004 of *LNCS*, pages 373–390. Springer, 2006.

[RSWO18] Eyal Ronen, Adi Shamir, Achi-Or Weingarten, and Colin O'Flynn. IoT Goes Nuclear: Creating a Zigbee Chain Reaction. *IEEE Security & Privacy*, 16(1):54–62, 2018.

[SPY13] François-Xavier Standaert, Olivier Pereira, and Yu Yu. Leakage-Resilient Symmetric Cryptography under Empirically Verifiable Assumptions. In Ran Canetti and Juan A. Garay, editors, *Advances in Cryptology - CRYPTO 2013 - 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part I*, volume 8042 of *Lecture Notes in Computer Science*, pages 335–352. Springer, 2013.

[Sta19] François-Xavier Standaert. Towards and Open Approach to Secure Cryptographic Implementations (Invited Talk). In Yuval Ishai and Vincent Rijmen, editors, *Advances in Cryptology - EUROCRYPT 2019 - 38th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Darmstadt, Germany, May 19-23, 2019, Proceedings, Part I*, volume 11476 of *Lecture Notes in Computer Science*, pages xv, https://www.youtube.com/watch?v=KdhrsuJT1sE. Springer, 2019.

[TLS16] Yosuke Todo, Gregor Leander, and Yu Sasaki. Nonlinear Invariant Attack - Practical Attack on Full SCREAM, iSCREAM, and Midori64. In Jung Hee Cheon and Tsuyoshi Takagi, editors, *Advances in Cryptology - ASIACRYPT 2016 - 22nd International Conference on the Theory and Application of Cryptology and Information Security, Hanoi, Vietnam, December 4-8, 2016, Proceedings, Part II*, volume 10032 of *LNCS*, pages 3–33, 2016.

[Tod15] Yosuke Todo. Structural Evaluation by Generalized Integral Property. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part I*, volume 9056 of *LNCS*, pages 287–314. Springer, 2015.

[TOS10] Eran Tromer, Dag Arne Osvik, and Adi Shamir. Efficient Cache Attacks on AES, and Countermeasures. *J. Cryptology*, 23(1):37–71, 2010.

[VMKS12] Nicolas Veyrat-Charvillon, Marcel Medwed, Stéphanie Kerckhof, and François-Xavier Standaert. Shuffling against Side-Channel Attacks: A Comprehensive Study with Cautionary Note. In Xiaoyun Wang and Kazue Sako, editors, *Advances in Cryptology - ASIACRYPT 2012 - 18th International Conference on the Theory and Application of Cryptology and Information Security, Beijing, China, December 2-6, 2012. Proceedings*, volume 7658 of *LNCS*, pages 740–757. Springer, 2012.

[XZBL16] Zejun Xiang, Wentao Zhang, Zhenzhen Bao, and Dongdai Lin. Applying MILP Method to Searching Integral Distinguishers Based on Division Property for 6 Lightweight Block Ciphers. In Jung Hee Cheon and Tsuyoshi Takagi, editors, *Advances in Cryptology - ASIACRYPT 2016 - 22nd International Conference on the Theory and Application of Cryptology and Information Security, Hanoi, Vietnam, December 4-8, 2016, Proceedings, Part I*, volume 10031 of *Lecture Notes in Computer Science*, pages 648–678, 2016.

A TETSponge mode of operation: specifications

Notation. For a bitstring $S = b_0 \dots b_{m-1}$, we denote the bitstring of first bits $b_0 \dots b_{x-1}$ as $S[: x]$ and we denote the bitstring of last bits $b_x \dots b_{m-1}$ as $S[x :]$.

Algorithm 3 $\mathsf{TETSponge}[\mathsf{E}, \pi].\mathsf{Enc}(\boldsymbol{A}, \boldsymbol{M}, N, K||P)$.

1. $\ell \leftarrow \lceil |\boldsymbol{M}|/r \rceil$, $\lambda \leftarrow \lceil |\boldsymbol{A}|/r \rceil$;
2. Parse $\boldsymbol{M}$ as $M[0]\|\dots\|M[\ell-1]$, with $|M[0]| = \dots = |M[\ell-2]| = r$ and $1 \leq |M[\ell-1]| \leq r$;
3. Parse $\boldsymbol{A}$ as $A[0]\|\dots\|A[\lambda-1]$, with $|A[0]| = \dots = |A[\lambda-2]| = r$ and $1 \leq |A[\lambda-1]| \leq r$;
4. $B \leftarrow \mathsf{E}_K^{P\|0}(N\|0^*)$;
5. $IV \leftarrow P\|0\|N\|0^*$ (with size $r + c - n$);
6. $S_0 \leftarrow \pi(IV\|B)$;
7. **if** $\lambda \geq 1$ **then**
   (a) **for** $i = 0$ **to** $\lambda - 2$ **do**
   - $S_i \leftarrow S_i \oplus (A[i]\|0^c)$;
   - $S_{i+1} \leftarrow \pi(S_i)$;

   (b) **if** $|A[\lambda-1]| < r$ **then**
   - $A[\lambda-1] \leftarrow A[\lambda-1]\|10^{r-|A[\lambda-1]|-1}$;
   - $S_{\lambda-1} \leftarrow S_{\lambda-1} \oplus (0^r\|01\|0^{c-2})$;

   (c) $S_{\lambda-1} \leftarrow S_{\lambda-1} \oplus (A[\lambda-1]\|0^c)$;
   (d) $S_\lambda \leftarrow \pi(S_{\lambda-1})$;
8. **if** $\ell \geq 1$ **then**
   (a) $S_\lambda \leftarrow S_\lambda \oplus (0^r\|10\|0^{c-2})$;
   (b) **for** $i = 0$ **to** $\ell - 2$ **do**
   - $j \leftarrow i + \lambda$;
   - $C[i] \leftarrow S_j[: r] \oplus M[i]$;
   - $S_j \leftarrow C[i]\|S_j[r :]$;
   - $S_{j+1} \leftarrow \pi(S_j)$;

   (c) $C[\ell-1] \leftarrow S_{\lambda+\ell-1}[: |M[\ell-1]|] \oplus M[\ell-1]$;
   (d) **if** $|C[\ell-1]| < r$ **then**
   - $S_{\lambda+\ell-1} \leftarrow S_{\lambda+\ell-1} \oplus (0^{|C[\ell-1]|}\|10^{r-|C[\ell-1]|-1}\|01\|0^{c-2})$;
   - $S_{\lambda+\ell-1} \leftarrow C[\ell-1]\|S_{\lambda+\ell-1}[|C[l-1]| :]$;

   (e) **else** $S_{\lambda+\ell-1} \leftarrow C[\ell-1]\|S_{\lambda+\ell-1}[r :]$;
   (f) $S_{\lambda+\ell} \leftarrow \pi(S_{\lambda+\ell-1})$;
9. $U||V \leftarrow S_{\lambda+\ell}[: 2n-1]$;
10. $Z \leftarrow \mathsf{E}_K^{V||1}(U)$;
11. $\boldsymbol{c} \leftarrow C[0]\|\dots\|C[\ell-1]$, $\boldsymbol{C} \leftarrow \boldsymbol{c}||Z$;
12. **return** $\boldsymbol{C}$;

Algorithm 4 $\mathsf{TETSponge}[\mathsf{E},\pi].\mathsf{Dec}(\boldsymbol{A},\boldsymbol{C},N,K||P)$.

1. $\ell \leftarrow \lceil \frac{|C|-n}{r} \rceil$, $\lambda \leftarrow \lceil |A|/r \rceil$;
2. Parse $\boldsymbol{C}$ as $C[0]\| \ldots \|C[\ell-1]||Z$, with $|C[0]| = \ldots = |C[\ell-2]| = r$, $1 \leq |C[\ell-1]| \leq r$ and $|Z| = n$;
3. Parse $\boldsymbol{A}$ as $A[0]\| \ldots \|A[\lambda-1]$, with $|A[0]| = \ldots = |A[\lambda-2]| = r$ and $1 \leq |A[\lambda-1]| \leq r$;
4. $B \leftarrow \mathsf{E}_K^{P\|0}(N\|0^*)$;
5. $IV \leftarrow P\|0\|N\|0^*$ (with size $r + c - n$);
6. $S_0 \leftarrow \pi(IV\|B)$;
7. **if** $\lambda \geq 1$ **then**
   (a) **for** $i = 0$ **to** $\lambda - 2$ **do**
      - $S_i \leftarrow S_i \oplus (A[i]\|0^c)$;
      - $S_{i+1} \leftarrow \pi(S_i)$;

   (b) **if** $|A[\lambda-1]| < r$ **then**
      - $A[\lambda] \leftarrow A[\lambda-1]\|10^{r-|A[\lambda-1]|-1}$;
      - $S_{\lambda-1} \leftarrow S_{\lambda-1} \oplus (0^r\|01\|0^{c-2})$;

   (c) $S_{\lambda-1} \leftarrow S_{\lambda-1} \oplus (A[\lambda-1]\|0^c)$;
   (d) $S_\lambda \leftarrow \pi(S_{\lambda-1})$;
8. **if** $\ell \geq 1$ **then**
   (a) $S_\lambda \leftarrow S_\lambda \oplus (0^r\|10\|0^{c-2})$;
   (b) **for** $i = 0$ **to** $\ell - 2$ **do**
      - $j \leftarrow i + \lambda$;
      - $M[i] \leftarrow S_j[: r] \oplus C[i]$;
      - $S_j \leftarrow C[i]\|S_j[r :]$;
      - $S_{j+1} \leftarrow \pi(S_j)$;

   (c) $M[\ell-1] \leftarrow S_{\lambda+\ell-1}[: |C[\ell-1]|] \oplus C[\ell-1]$;
   (d) **if** $|C[\ell-1]| < r$ **then**
      - $S_{\lambda+\ell-1} \leftarrow S_{\lambda+\ell-1} \oplus (0^{|C[\ell-1]|}\|10^{r-|C[\ell-1]|-1}\|01\|0^{c-2})$;
      - $S_{\lambda+\ell-1} \leftarrow C[\ell-1]\|S_{\lambda+\ell-1}[|C[l-1]| :]$;

   (e) **else** $S_{\lambda+\ell-1} \leftarrow C[\ell-1]\|S_{\lambda+\ell-1}[r :]$;
   (f) $S_{\lambda+\ell} \leftarrow \pi(S_{\lambda+\ell-1})$;
9. $U||V \leftarrow S_{\lambda+\ell}[: 2n-1]$;
10. $U^* \leftarrow \left(\mathsf{E}_K^{V||1}\right)^{-1}(Z)$;
11. **if** $U \neq U^*$ **then return** $\perp$;
12. **else if** $\ell > 0$ **then return** $M[0]\| \ldots \|M[\ell-1]$;
13. **else return true**;

B Cases of the TETSponge mode of operation

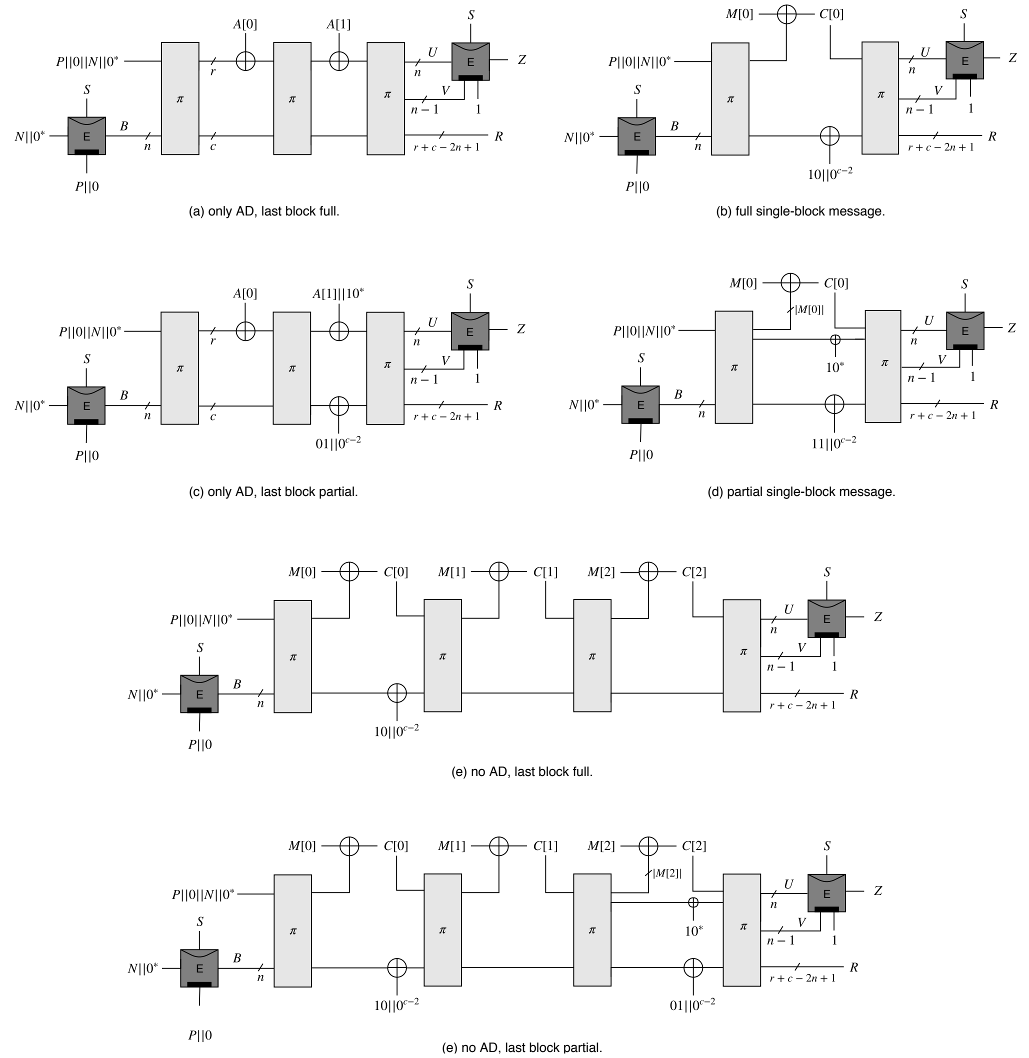

Figure 3: Different cases of the TETSponge mode of operation.

C Clyde-128 and Shadow-512 illustrations

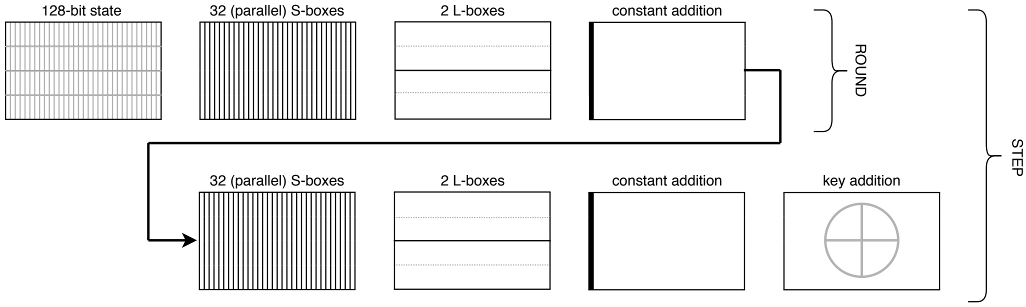

Figure 4: Round and step of Clyde-128: high-level view.

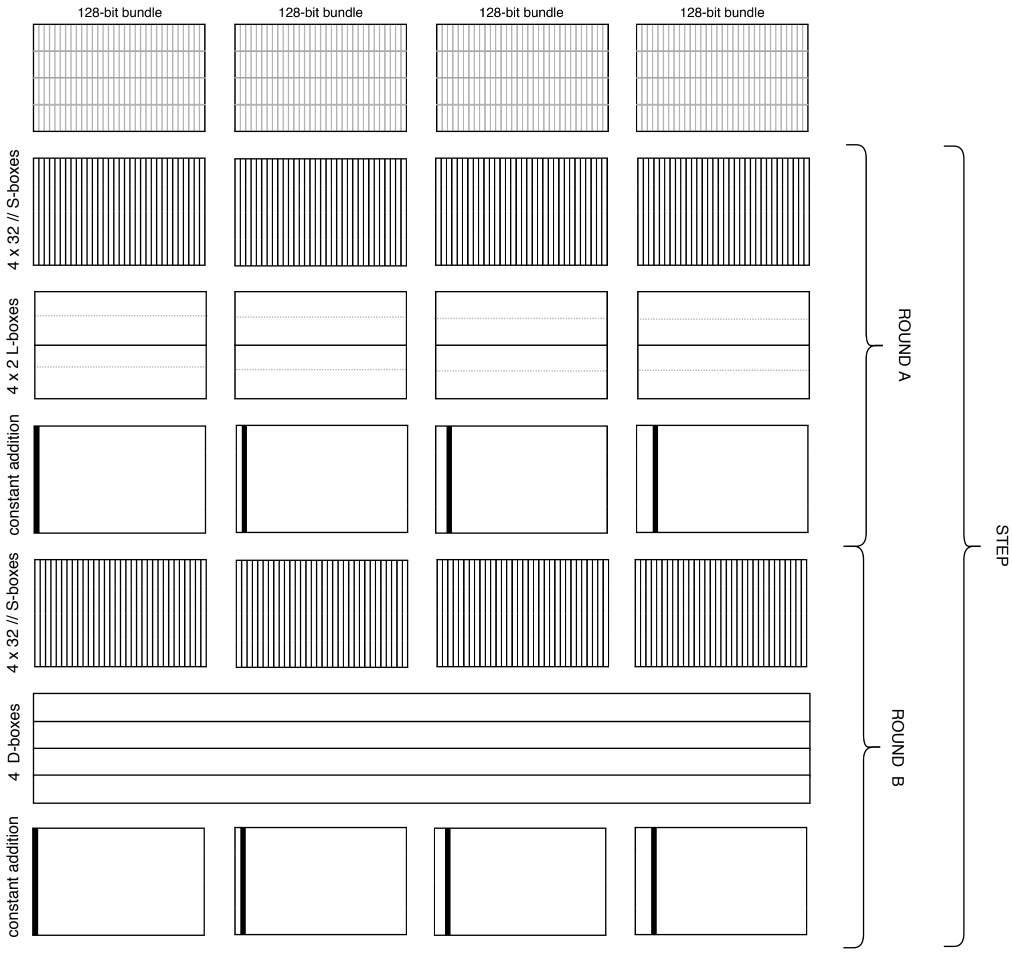

Figure 5: Round and step of Shadow-512: high-level view.

D Clyde-128 and Shadow-512 components

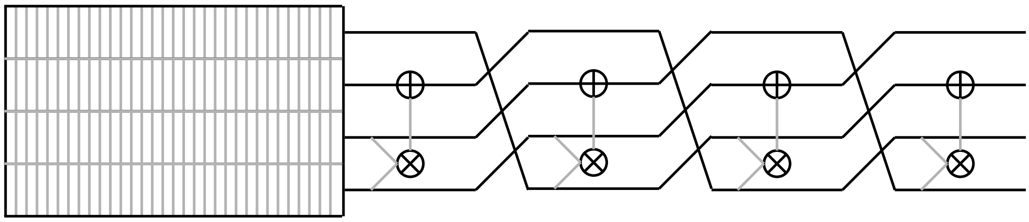

Figure 6: 32 parallel executions of the Clyde-128 and Shadow-512 S-box.

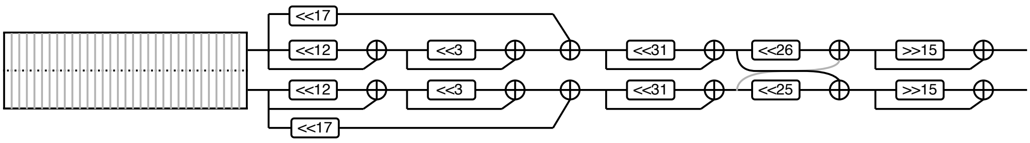

Figure 7: Clyde-128 and Shadow-512 L-box.

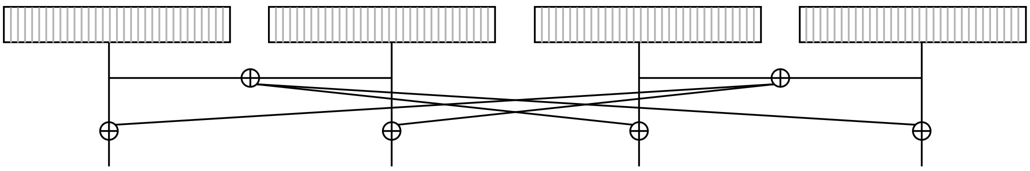

Figure 8: Shadow-512 diffusion layer.

E Inverse S-box implementation

- $y[3] = (x[0] \odot x[1]) \oplus x[2];$
- $y[0] = (x[1] \odot y[3]) \oplus x[3];$
- $y[1] = (y[3] \odot y[0]) \oplus x[0];$
- $y[2] = (y[0] \odot y[1]) \oplus x[1];$

F Inverse L-box implementation

- $a = x \oplus \mathsf{rot}(x, 25);$
- $b = y \oplus \mathsf{rot}(y, 25);$
- $c = x \oplus \mathsf{rot}(a, 31);$
- $d = y \oplus \mathsf{rot}(b, 31);$
- $c = c \oplus \mathsf{rot}(a, 20);$
- $d = d \oplus \mathsf{rot}(b, 20);$
- $a = c \oplus \mathsf{rot}(c, 31);$
- $b = d \oplus \mathsf{rot}(d, 31);$
- $c = c \oplus \mathsf{rot}(b, 26);$
- $d = d \oplus \mathsf{rot}(a, 25);$
- $a = a \oplus \mathsf{rot}(c, 17);$
- $b = b \oplus \mathsf{rot}(d, 17);$
- $a = \mathsf{rot}(a, 16);$
- $b = \mathsf{rot}(b, 16);$

G Spook v2 illustrations & equations

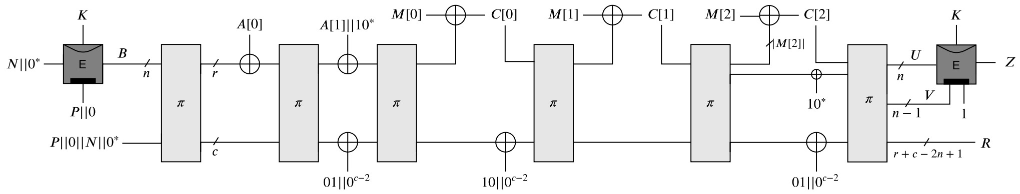

Figure 9: Spook v2 mode (switching $P||0||N||0^*$ and B in the first Shadow-512 call).

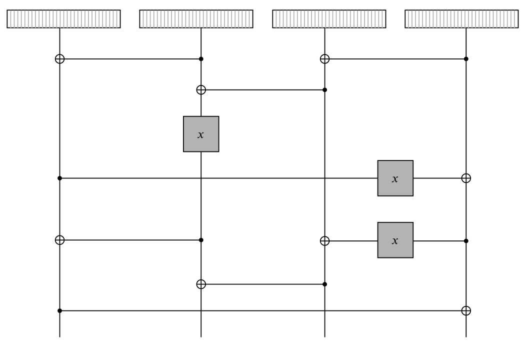

Figure 10: Shadow-512 v2 diffusion layer.

Shadow v2 diffusion. We use the ring of polynomials modulo $x^{32} + x^8 + 1$ with $\alpha = x$.

Shadow-512 v2

$$
\begin{aligned}
x_0 &:= x_0 \oplus x_1 \\
x_2 &:= x_2 \oplus x_3 \\
x_1 &:= x_1 \oplus x_2 \\
x_3 &:= x_3 \oplus \alpha x_0 \\
x_1 &:= \alpha x_1 \\
x_0 &:= x_0 \oplus x_1 \\
x_2 &:= x_2 \oplus \alpha x_3 \\
x_1 &:= x_1 \oplus x_2 \\
x_3 &:= x_3 \oplus x_0 \\
y_0 &:= x_0 \\
y_1 &:= x_1 \\
y_2 &:= x_2 \\
y_3 &:= x_3
\end{aligned}
$$

Shadow-384 v2

$$
\begin{aligned}
a &:= x_0 \oplus x_1 \\
b &:= x_0 \oplus x_2 \\
c &:= x_1 \oplus b \\
d &:= a \oplus \alpha b \\
y_0 &:= b \oplus d \\
y_1 &:= c \\
y_2 &:= d
\end{aligned}
$$

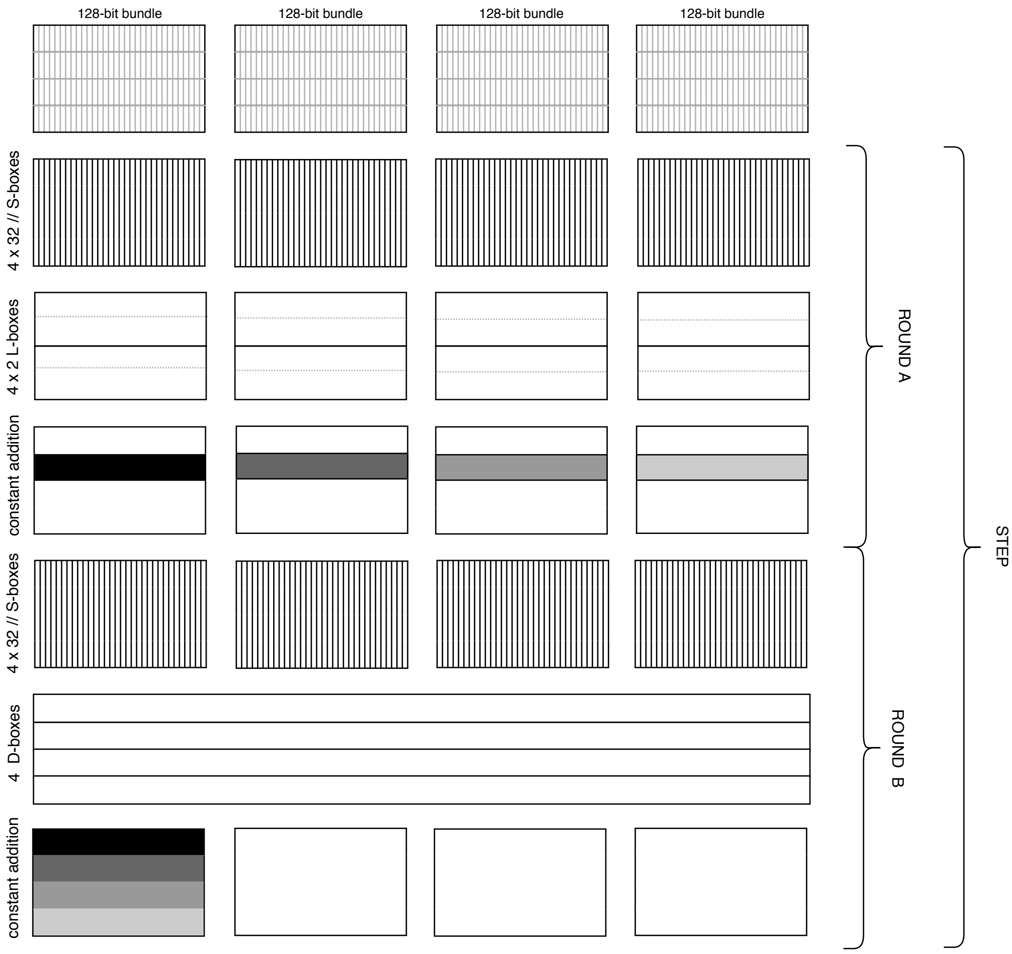

Figure 11: Round and step of Shadow-512 v2: high-level view.

H Spook v2 performances

The modifications of Spook v2 with respect to Spook v1 have an impact on the performance. In order to evaluate it, we implemented Spook v2 on the same platforms as Spook v1 (excepted in the high-end processors case for SIMD with 256 and 512 bit words, which were not bringing improvement over 128 bit SIMD in the Spook v1 case, which expect to be similar for Spook v2). We next observe the combined impact of the change of D-box, the change of round constants and the change in the initial state of Shadow.

H.1 Software implementations

For high-end platforms, the D-box change has the most significant impact (since it requires to transpose the Shadow state when viewed as a 4x4 matrix of 32-bit words), while the impact of the change of constants remains small since these constants are pre-computed anyway (and loads use different computing resources than the main computation). Overall, Spook v2 increases the cost by up to 22% on the selected platform.

For embedded platforms, Spook v2 either increases slightly (up to 2.5%) or reduces (up to 19%) the cost of software implementations, depending on the platorm and optimisation target. This is the combination of the cost increase of the D-box and the cost reduction of the constants, while the initialization change has no performance impact.

We note that in the case of size-optimized code, Spook v2 is faster on all the considered platforms. In this setting, the compiler does not inline functions (e.g., S-boxes), forcing the state to be stored in memory. Therefore before each constant addition, parts of the state have to be loaded from memory (and stored back afterwards). In Spook v1 (resp., Spook v2), at least 16 (resp., 4) loads and stores are needed in each round.

Table 15: High-end software performance results. Number of cycles compiled for various micro-architectures, and throughput (cycles per byte) for a message of 2048 bytes. The percentages are the comparison with the Spook v1 corresponding metrics. The Shadow-512 v2 32-bit implementatuib exhibits large cost increase on some platoforms due to the inability of the compiler to perform some (vectorization) optimizations.

	x86-64 (SSE2)	Haswell (AVX2)	Skylake-AVX512
Shadow-512 v2 (32-bit)	959 (105%)	831 (233%)	830 (242%)
Shadow-512 v2 (128-bit)	474 (116%)	444 (112%)	362 (119%)
Spook v2 (C32bit-S128bit)	15.3 (115%)	15.1 (114%)	12.3 (122%)

Table 16: Size-optimized performances on embedded platforms (-Os). The percentages are the comparison with the Spook v1 corresponding metrics.

	Size [Bytes]	Clyde-128 [Cycles]	Shadow-512 v2 [Cycles]	Spook v2 [Cycles/byte]
Cortex-M0	1864 (96%)	3274 (100%)	7520 (87%)	266
Cortex-M3	1860 (99%)	1763 (100%)	4394 (80%)	152
RI5CY	2044 (96%)	1851 (100%)	4309 (91%)	149

Table 17: Speed-optimized performances on embedded platforms (-O3). The percentages are the comparison with the Spook v1 corresponding metrics.

	Size [Bytes]	Clyde-128 [Cycles]	Shadow-512 v2 [Cycles]	Spook v2 [Cycles/byte]
Cortex-M0	4576 (99%)	2450 (100%)	5192 (82%)	169
Cortex-M3	3951 (103%)	802 (100%)	2407 (103%)	79
RI5CY	4624 (100%)	1259 (100%)	3787 (93%)	123

H.2 Hardware implementations

For the hardware implementations, the main change is that the combination of an S-box layer and a D-box layer cannot be split as a parallel combination of small permutations. Therefore, we changed the hardware architecture as illustrated in Figure 12, where the S-box of round B is moved to the round A. We additionally move the digestion unit, now performed during the round B instead of round A, in order to limit the critical path. Furthermore, the new constants are generated sequentially from an LFSR, forcing the bundles to be processed in-order (while it was out-of-order in the v1 architecture). This is done at zero cost thanks to the change in the initial state of Shadow. The slightly increased logic depth of the round A logic has a limited (3%) impact on the maximum frequency. Overall, the changes have thus a slight impact on the area requirement (4%) and a slightly larger impact on the energy consumption (about 15%).

Table 18: Spook v2: Artix-7 implementations results (post place-and-route). The percentages are the comparison with the Spook v1 corresponding metrics.

N_u	Opt. Strat.	Slices	Regs	LUTs	Freq. [MHz]	Lat. [Cycles]	TP [Mbps]	TPA [Mbps/LUT]
8	Speed	567 (100%)	1481 (102%)	2132 (101%)	179 (97%)	48 (100%)	954 (97%)	0.447 (96%)

Table 19: Spook v2 ASIC implementation results (post place-and-route) with $N_u = 8$. The percentages are the comparison with the Spook v1 corresponding metrics.

Area	Max. Freq [kGE]	Power [MHz]	Throughput [mW]	Energy [Mbps]	[pJ/bit]
18.2 (104%)	403 (97%)	8.75 (106%)	2149 (97%)	4.07 (109%)	

Table 20: Impact of clock frequency on the ASIC results with $N_u = 8$. The percentages are the comparison with the Spook v1 corresponding metrics.

Frequency [MHz]	Power [mW]	Throughput [Mbps]	Energy [pJ/bit]
403	8.75	2149	4.07
400	8.4 (100%)	2133	3.93
333	7.4 (111%)	1776	4.16
100	2.2 (112%)	533.3	4.125
10	0.22 (133%)	53.3	4.125
0.10	0.004	0.53	8.43

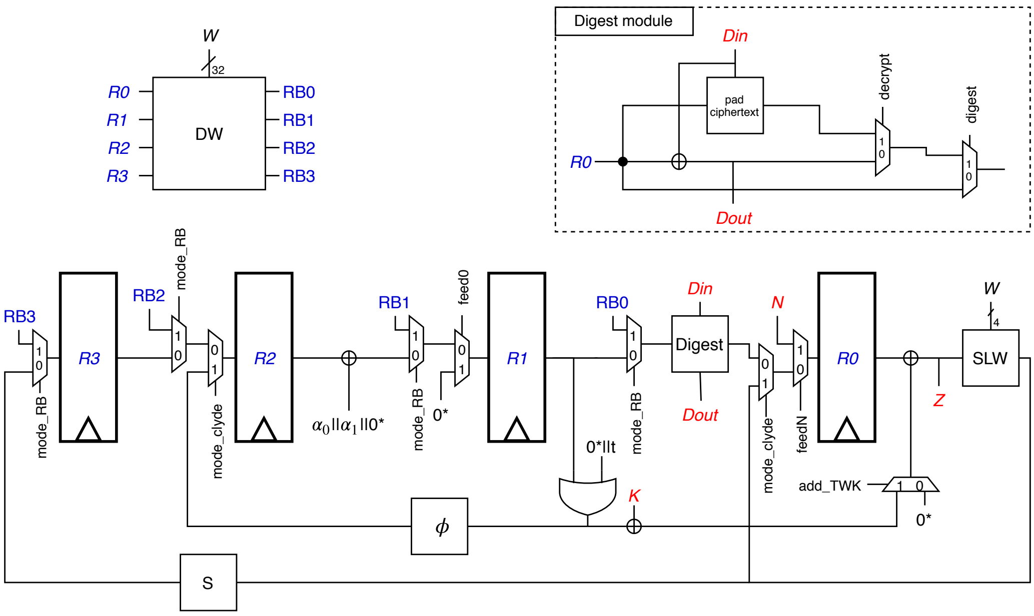

Figure 12: Architecture of Spook[128,512,su] v2 (unprotected, inverse-free variant).

I Deferred muCIML2 proof for Spook

Proof. To prove the theorem, we use a sequence of games. Given an adversary $\mathcal{A}$, we start with Game 0 which is the $\mathsf{muCIML2}_{\mathcal{A},\mathsf{Spook},\mathsf{L}^*,u}$ experiment and we end with a game where all the leaking decryption queries (i_j, N_j, A_j, C_j) are deemed invalid, including the last and (q_d+1)-th decryption which tests the validity of the potential forgery ciphertext (i, N, A, C). In the sequel, we make use of the notations introduced on Page 318.

Game 0. This game is depicted in Table 14. Let E_0 be the event that the adversary $\mathcal{A}^{\mathsf{L}^*}$ wins this game, that is, the output of the experiment is 1.

Game 1. We introduce a failure event F_1 with respect to Game 0, where F_1 occurs if among the at most $Q \leq \sigma + q_\pi + q_e + q_d + 1$ distinct computations of forward or backward evaluations of π there is at least one collision on the last $c-2 = 2n-2$ bits or the first $2n-1$ bits of any outputs. In Game 1, if F_1 occurs we abort the game and return 0. We let E_1 be the event that the adversary $\mathcal{A}^{\mathsf{L}^*}$ wins this game.

Bounding $|\Pr[E_0] - \Pr[E_1]|$. Since Game 0 and Game 1 are identical as long as F_1 does not occur, and $4 \leq Q$, we have:

$$|\Pr[E_0] - \Pr[E_1]| \leq \Pr[F_1] \leq Q^2/2^{2n-3}.$$

Note: from now on, in the case of a winning adversary, no TBC quadruple of the form $(i, \star, V\|1, Z)$ appears when answering to an LEnc query.

Game 2. We introduce a failure event F_2 with respect to Game 1, where F_2 occurs if there is an LDec query, including the forgery ciphertext, on some $(i', N', A', C' = c'\|Z')$ such that the ciphertext is valid and the TBC quadruple $(i', U', V'\|1, Z')$ appears for the first time in an LDec query. Then, $\Pr[E_1] \leq \Pr[E_2] + \Pr[E_1|F_2]$, where $E_2 = E_1|\neg F_2$.

Bounding $\Pr[E_1|F_2]$. A straightforwards argument gives $\Pr[F_2] \leq (q_d+1)\epsilon$.

Game 3. We introduce a failure event F_3 with respect to Game 2, where F_3 occurs if, at the end of the game, there is an input-output couple (R, S) defined in the π-history from a backward query and $R = \star\|\star\|0^n\|\star$. By definition, $E_3 = E_2|\neg F_3$.

Bounding $\Pr[F_3]$. It comes to bound the probability of computing a preimage to that 0^n which gives $\Pr[F_3] \leq q_\pi/2^n$, since $2^{r+c}/2^{3n} = 2^n$. Note: from now on, since $F_1 \cup F_3$ no more occur for a winning adversary, $U\|V$ can only be reached by a chain of states computed exclusively from some input $\star\|\star\|0^n\|\star$ by forward evaluations of π.

Game 4. We modify the winning condition of the previous game. In the finalization, once $\mathcal{A}$ outputs $(i, N, A, C = c\|Z)$ we say that $\mathcal{A}$ does not win and returns 0 if $\mathcal{A}$ fails as before or if $\pi(R_{\lambda+\ell}) = U\|V$ appears before the first apparition of $(V\|1, Z)$ as input to $\mathsf{E}^{-1}_{K_i}$ in a leaking decryption query. If we call F_4 the event that makes the adversary winning in Game 3 but loosing in Game 4, we have $|\Pr[E_4] - \Pr[E_3]| \leq \Pr[E_3|F_4]$, where $E_4 = E_3|\neg F_4$ is the event that $\mathcal{A}$ wins in this game.

Bounding $\Pr[E_4']$, for $E_4' := E_3|F_4$. If we call D_j the event that the first time $(V\|1, Z)$ appears during the computation of the answer to a leaking decryption query is in the j-th leaking decryption query $(i_j, N_j, A_j, C_j = c_j\|Z_j)$, that is necessarily invalid if E_4' occurs, we just have to bound $\Pr[E_4' \cap V_j]$, for all $j = 1$ to q_d. (Note that we do not have to guess the user i with such an argument.) When we process the j-th leaking decryption query until the computation of $U_j\|V_j$, we know that $V_j = V$ and $Z_j = Z$. At that time, let q_j be the total number of forward π evaluations made (online or offine) in the experiment. Let also $\mathcal{S}_j$ be the random variable counting the number of "V-collisions" with V_j, for $j = 1$ to

q_d. (Note that $\mathcal{S}_j \geq 2$ if E'_4 occurs.) We have:

$$\begin{aligned}\Pr[E'_4 \cap D_j] &= \sum_{s=2}^{q_j} \Pr[E'_4 \mid D_j \cap \mathcal{S}_j = s] \cdot \Pr[D_j \cap \mathcal{S}_j = s] \\ &\leq \sum_{s=2}^{q_j} \sum_{k=1}^{s-1} \Pr[E'_4 \mid D_j \cap \mathcal{S}_j = s \cap H_{j,k}] \cdot \Pr[\mathsf{V\text{-}Coll}(q_j) \geq s],\end{aligned}$$

where $H_{j,k}$ is the event that among the s distinct input state that "V"-collide on V_j the k-th one is $R_{\lambda+\ell}$. By convention, we always see $R^j_{\lambda^j+\ell^j}$ as the s-th and last such message even if the computation of $\pi(R^j_{\lambda^j+\ell^j})$ appears earlier than in the j-th LDec query[13]. Note that $\Pr[E'_4|H_{j,s}] = 0$. For each $k = 1$ to $s-1$, it is now easy to see that the event $E'_4 \mid (D_j \cap \mathcal{S}_j = s \cap H_{j,k})$ reduces to $\mathsf{muSUL2}$ by using the first $2n-1$ bits of the output state of the k-th "V-collisions," say $U_{j,k}\|V_{j,k}$ with $V_{j,k} = V_j$, and Z_j to compute the quadruple $(i_j, U_{j,k}, V_j, Z_j)$ as our guess against the TBC. Therefore,

$$\begin{aligned}\Pr[E'_4 \cap D_j] &\leq \epsilon \cdot \sum_{s=2}^{q_j} (s-1) \cdot \Pr[\mathsf{V\text{-}Coll}(q_j) \geq s] \\ &\leq \epsilon \cdot \frac{1}{2^{n-1}} \binom{q_j}{2} \left(1 + \frac{2q_j}{2^{n-1}}\right)\end{aligned}$$

by lemma 1, since $2q_j \leq 2Q \leq 2^{n-3} \leq 2^{n-1}$ by assumption on the number of queries. In addition, $1 + 2q_j/2^n \leq 5/4$. Summing on all the j's, gives:

$$\Pr[E'_4] \leq \epsilon \cdot \frac{1}{2^{n-1}} \cdot \frac{5}{4} \cdot \sum_{j=1}^{q_d} \binom{q_j}{2}.$$

Some basic computation shows that $\sum_{j=1}^{q_d} \binom{q_j}{2} \leq \sum_{j=1}^{q_d} \binom{Q-q_d+j}{2} \leq \frac{1}{2} q_d Q^2 (1 + \frac{2q_d}{Q})$, if $q_d \leq Q$. But then, as $q_d \leq Q/4$ by assumption, we have:

$$\Pr[E'_4] \leq \frac{q_d Q^2}{2^{n-1}} \cdot \epsilon.$$

Game 5. In this game we follow the specification of $\mathsf{muCIML2}_{\mathcal{A},\mathsf{Spook},\mathsf{L}^*}$ except that we always output 0 at the end of the game.

Bounding $|\Pr[E_5] - \Pr[E_4]| = \Pr[E_4]$. We end by showing that winning while the TBC input $(V\|1, Z)$ for inversion appears when answering a leaking decryption query before the computation of $\mathsf{H}(R_{\lambda+\ell})$ is negligible. For each (V_j, Z_j) that appears when answering a leaking decryption query and *before* any fresh computation of some $\pi(R')$, the probability that $\pi(R')[{:}\, 2n-1] = U^*_j\|V_j$ is $1/2^{2n-1}$ since the value $\pi(R')[{:}\, 2n-1]$ remains uniform and independent of the adversary's view. (Note that we still do not have to make a guess on i with this argument.) We now count the number of tries a winning adversary can make in it that case. Considering the event D_j as in the previous analysis of Game 4, in $E_4 \cap D_j$ there at most $Q - 2j$ remaining π evaluations left after the j-th LDec query. Then, $\Pr[E_4|D_j] \leq Q/2^{2n-1}$, for $i = 1$ to q_d. Note that $\Pr[E_4|D_{q_d+1}] = 0$ by definition. Finally, we get:

$$\Pr[E_4] \leq \frac{q_d Q}{2^{2n-1}}.$$

Hence, the bound of the theorem. □

[13] Note that this is without loss of generality as the enumeration only matters in the reduction at the time we get the adversary's j-th LDec query (i_j, N_j, A_j, C_j). The choice of (which is) the k-th state colliding on V_j can be made once the s input states are known.

The leading term that bounds the advantage is $\frac{q_d Q^2}{2^{n-1}\epsilon^{-1}}$. Any improvement of the term will give roughly the same improvement of the bound. In that respect, we can see that $q_j \leq Q - q_d + j$ is a pretty loose upper-bound. Indeed, it does not take into account that the among q_j forward evaluations of π the input-output couple must be "castable" in a next LDec query. A scope of improvement is thus to figure out whether an input-output pair of π can be among the chain of states for two different users. This direction could lead to a further ϵ dues to the fact that the adversary implicitly attacks E during the first evaluation of E in encryption or decryption. Furthermore, it could be shown that $q_{j_1} + \cdots + q_{j_l} \leq Q$, for some $l \leq q_d$, as long as all the users $i_{j_1}, \ldots, i_{j_l}$ are pairwise distinct. Finally, it could be worth determining if actually among the π input-output to take into account in $q_1 + \cdot + q_d$, the total is actually bound by some value not too bigger than Q, which will save, hopefully, a factor less but close to Q.

I.1 Multi-Collisions

Let $1 \leq s \leq q \leq N$. We consider the experiment where we uniformly throw q balls at random into N bins. $\mathsf{MultiColl}(N, q) \geq s$ denotes the event that at least one bin contains at least s balls. We recall a useful upper-bound on the probability of multi-collisions.

Theorem 2.

$$\Pr[\mathsf{MultiColl}(N, q) \geq s] \leq \frac{1}{N^{s-1}} \binom{q}{s}.$$

We also need the following technical result.

Lemma 1. *If* $2q \leq N$,

$$\sum_{s=1}^{q} (s-1) \cdot \Pr[\mathsf{MultiColl}(N, q) \geq s] \leq \frac{1}{N} \binom{q}{2} \left(1 + \frac{2q}{N}\right).$$

Proof. Looking at the generic term for $s \geq 3$ after applying the theorem leads to:

$$\frac{s-1}{N^{s-1}} \binom{q}{s} \leq \frac{1}{N^{s-2}} \binom{q}{s-1} \cdot \frac{q}{N} \leq \frac{1}{N} \binom{q}{2} \cdot \left(\frac{q}{N}\right)^{s-2}.$$

Then, the whole sum is upper-bounded by:

$$\frac{1}{N} \binom{q}{2} \cdot \sum_{s=2}^{q} \left(\frac{q}{N}\right)^{s-2} \leq \frac{1}{N} \binom{q}{2} \frac{N}{N-q} = \frac{1}{N} \binom{q}{2} \left(1 + \frac{q}{N-q}\right).$$

Hence, the result since $q \leq N - q$. □

J MILP Model for Division Property Analysis

SageMath code to generate the MILP model for the division property analysis of Clyde-128 is listed below. For easier verification and further work, it can also be found online at https://gist.github.com/pfasante/3a2f087e74cd0f2a10853c8a5d036d85.

```
from sage.crypto.boolean_function import BooleanFunction
from sage.crypto.sbox import SBox

def algebraic_normal_form(self):
    """
    Computes the algebraic normal forms (ANFs) of every coordinate.
```

```
    """
    n = self.input_size()
    return [self.component_function(i).algebraic_normal_form()
            for i in [1<<j for j in range(n)]]

SBox.algebraic_normal_form = algebraic_normal_form

def division_trail(self, k):
    """
    Computes the output division property for the starting input dp k.

    INPUT:

        - ``k``, the input division property
    """

    def gt(a, b):
        """
        check whether a >= b
        """
        from operator import ge
        return all(map(lambda x: ge(*x), zip(a, b)))

    n = self.input_size()

    S = set()
    for e in range(2^n):
        kbar = ZZ(e).digits(base=2, padto=n)
        if gt(kbar, k):
            S.add(tuple(kbar))

    ys = self.algebraic_normal_form()[::-1]
    P = ys[0].ring()
    x = P.gens()[::-1]

    F = set()
    for kbar in S:
        F.add(P(prod([x[i] for i in range(n) if kbar[i] == 1])))

    Kbar = set()
    for e in range(2^n):
        u = ZZ(e).digits(base=2, padto=n)
        puy = prod([ys[i] for i in range(n) if u[i] == 1])
        puyMon = P(puy).monomials()
        contains = False
        for mon in F:
            if mon in puyMon:
                contains = True
                break

        if contains:
            Kbar.add(tuple(u))
```

```
    K = []
    for kbar in Kbar:
        greater = False
        for kbar2 in Kbar:
            if(kbar != kbar2 and gt(kbar, kbar2)):
                greater = True
                break
        if not greater:
            K.append(kbar)

    return sorted(K)

SBox.division_trail = division_trail

def division_trail_table(self):
    """
    Return a dict containing all possible division propagation of the SBOX,
    where y is a list containing the ANF of each output bits
    """
    n = self.input_size()

    D = dict()
    for c in range(2^n):
        k = tuple(ZZ(c).digits(base=2, padto=n))
        D[k] = self.division_trail(k)

    return D

SBox.division_trail_table = division_trail_table

def sbox_inequalities(sbox, analysis="differential", algorithm="greedy", big_endian=False):
    """
    Computes inequalities for modeling the given S-box.

    INPUT:

    - ``sbox`` - the S-box to model
    - ``analysis`` - string, choosing between 'differential' and 'linear' cryptanalysis
        or 'division_property'
        (default: ``differential``)
    - ``algorithm`` - string, choosing the algorithm for computing the S-box model,
        one of ['none', 'greedy', 'milp'] (default: ``greedy``)
    - ``big_endian`` - representation of transitions vectors (default: little endian)
    """
    ch = convex_hull(sbox, analysis, big_endian)

    if algorithm is "greedy":
        return cutting_off_greedy(ch)
    elif algorithm is "milp":
        return cutting_off_milp(ch)
    elif algorithm is "none":
```

```
        return list(ch.inequalities())
    else:
        raise ValueError("algorithm (%s) has to be one of ['greedy', 'milp']" % \
                         (algorithm,))

SBox.milp_inequalities = sbox_inequalities

def convex_hull(sbox, analysis="differential", big_endian=False):
    """
    Computes the convex hull of the differential or linear behaviour of the given S-box.

    INPUT:

    - ``sbox`` - the S-box for which the convex hull should be computed
    - ``analysis`` - string choosing between differential and linear behaviour
      (default: ``differential``)
    - ``big_endian`` - representation of transitions vectors (default: little endian)
    """
    from sage.geometry.polyhedron.constructor import Polyhedron

    if analysis is "differential":
        valid_transformations_matrix = sbox.difference_distribution_table()
    elif analysis is "linear":
        valid_transformations_matrix = sbox.linear_approximation_table()
    elif analysis is "division_property":
        valid_transformations = sbox.division_trail_table()
    else:
        raise TypeError("analysis (%s) has to be one of ['differential', 'linear']" % \
                        (analysis,))

    if analysis is "division_property":
        points = [tuple(x) + tuple(y)
                  for x, ys in valid_transformations.iteritems() for y in ys]
    else:
        n, m = sbox.input_size(), sbox.output_size()

        if big_endian:
            def to_bits(x):
                return ZZ(x).digits(base=2, padto=sbox.n)
        else:
            def to_bits(x):
                return ZZ(x).digits(base=2, padto=sbox.n)[::-1]

        points = [to_bits(i) + to_bits(o)
                for i in range(1 << n)
                for o in range(1 << m)
                if valid_transformations_matrix[i][o] != 0]

    return Polyhedron(vertices=points)

def cutting_off_greedy(poly):
    """
```

```
    Computes a set of inequalities that is cutting-off equivalent to the
    H-representation of the given convex hull.

    INPUT:

    - ``poly`` - the polyhedron representing the convex hull
    """
    from sage.modules.free_module import VectorSpace
    from sage.modules.free_module_element import vector
    from sage.rings.finite_rings.finite_field_constructor import GF
    from sage.modules.free_module_element import vector

    chosen_ineqs = []

    poly_points = poly.integral_points()
    remaining_ineqs = list(poly.inequalities())
    impossible = [vector(poly.base_ring(), v)
                  for v in VectorSpace(GF(2), poly.ambient_dim())
                  if v not in poly_points]

    while impossible != []:

        if len(remaining_ineqs) == 0:
            raise ValueError("no more inequalities to choose, but still "\
                             "%d impossible points left" % len(impossible))

        # find inequality in remaining_ineqs that cuts off the most
        # impossible points and add this to the chosen_ineqs
        ineqs = []
        for i in remaining_ineqs:
            cnt = sum(map(lambda x: not(i.contains(x)), impossible))
            ineqs.append((cnt, i))
        chosen_ineqs.append(sorted(ineqs, reverse=True)[0][1])

        # remove ineq from remaining_ineqs
        remaining_ineqs.remove(chosen_ineqs[-1])

        # remove all cut off impossible points
        impossible = [v
                      for v in impossible
                      if chosen_ineqs[-1].contains(v)
                      ]
    return chosen_ineqs

def cutting_off_milp(poly, number_of_ineqs=None, **kwargs):
    """
    Computes a set of inequalities that is cutting-off equivalent to the
    H-representation of the given convex hull by solving a MILP.

    The representation can either be computed from the minimal number of
    necessary inequalities, or by a given number of inequalities. This
    second variant might be faster, because the MILP solver that later
```

```
uses this representation can do some optimizations itself.

INPUT:

- ``poly`` - the polyhedron representing the convex hull
- ``number_of_ineqs`` - integer; either `None` (default) or the number
    of inequalities that should be used for representing the S-box.

REFERENCES:

- [SasTod17]_ "New Algorithm for Modeling S-box in MILP Based
    Differential and Division Trail Search"
"""

from sage.matrix.constructor import matrix
from sage.modules.free_module import VectorSpace
from sage.modules.free_module_element import vector
from sage.numerical.mip import MixedIntegerLinearProgram
from sage.rings.finite_rings.finite_field_constructor import GF

ineqs = list(poly.inequalities())
poly_points = poly.integral_points()
impossible = [vector(poly.base_ring(), v)
              for v in VectorSpace(GF(2), poly.ambient_dim())
              if v not in poly_points]

# precompute which inequality removes which impossible point
precomputation = matrix(
    [[int(not(ineq.contains(p)))
      for p in impossible]
     for ineq in ineqs]
)

milp = MixedIntegerLinearProgram(maximization=False, **kwargs)
var_ineqs = milp.new_variable(binary=True, name="ineqs")

# either use the minimal number of inequalities for the representation
if number_of_ineqs is None:
    milp.set_objective(sum([var_ineqs[i] for i in range(len(ineqs))]))
# or the given number
else:
    milp.add_constraint(sum(
        [var_ineqs[i]
         for i in range(len(ineqs))]
    ) == number_of_ineqs)

nrows, ncols = precomputation.dimensions()
for c in range(ncols):
    lhs = sum([var_ineqs[r]
               for r in range(nrows)
               if precomputation[r][c] == 1])
    milp.add_constraint(lhs >= 1)
```

```
    milp.solve()

    remaining_ineqs = [
        ineq
        for ineq, (var, val) in zip(ineqs, milp.get_values(var_ineqs).iteritems())
        if val == 1
    ]

    return remaining_ineqs

def milp_spook_sbox_constraints(milp, sbox, xi, yi):
    sbox_ineqs = sbox.milp_inequalities(analysis="division_property", algorithm="greedy")

    permuted_bits = matrix(ZZ, 4, 32, range(128)).columns()
    in_outs = [([xi[i] for i in sbox_indices], [yi[i] for i in sbox_indices])
               for sbox_indices in permuted_bits]

    for ineq in sbox_ineqs:
        for inputs, outputs in in_outs:
            milp.add_constraint(sum([inputs[i] * ineq[i+1]
                                     for i in range(len(inputs))] +
                                    [outputs[i] * ineq[i+1+len(inputs)]
                                     for i in range(len(outputs))]
                                   ) + ineq[0] >= 0)

def rotate(x, n):
    return x[n:] + x[:n]

def copy2(milp, x, y0, y1):
    for i in range(len(x)):
        milp.add_constraint(x[i] - y0[i] - y1[i] == 0)

def copy3(milp, x, y0, y1, y2):
    for i in range(len(x)):
        milp.add_constraint(x[i] - y0[i] - y1[i] - y2[i] == 0)

def xor2(milp, x0, x1, y):
    for i in range(len(x0)):
        milp.add_constraint(x0[i] + x1[i] - y[i] == 0)

def milp_spook_llayer_constraints(milp, xi, yi, ai, bi, rnd=0):
    s = milp.new_variable(binary=True, name="tmp_s")
    t = milp.new_variable(binary=True, name="tmp_t")
    u = milp.new_variable(binary=True, name="tmp_u")
    v = milp.new_variable(binary=True, name="tmp_v")

    s0 = [s[rnd, 0, i] for i in range(32)]
    s1 = [s[rnd, 1, i] for i in range(32)]
    s2 = [s[rnd, 2, i] for i in range(32)]
    s3 = [s[rnd, 3, i] for i in range(32)]
    s4 = [s[rnd, 4, i] for i in range(32)]
```

```
s5 = [s[rnd, 5, i] for i in range(32)]
s6 = [s[rnd, 6, i] for i in range(32)]
s7 = [s[rnd, 7, i] for i in range(32)]

copy3(milp, xi, s0, s1, s2)
xor2(milp, s1, rotate(s2, 12), s3)
copy2(milp, s3, s4, s5)
xor2(milp, s4, rotate(s5, 3), s6)
xor2(milp, s6, rotate(s0, 17), s7)

t0 = [t[rnd, 0, i] for i in range(32)]
t1 = [t[rnd, 1, i] for i in range(32)]
t2 = [t[rnd, 2, i] for i in range(32)]
t3 = [t[rnd, 3, i] for i in range(32)]
t4 = [t[rnd, 4, i] for i in range(32)]
t5 = [t[rnd, 5, i] for i in range(32)]
t6 = [t[rnd, 6, i] for i in range(32)]
t7 = [t[rnd, 7, i] for i in range(32)]

copy3(milp, yi, t0, t1, t2)
xor2(milp, t1, rotate(t2, 12), t3)
copy2(milp, t3, t4, t5)
xor2(milp, t4, rotate(t5, 3), t6)
xor2(milp, t6, rotate(t0, 17), t7)

s8 = [s[rnd, 8, i] for i in range(32)]
s9 = [s[rnd, 9, i] for i in range(32)]
u0 = [u[rnd, 0, i] for i in range(32)]
u1 = [u[rnd, 1, i] for i in range(32)]
u2 = [u[rnd, 2, i] for i in range(32)]
u3 = [u[rnd, 3, i] for i in range(32)]
u4 = [u[rnd, 4, i] for i in range(32)]

copy3(milp, s7, s8, u0, u1)
xor2(milp, rotate(u0, 31), u1, u2)
copy2(milp, u2, u3, u4)
xor2(milp, s8, rotate(u3, 15), s9)

t8 = [t[rnd, 8, i] for i in range(32)]
t9 = [t[rnd, 9, i] for i in range(32)]
v0 = [v[rnd, 0, i] for i in range(32)]
v1 = [v[rnd, 1, i] for i in range(32)]
v2 = [v[rnd, 2, i] for i in range(32)]
v3 = [v[rnd, 3, i] for i in range(32)]
v4 = [v[rnd, 4, i] for i in range(32)]

copy3(milp, t7, t8, v0, v1)
xor2(milp, rotate(v0, 31), v1, v2)
copy2(milp, v2, v3, v4)
xor2(milp, t8, rotate(v3, 15), t9)

xor2(milp, s9, rotate(v4, 26), ai)
```

```
    xor2(milp, t9, rotate(u4, 25), bi)

def milp_model_spook(initial_dp, rnds=1):
    sbox = SBox([0, 8, 1, 15, 2, 10, 7, 9, 4, 13, 5, 6, 14, 3, 11, 12])

    from itertools import product

    # initialise MILP object
    milp = MixedIntegerLinearProgram(maximization=False, solver="CPLEX")

    # sbox layer inputs
    xs = milp.new_variable(binary=True, name="x", indices=product(range(rnds), range(128)))
    # sbox layer outputs / linear layer inputs
    ys = milp.new_variable(binary=True, name="y", indices=product(range(rnds), range(128)))
    # linear layer outputs
    zs = milp.new_variable(binary=True, name="z", indices=product(range(rnds), range(128)))

    # model for each round the sbox layer and linear layer transitions
    for r in range(rnds):
        xi = [xs[(r, i)] for i in range(128)]
        yi = [ys[(r, i)] for i in range(128)]
        milp_spook_sbox_constraints(milp, sbox, xi, yi)

        yi = [[ys[(r, 32*j+i)] for i in range(32)] for j in range(4)]
        zi = [[zs[(r, 32*j+i)] for i in range(32)] for j in range(4)]
        milp_spook_llayer_constraints(milp, yi[0], yi[1], zi[0], zi[1], rnd=(r, 0))
        milp_spook_llayer_constraints(milp, yi[2], yi[3], zi[2], zi[3], rnd=(r, 1))

        # link each rounds output with next rounds input
        if r < rnds-1:
            for i in range(128):
                milp.add_constraint(zs[(r, i)] == xs[(r+1, i)])

    # Set input variables to initial division property
    from sage.crypto.sbox import integer_types
    if type(initial_dp) in integer_types + (Integer, ):
        initial_dp = ZZ(initial_dp).digits(base=2, padto=128)
    for i in range(128):
        milp.add_constraint(xs[(0, i)] == initial_dp[i])

    # Objective function is to minimize the weight of the output division property
    milp.set_objective(sum(zs[(rnds-1, i)] for i in range(128)))

    return milp, xs, ys, zs


def check_dp(rnds=1, milp_model=milp_model_spook, block_size=128):
    from sage.numerical.mip import MIPSolverException

    for i in range(block_size):
        k = ((1<<block_size) - 1)^^(1<<i)
        milp, xs, ys, zs = milp_model(initial_dp=k, rnds=rnds)
```

```
        cnt = 0
        found_unit_vector = True
        while found_unit_vector:
            try:
                obj = int(milp.solve())
                inp = [int(x)
                       for x in milp.get_values([xs[(   0   , j)]
                                                 for j in range(block_size)])]
                out = [int(x)
                       for x in milp.get_values([zs[(rnds-1, j)]
                                                 for j in range(block_size)])]

                inpstr = "".join(map(lambda x:"%d" % x, inp))
                outstr = "".join(map(lambda x:"%d" % x, out))

                cnt += 1
                print("%3d/%3d: %3d %s -> %s" % (i, cnt, obj, inpstr, outstr))

                if obj > 1:
                    print("found a distinguisher:")
                    print("%3d: %3d %s -> %s" % (i, obj, inpstr, outstr))
                    return inp, out
                else:
                    idx = out.index(1)
                    milp.add_constraint(zs[(rnds-1, idx)] == 0)

            except MIPSolverException as e:
                print("i = %d: no feasible solution" % i)
                found_unit_vector = False

    return None, None


if __name__ == "__main__":
    import sys
    if len(sys.argv) < 2:
        print("Usage:\n%s rounds" % (sys.argv[0]))
        sys.exit(1)

    rounds = int(sys.argv[1])
    check_dp(rounds, milp_model_spook, block_size=128)
```