Submission to the NIST Lightweight Cryptography Standardization

ForkAE v.1

Designers/Submitters (listed in alphabetical order)

Elena Andreeva[1]
elena.andreeva@kuleuven.be
+32 16321800

Virginie Lallemand[2]
virginie.lallemand@loria.fr
+33 354958659

Antoon Purnal[1]
antoon.purnal@kuleuven.be
+32 16193375

Reza Reyhanitabar[3]
reza.reyhanitabar@elektrobit.com
+49 913177016208

Arnab Roy[4]
arnab.roy@bristol.ac.uk
+44 01179545488

Damian Vizár[5]
damian.vizar@csem.ch
+41 327205269

Affiliations (of corresponding designers)

[1] imec-COSIC, KU Leuven, Kasteelpark Arenberg 10 - bus 2452, 3001 Heverlee, Belgium
[2] Université de Lorraine, CNRS, Inria, LORIA-Campus Scientifique - BP, 239, 54506, Vandoeuvre-les-Nancy, France
[3] Elektrobit Automotive GmbH, Am Wolfsmantel 46, 91058 Erlangen, Germany
[4] University of Bristol, Wooldland Road, Bristol BS8 1UB, United Kingdom
[5] CSEM SA, Jaquet-Droz 1, CH-2002 Neuchâtel, Switzerland

https://www.esat.kuleuven.be/cosic/forkae/

Contents

1 Introduction **1**

2 ForkSkinny Family **3**
2.1 Primary NIST compliant recommendation . . . 4
2.2 Use case recommendations . . . 4
2.3 Operational limits . . . 4

3 Notation **4**
3.1 Forkcipher . . . 5
3.2 Authenticated Encryption . . . 6

4 Specification **7**
4.1 ForkSkinny . . . 7
4.2 Parallel AEAD from a Forkcipher . . . 13
4.3 Sequential AEAD from a Forkcipher . . . 14

5 Security Claims **15**
5.1 NIST security requirement . . . 15
5.2 Security for our primary and targeted use cases. . . . 16

6 Security Analysis **17**
6.1 Cryptanalysis of ForkSkinny . . . 17
6.2 Modes . . . 18

7 Efficiency/Implementation **18**

8 Design Rationale **20**
8.1 Design Decisions in the Primitive Level . . . 20
8.2 Design Decisions in the Modes of Operation . . . 23
8.3 Distinct Instances, Advantages and Limitations . . . 24

A Formal Definitions of Forkcipher and Authentiated Encryption **30**
A.1 Syntax . . . 30
A.2 Security Definition of Forkcipher . . . 31
A.3 Security Definition of Authenticated Encryption . . . 31

B Security Analysis of PAEF **31**

C Security Analysis of SAEF **35**

D Security Analysis of ForkSkinny 40
D.1 Arguments deduced from the Security of SKINNY 40
D.2 Differential and Linear analysis 41
D.3 Impossible Differential 42
D.4 Boomerang Attack 44
D.5 Meet-in-the-Middle Attack 44
D.6 Integral Attack 45
D.7 Algebraic Attack 46

E The Sboxes of SKINNY 46

1 Introduction

This document introduces the ForkAE family of algorithms, our submission to the NIST's lightweight cryptography project, for the category of authenticated encryption with associated data (AEAD). ForkAE is a new family of AEAD schemes optimized for the cost of handling *short messages*. Its design combines performance, security and simplicity in a modular package that is easy to analyze. This is achieved by combining and enhancing novel provably secure modes of operation with known and well-analyzed lightweight primitives and operations from [9, 14].

Most AEAD schemes are optimized for the cost of handling long messages. However, an increasingly common scenario is that AEAD algorithms are applied to many small messages. Examples are Secure Onboard Communication (SecOC) in automotive systems [6], handling of short data bursts in critical communication and massive IoT domains of 5G [1], and Narrowband IoT (NB-IoT) [2, 5] systems. For instance, the new CAN FD standard (ISO 11898-1) for vehicle bus technology [3, 4], which is expected to be implemented in most cars by 2020, allows for a payload up to 64 bytes. In NB-IoT standard [2, 5] the maximum transport block size (TBS) is 680 bits in downlink and 1000 bits in uplink (the minimum TBS size is 16 bits in both cases). In NB-IoT use cases such as smart parking lots the actual data to be sent is just a bit (for free or occupied status), so a minimum allowed TBS size of 2 bytes (16 bits) would suit the application.

In the call for submissions for the lightweight cryptography project, NIST has stressed as a *design requirement* that lightweight AEAD submissions shall be "optimized to be efficient for short messages (e.g., as short as 8 bytes)" [35].

The main design goal for the ForkAE family is to excel at processing short inputs, while also being able to process arbitrary-length inputs, albeit somewhat less efficiently. To achieve this goal, the ForkAE family is built based on a novel combination of the following three well-analyzed ingredients:

- Forkcipher: a symmetric-key building block that is introduced in [9] as an essential ingredient in the quest for lightweight AEAD for very short inputs.
- `SKINNY`: a lightweight tweakable blockcipher family from [14] with very appealing hardware/software performances and strong security guarantees with regards to known attacks.
- SAEF and PAEF: two provably-secure modes of operation for a forkcipher that are defined in [9].

Forkcipher. A forkcipher [9] is *nearly*—but not exactly—a fixed-input-length AEAD primitive; "nearly" because it produces *expanded* ciphertexts with a non-trivial redundancy, and not exactly because it has no integrity-checking mechanisms. When keyed and tweaked, a forkcipher maps an n-bit input block to an output of $2n$ bits. Intuitively, evaluating a secure forkcipher on an input M is equivalent to evaluating *two independent* tweakable permutations on M but with an *amortized computational cost*.

ForkSkinny. This is our concrete instantiation for a forkcipher family based on the lightweight tweakable blockcipher family SKINNY [14]. Our motivation for the choice of SKINNY was both its area, throughput, power, efficiency and software advantages in lightweight applications, as well as extensive analysis with regard to state-of-the-art cryptanalytic techniques [10, 11, 37, 42, 45, 46]. SKINNY is a SPN (substitution permutation network) and "AND-rotation-XOR"-mix design with a tweakey schedule following the TWEAKEY approach of [25] and a linear feedback shift register (LFSR) to minimize hardware costs. Both SPN and TWEAKEY allow for stronger bounds on the number of active Sboxes, while the addition of "AND-rotation-XOR" and LFSR allow for efficiency optimizations.

PAEF and SAEF Modes of Operation. To build our full-fledged AEAD schemes optimized exclusively for processing short inputs, we use two provably secure modes of operation for a forkcipher. PAEF (**P**arallel **AE**AD from a **F**orkcipher) is a fully parallelizable mode and is suited for applications where one of the parties is able to perform parallel computations (e.g., in dedicated hardware). SAEF (**S**equential **AE**AD from a **F**orkcipher) has an online encryption, but is not parallelizable. SAEF lends itself well to low-overhead implementations, as it does not need to store the nonce or any block counters. Both PAEF and SAEF use a unique nonce input value.

ForkAE family. Our proposed family of variable-input-length (VIL) AEAD schemes is obtained by applying PAEF and SAEF modes of operation to the ForkSkinny family of primitives. Our primary recommendation is **PAEF-ForkSkinny-128-288** that is fully compliant with the functional and security requirements set by NIST for the LWC project. PAEF-ForkSkinny-128-288 is a member of the ForkAE family obtained by plugging the ForkSkinny-128-288 primitive in the PAEF mode. ForkSkinny-128-288 is a forkcipher with a 128-bit block and 288-bit tweakey. The full-fledged VIL AEAD scheme, PAEF-ForkSkinny-128-288, supports a 128-bit key, 104-bit nonce and 128-bit tag.

The design of ForkAE family combines security, efficiency and a great deal of flexibility in a modular package:

The ForkAE family is secure. ForkSkinny is benefiting from security properties of SKINNY, and lessons learned from an extensive *third-party cryptanalysis* of ForkAES [12] (Fork**AES** was put forth in [9] as an initial proof of concept for a forkcipher based on AES). The modes are *provably secure* [9]; PAEF achieves n-bit and SAEF $n/2$-bit AE (confidentiality and integrity) security.

The ForkAE family is efficient. Inheriting most of the lightweight implementation features from SKINNY, it provides excellent throughput per area in hardware with round-based implementation and facilitates many trade-offs in the speed-resource design space on a variety of hardware and software platforms.

The ForkAE family is flexible. For a 128-bit key, it supports different block lengths (64 and 128 bits), nonce lengths (120, 112, 104, 56 and 48 bits), and tag lengths (64 and 128 bits) which allows for a number of performance and security tradeoffs.

2 ForkSkinny Family

Our ForkAE family comprises of 6 members which output a variable-length ciphertext on inputs a variable-length plaintext, variable-length associated data and a fixed-length nonce with a fixed-length key in two modes: PAEF and SAEF, and are based on the forkcipher family ForkSkinny. All members are externally parameterized by the mode, the block size n and the tweakey size t (determining the ForkSkinny-n-t primitive instantiation). The tweakey consists of a tweak and the fixed key where the nonce is an internal parameter and part of the tweak. The members of our ForkAE family can be instantiated with tunable nonces of up to a maximal permissible length. We propose ForkAE *concrete instances* with fixed nonce sizes in Table 1.

mode	primitive	block	tweakey	key	nonce	tag	max. nonce
PAEF	ForkSkinny-64-192	64	192	128	48	64	60
	ForkSkinny-128-192	128	192	128	48	128	60
	ForkSkinny-128-256	128	256	128	112	128	124
	ForkSkinny-128-288	**128**	**288**	**128**	**104**	**128**	156
SAEF	ForkSkinny-128-192	128	192	128	56	128	60
	ForkSkinny-128-256	128	256	128	120	128	124

Table 1: The ForkAE instances with the mode, the ForkSkinny-n-t primitive, the block, the tweakey, the key, the nonce, and the tag *lengths in bits*. Our primary recommendation is in bold. The max. nonce column indicates the maximum permissible nonce bit size that may be used in the corresponding mode with ForkSkinny.

The tweakey consists of a tweak and a key. The tweak is a known public value and for our uses consists of a public nonce (unique per message), a domain separator (DS), and a block counter (used only in PAEF) as described in the Table 2.

mode	ForkSkinny-n-t	nonce	DS	block counter	key	tweakey
PAEF	64-192	48	3	13	128	192
	128-192	48		13		192
	128-256	112		13		256
	128-288	104		53		288
SAEF	128-192	56	4	no	128	192
	128-256	120		no		256

Table 2: Tweakeys for all ForkAE instances in *in bits*.

2.1 Primary NIST compliant recommendation

Our primary ForkAE recommendation **PAEF-ForkSkinny-128-288** is compliant with the functional and security requirements set by NIST.

2.2 Use case recommendations

Our primary use case recommendation for *all* ForkAE members are applications which process *short messages* of up to 4 data blocks.

2.3 Operational limits

We summarize the algorithmic or operational limits:

1) the maximum number for AEAD encryption calls with a single key (calls/key), and
2) the data (in bytes) limits per single AEAD call (bytes/message);

for each of the instances in Table 3. We stress that these values reflect only *algorithm specific* limits and do **not** take security into consideration.

mode	ForkSkinny-n-t	nonce size	calls/key	bytes/message	rnd nonce?
PAEF	64-192	48	2^{48}	2^{16}	no
	128-192	48	2^{48}	2^{17}	no
	128-256	112	2^{112}	2^{17}	OK
	128-288	104	2^{104}	2^{57}	OK
SAEF	128-192	56	2^{56}	no limit	no
	128-256	120	2^{120}	no limit	OK

Table 3: The algorithmic limits of the ForkAE family. The column “rnd nonce?” is set to “no” when the nonce is only used as a counter per message (or equivalent stateful mode) and to “OK” for instances where we admit the use of a random nonce (the max. number of encryption calls with a random nonce per message ought to be strictly less than square root of the “calls/key” value).

3 Notation

This section introduces the notation and conventions used in the following sections.

All strings are binary strings. The set of all strings of length n (for a positive integer n) is denoted $\{0,1\}^n$. We let $\{0,1\}^{\leq n}$ denote the set of all strings of length at most n.

We let $\mathsf{left}_\ell(X)$ denote the ℓ leftmost bits of X and $\mathsf{right}_r(X) = X[(|X|-r)\ldots(|X|-1)]$ the r rightmost bits of X, such that $X = \mathsf{left}_\chi(X)\|\mathsf{right}_{|X|-\chi}(X)$ for any $0 \leq \chi \leq |X|$. Given a (possibly implicit) positive integer n and an $X \in \{0,1\}^*$, we let denote

$X\|10^{n-(|X| \bmod n)-1}$ for simplicity. Given an implicit block length n, we let $\mathsf{pad10}(X) = X\|10^*$ return X if $|X| \equiv 0 \pmod n$ and $X\|10^*$ otherwise.

Given a string X and an integer n, we let $X_1, \dots, X_x, X_* \overset{n}{\leftarrow} X$ denote partitioning X into n-bit blocks, such that $|X_i| = n$ for $i = 1, \dots, x$, $0 \leq |X_*| \leq n$ and $X = X_1\| \dots \|X_x\|X_*$, so $x = \max(0, \lfloor(|X| - 1)/n\rfloor)$. We let $|X|_n = \lceil X/n \rceil$. We let $(M', M_*) = \mathsf{msplit}_n(M)$ (as in message split) denote a splitting of a string $M \in \{0,1\}^*$ into two parts $M'\|M_* = M$, such that $|M_*| \equiv |M| \pmod n$ and $0 \leq |M_*| \leq n$, where $|M_*| = 0$ if and only if $|M| = 0$. We let $(C', C_*, T) = \mathsf{csplit}_n(C)$ (as in ciphertext split) denote splitting a string C of at least n bits into three parts $C'\|C_*\|T = C$, such that $|C_*| = n$, $|T| \equiv |C| \pmod n$, and $0 \leq |T| \leq n$, where $|T| = 0$ if and only if $|C| = n$. Finally, we let $C'_1, \dots, C'_m, C_*, T \leftarrow \mathsf{csplit\text{-}b}_n(C)$ (as in csplit to blocks) denote the result of $\mathsf{csplit}_n(C)$ followed by partitioning of C' into $m = |C'|_n$ blocks of n bits, such that $C' = C'_1\| \dots \|C'_m$.

The symbol $\perp$ denotes an error signal, or an undefined value. We denote by $X \leftarrow\$ \mathcal{X}$ sampling an element X from a finite set $\mathcal{X}$ following the uniform distribution.

3.1 Forkcipher

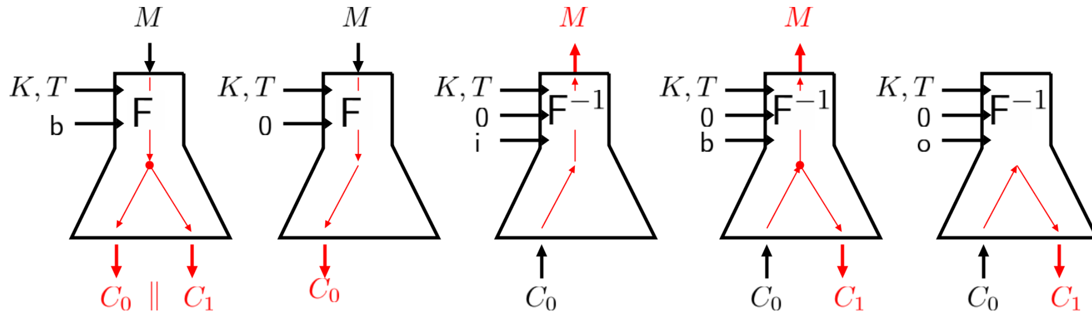


Figure 1: Illustration of the encryption and decryption by a forkcipher.

Our AEAD schemes use the low-level primitive *forkcipher* [9]. A forkcipher F is a fixed-input length and expanding fixed-output length tweakable symmetric key primitive. Similarly to a tweakable blockcipher, a forkcipher takes a secret key, a tweak and a plaintext block, but instead produces *two* output blocks simultaneously. Informally, applying a secure forkcipher is equivalent to applying a secure tweakable blockcipher to the same plaintext, with two independent keys. The forkcipher encryption algorithm $\mathsf{F}(K, T, M, s)$ works as follows:

Encryption algorithm F	
Inputs:	Outputs:
• a key K of k bits • a tweak T of τ bits • a plaintext block M of n bits • an *output selector* s	**if** $s = \mathbf{0}$: the "left" n-bit ciphertext block C_0 **if** $s = \mathbf{1}$: the "right" n-bit ciphertext block C_1 **if** $s = \mathbf{b}$: both n-bit ciphertext blocks C_0, C_1

The selector chooses whether both, or only one of the output blocks will be computed by the forkcipher. The values of the selector can be encoded in any way, as long as they are distinct.

A forkcipher applies two independent permutations to the plaintext block which makes either of the ciphertext blocks sufficient to compute the original plaintext but also to re-compute the "other" ciphertext block. The inverse algorithm F^{-1} works as follows:

Inverse algorithm F^{-1}	
Inputs:	Outputs:
• a key K of k bits • a tweak T of τ bits • a ciphertext block C of n bits • a binary "side" indicator b • an output selector s	**if** $s = \mathsf{i}$: the n-bit plaintext block M (**i**nvert only), **if** $s = \mathsf{o}$: the "**o**ther" n-bit ciphertext block $C_{b\oplus 1}$, **if** $s = \mathsf{b}$: **b**oth these n-bit blocks $M, C_{b\oplus 1}$.

The side indicator decides whether the block C is treated as the "left" ciphertext block C_0 (if $b = 0$), or the "right" ciphertext block C_1 (if $b = 1$). The algorithms of a forkcipher are illustrated in Figure 1.

Informally, a *correct* forkcipher's encryption algorithm F implements a pair of permutations of the set $\{0,1\}^n$ for every $K \in \{0,1\}^k$ and $\mathsf{T} \in \mathcal{T}$, and its decryption algorithm consistently inverts and re-applies these permutations. A formal mathematical definition of a forkcipher can be found in Appendix A.

3.2 Authenticated Encryption

The modes of operation in this proposal follow the AEAD syntax proposed by Rogaway [36]. A nonce-based AEAD scheme is a triplet $\Pi = (\mathcal{K}, \mathcal{E}, \mathcal{D})$. The key space $\mathcal{K}$ is a finite set endowed with the uniform distribution. The deterministic encryption algorithm $\mathcal{E} : \mathcal{K} \times \mathcal{N} \times \mathcal{A} \times \mathcal{M} \to \mathcal{C}$ maps a secret key K, a nonce N, an associated data A and a message M to a ciphertext $C = \mathcal{E}(K, N, A, M)$. The nonce, AD and message domains are all subsets of $\{0,1\}^*$. The deterministic decryption algorithm $\mathcal{D} : \mathcal{K} \times \mathcal{N} \times \mathcal{A} \times \mathcal{C} \to \mathcal{M} \cup \{\bot\}$ takes a tuple (K, N, A, C) and either returns a mesage $M \in \mathcal{M}$, or a distinguished symbol $\bot$ to indicate an authentication error.

We require that for every $M \in \mathcal{M}$, we have $\{0,1\}^{|M|} \subseteq \mathcal{M}$ (i.e. for any integer m, either all or no strings of length m belong to $\mathcal{M}$) and that for all $K, N, A, M \in \mathcal{K} \times \mathcal{N} \times \mathcal{A} \times \mathcal{M}$ we have $|\mathcal{E}(K, N, A, M)| = |M| + \theta$ for some non-negative integer θ called the stretch of Π. For correctness of Π, we require that for all $K, N, A, M \in \mathcal{K} \times \mathcal{N} \times \mathcal{A} \times \mathcal{M}$ we have $M = \mathcal{D}(K, N, A, \mathcal{E}(K, N, A, M))$. We let $\mathcal{E}_K(N, A, M) = \mathcal{E}(K, N, A, M)$ and $\mathcal{D}_K(N, A, M) = \mathcal{D}(K, N, A, M)$.

4 Specification

Below we give the full detailed specification of our ForkAE members as defined by their underlying primitive ForkSkinny and PAEF or SAEF mode of operations.

4.1 ForkSkinny

We propose the ForkSkinny cipher family as an underlying primitive or building block for the modes PAEF and SAEF. ForkSkinny is a slight modification of the SKINNY family [14]. Here, we detail the concrete instantiations, the specification, and transcribe the parts of SKINNY that we keep unchanged in our design.

SKINNY is a family of lightweight tweakable block ciphers that was presented at Crypto 2016 by Beierle et al. [14] with the objective to have comparable performance but stronger security guaranties than SIMON, a Feistel cipher proposed by the NSA [13]. The 6 variants described in [14] differ from block size ($n = 64$ or $n = 128$ bits) and from the tweakey size ($z \times n$ bits, where z is either 1, 2 or 3). They are denoted as SKINNY-n-zn.

In a similar way, by ForkSkinny-n-zn we denote one variant of our cipher with a block size of n bits (either 64 or 128) and of $z \times n$ tweakey bits. We further consider versions where the tweakey size is not a multiple of the block size n. In general, ForkSkinny-n-t here will denote the ForkSkinny with n bit block and t bit tweakey. ForkSkinny has an *initial* number of rounds r_{init} which accounts for the number of rounds before the *forking* step. The other two parameters r_0 and r_1 denote the number of rounds after the forking step for the left and right branch of the function, respectively. The two branches of ForkSkinny produce two ciphertexts each of length n bits. Our ForkSkinny primitive instantiations are defined by the distinct value of the latter parameters which are depicted in Table 4.

Primitive	block	tweak	tweakey	r_{init}	r_0	r_1
ForkSkinny-64-192	64	64	192	17	23	23
ForkSkinny-128-192	128	64	192	21	27	27
ForkSkinny-128-256	128	128	256	21	27	27
ForkSkinny-128-288	128	128	288	25	31	31

Table 4: The ForkSkinny primitives with their internal parameters for round numbers r_{init}, r_0 and r_1 and their corresponding external parameters of block and tweakey sizes (in bits) for fixed 128 bit keys.

The ciphers have a Substitution-Permutation-Network (SPN) structure, and as in the AES [8], the internal state is organised as a 4×4 matrix, where each cell is either a byte (when $n = 128$) or a nibble (when $n = 64$). The $n-$bit messages are loaded row-wisely in the internal state IS, as depicted below.

$$IS = \begin{bmatrix} m_0 & m_1 & m_2 & m_3 \\ m_4 & m_5 & m_6 & m_7 \\ m_8 & m_9 & m_{10} & m_{11} \\ m_{12} & m_{13} & m_{14} & m_{15} \end{bmatrix}$$

In the following, we review the most important aspects of the design of SKINNY, and refer to the original SKINNY specification [14] for more details.

In Figure 2, we use $\mathsf{Enc\text{-}Skinny}_r(\cdot,\cdot,j)$, to denote the r-round of SKINNY encryption starting with round i and $\mathsf{Dec\text{-}Skinny}_r(\cdot,\cdot,j)$, to denote r rounds of SKINNY decryption which starts with round $i+r-1$, i.e.

$$\mathsf{Enc\text{-}Skinny}_r = \mathcal{R}_{j+r-1} \circ \ldots \circ \mathcal{R}_j \quad (1)$$

$$\mathsf{Dec\text{-}Skinny}_r = \mathcal{R}_j^{-1} \circ \ldots \mathcal{R}_{j+r-1}^{-1}, \quad j \geq 0 \quad (2)$$

where $\mathcal{R}_i$ is defined in Equation (3).

The TKS and TKS_r denote the tweakey scheduling function per round and r rounds respectively and is described in Figure 3.

Round Function ForkSkinny round function (see Figure 4) only differs slightly from the SKINNY one: it reuses the 5 operations described in SKINNY, but considers different round constant in the AddConstants step to take into account the fact that more rounds are iterated.

The round function operations are the following (see Figure 4):

- SubCells (SC): each of the 16 words of the internal state is modified by a 4×4 (if $n = 64$) or 8×8 Sbox (if $n = 128$). The definition of the Sboxes is recalled in Appendix E. ForkSkinny reuses the Sboxes of SKINNY without any change.
- AddConstants (AC): A LFSR is used to produce constants that are added in the first 3 cells of the first column. Since in total ForkSkinny iterates more rounds than SKINNY, we changed the definition of the LFSR to avoid repetitions.
- AddRoundTweakey (ART): Exactly as in SKINNY, the addition of the tweakey material is done in the first two lines of the state.
- ShiftRows (SR): The second line of the internal state is right rotated by 1 cell, the third line is right rotated by 2 cells, and the last line is right rotated by 3 cells.
- MixColumns (MC): This operation modifies each column by multiplying it with a binary matrix M, given by:

$$M = \begin{pmatrix} 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 \end{pmatrix}$$

```
algo ForkSkinny-Enc(M, K, T, s)
    tk ← K||T
    L ← Enc-Skinny_{r_init}(M, tk, 0)
    if s = 1 or s = b then
        C_1 ← Enc-Skinny_{r_1}(L, TKS_{r_init}(tk), r_init)
    end if
    if s = 0 or s = b then
        tk' ← TKS_{r_init+r_1}(tk)
        C_0 ← Enc-Skinny_{r_0}(L ⊕ BC, tk', r_init)
    end if
    if s = 0 return C_0
    if s = 1 return C_1
    if s = b return C_0, C_1
end algo

algo Enc-Skinny_r(S, tk, i)
    for j ← 0 to r − 1 do
        S ← R_{i+j}(S, tk)
        tk ← TKS_{i+j}(tk)
    end for
    return S
end algo

algo Dec-Skinny_r(S, tk, i)
    for j ← 0 to r − 1 do
        tk ← TKS_{i+j}(tk)
    end for
    for j ← r − 1 to 0 do
        S ← R^{-1}_{i+j}(S, tk)
        tk ← TKS^{-1}_{i+j}(tk)
    end for
    return S
end algo

algo ForkSkinny-Dec(C, K, T, b, s)
    tk ← K||T
    tk' ← TKS_{r_init}(tk)
    if b = 1 then
        L ← Dec-Skinny_{r_1}(C, tk', r_init)
    else if b = 0 then
        tk'' ← TKS_{r_1}(tk')
        L ← Dec-Skinny_{r_0}(C_b, tk'', r_init) ⊕ BC
    end if
    if s = i or s = b then
        M ← Dec-Skinny_{r_init}(L, tk, 0)
    end if
    if s = o or s = b then
        tk' ← TKS_{r_1}(tk')
        β ← (b ⊕ 1) · r_1
        L ← b · BC ⊕ L
        C' ← Enc-Skinny_{r_{b⊕1}}(L, tk', r_init +
β)
    end if
    if s = i return M
    if s = o return C'
    if s = b return M, C'
end algo
```

Figure 2: ForkSkinny encryption and decryption algorithms. BC is a so-called *branch constant* that is explicited and justified in the following sections. The tweakey scheduling algorithms (TKS, TKS_r) is described in fig. 3

Note that all the rounds are identical, and in particular that no whitening keys are used. The ith round function can be described as

$$\mathcal{R}_i = \texttt{MixColumns} \circ \texttt{ShiftRows} \circ \texttt{ART}_i \circ \texttt{AddConstants}_i \circ \texttt{SubCells} \tag{3}$$

where the subscript i denotes the corresponding functions at iteration i. The Enc-Skinny and Dec-Skinny can be described as

$$\textsf{Enc-Skinny} = \mathcal{R}_{r-1} \circ \ldots \circ \mathcal{R}_0 \tag{4}$$

$$\textsf{Dec-Skinny} = \mathcal{R}_0^{-1} \circ \ldots \circ \mathcal{R}_{r-1}^{-1} \tag{5}$$

Round Constants Since the full forking structure requires more iterations of the round function than in SKINNY, we have changed the AddConstants step. Instead of using 6-bit round constants, we use 7-bit ones. Similarly to what is done for SKINNY, our implementation uses an affine LFSR to generate these round constants. The update function is defined as:

$$(rc_6||rc_5||\ldots||rc_0) \rightarrow (rc_5||rc_4||\ldots||rc_0||rc_6 \oplus rc_5 \oplus 1)$$

The 7 rc_i bits are initialized to 0 and updated before use in the round function. The bits from the LFSR are used exactly in the same way as in SKINNY. The 4×4 array

$$\begin{pmatrix} c_0 & 0 & 0 & 0 \\ c_1 & 0 & 0 & 0 \\ c_2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

is constructed depending on the size of the internal state, where $c_2 = \texttt{0x2}$ and

$$(c_0, c_1) = (rc_3||rc_2||rc_1||rc_0, 0||rc_6||rc_5||rc_4) \text{ when each cell is 4 bits}$$
$$(c_0, c_1) = (0||0||0||0||rc_3||rc_2||rc_1||rc_0, 0||0||0||0||0||rc_6||rc_5||rc_4) \text{ when each cell is 8 bits.}$$

Branch Constant As we are going to detail in the design rationale section (Section 8), we introduce additional constants to be added right after the forking point. Namely, when each cell is made of 4 bits we add the following state:

$$BC_4 = \begin{pmatrix} \texttt{1} & \texttt{2} & \texttt{4} & \texttt{9} \\ \texttt{3} & \texttt{6} & \texttt{d} & \texttt{a} \\ \texttt{5} & \texttt{b} & \texttt{7} & \texttt{f} \\ \texttt{e} & \texttt{c} & \texttt{8} & \texttt{1} \end{pmatrix}$$

right after forking, to the left branch leading to C_0. These constants are generated by clocking a 4-bit LFSR, given by: $(x_3||x_2||x_1||x_0) \rightarrow (x_2||x_1||x_0||x_3 \oplus x_2)$, and initialised with $x_0 = 1$, $x_1 = x_2 = x_3 = 0$.

Table 5: Constants used in ForkSkinny.

Rounds	Constants
1 - 16	01,03,07,0F,1F,3F,7E,7D,7B,77,6F,5F,3E,7C,79,73
17 - 32	67,4F,1E,3D,7A,75,6B,57,2E,5C,38,70,61,43,06,0D
33 - 48	1B,37,6E,5D,3A,74,69,53,26,4C,18,31,62,45,0A,15
49 - 64	2B,56,2C,58,30,60,41,02,05,0B,17,2F,5E,3C,78,71
65 - 80	63,47,0E,1D,3B,76,6D,5B,36,6C,59,32,64,49,12,25
81 - 87	4A,14,29,52,24,48,10

When each cell is a byte we add the following state:

$$BC_8 = \begin{pmatrix} \texttt{01} & \texttt{02} & \texttt{04} & \texttt{08} \\ \texttt{10} & \texttt{20} & \texttt{41} & \texttt{82} \\ \texttt{05} & \texttt{0a} & \texttt{14} & \texttt{28} \\ \texttt{51} & \texttt{a2} & \texttt{44} & \texttt{88} \end{pmatrix},$$

generated by the 8-bit LFSR: $(x_7||x_6||x_5||x_4||x_3||x_2||x_1||x_0) \rightarrow (x_6||x_5||x_4||x_3||x_2||x_1||x_0||x_7 \oplus x_5)$, again initialised with $x_0 = 1$ and all the other bits equal to 0.

Tweakey Again, the tweakey schedule works similarly to what is done in SKINNY, that is based on the TWEAKEY framework [25]. The first operation consists in filling the tweakey state, which is view as a collection of 4×4 matrices of the same cell-size as the considered internal state. If the cipher uses material that is not the key (that is, strictly a tweak), this one is positioned first in $TK1$, row wisely, and then is set the key (if that leaves an incomplete matrix we fill it with zeros). We denote these matrices by $TK1$, $TK2$ and $TK3$ (if any). As suggested in the SKINNY specification, when there is some tweak material, we add an extra 1 in the constant matrix from AddConstants, every round at line 0, column 2, to the second bit).

If the tweakey size is not a multiple of the state size but leaves 2 empty rows in the last tweakey matrix (as it is the case for ForkSkinny-128-192), instead of filling the remaining cells with zeros we simply don't use these cells, which allows to save some memory, some LFSR applications and also some XORs.

As can be seen on Figure 5, during the AddRoundTweakey step the first two rows of each tweakey are exclusive-ored together and then to the internal state. To update the tweakey arrays for the next round, each tweakey word is first modified by a cell-permutation P_T, given by:

$$P_T = [9, 15, 8, 13, 10, 14, 12, 11, 0, 1, 2, 3, 4, 5, 6, 7]$$

and which effect on the cell positioning is as follows:

$$\begin{bmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \\ 12 & 13 & 14 & 15 \end{bmatrix} \xrightarrow{P_T} \begin{bmatrix} 9 & 15 & 8 & 13 \\ 10 & 14 & 12 & 11 \\ 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \end{bmatrix}$$

Each cell (except the ones in TK1) is then linearly modified by a LFSR, following the definitions given in Table 6.

Table 6: LFSR used to update $TK2$ and $TK3$.

TK	cell size	LFSR
TK2	4	$(x_3\|\|x_2\|\|x_1\|\|x_0) \rightarrow (x_2\|\|x_1\|\|x_0\|\|x_3 \oplus x_2)$
	8	$(x_7\|\|x_6\|\|x_5\|\|x_4\|\|x_3\|\|x_2\|\|x_1\|\|x_0) \rightarrow (x_6\|\|x_5\|\|x_4\|\|x_3\|\|x_2\|\|x_1\|\|x_0\|\|x_7 \oplus x_5)$
TK3	4	$(x_3\|\|x_2\|\|x_1\|\|x_0) \rightarrow (x_0 \oplus x_3\|\|x_3\|\|x_2\|\|x_1)$
	8	$(x_7\|\|x_6\|\|x_5\|\|x_4\|\|x_3\|\|x_2\|\|x_1\|\|x_0) \rightarrow (x_0 \oplus x_6\|\|x_7\|\|x_6\|\|x_5\|\|x_4\|\|x_3\|\|x_2\|\|x_1)$

ForkSkinny-64-192 This member of ForkSkinny has block size $n = 64$ and tweakey size $t = 3n$ bits. The 192 bit tweakey contains 64 bit tweak and the rest are key bits.

ForkSkinny-128-192 This has block size $n = 128$ and tweakey size $t = 3n/2$ bits. The 192 bit tweakey contains 64 bit tweak and the rest are key bits. Note that the design of SKINNY allows to use tweakey such that $n < t < 2n$. In such cases, the $2n - t$ bits of the tweakey are set to 0.

ForkSkinny-128-256 For this version of ForkSkinny we use $n = 128$ with tweakey size $t = 2n$. The 256 bit tweakey contains 128 bit tweak and the rest are key bits.

ForkSkinny-128-288 For this version of ForkSkinny we use $n = 128$, with tweakey size $t = 9n/4$. The 288 bit tweakey contains 160 bit tweak and the rest are key bits. Note that the design of SKINNY allows to use tweakey such that $2n < t < 3n$. In such cases, SKINNY proposal recommends to set the $3n - t$ bits of the tweakey to 0.

Remark. SKINNY has a security margin of 30% for 64-bit block size and 50% for 128-bit block size. The existing security analyses of SKINNY, such as algebraic, differential/linear, and integral attacks, can be applied directly to ForkSkinny. Combining these results together with our security analysis of ForkSkinny, we believe that a more optimized version of ForkSkinny-128-192, ForkSkinny-128-256 and ForkSkinny-128-288 can be obtained by reducing their r_{init} parameter by 6 rounds while still maintaining the same security levels.

4.2 Parallel AEAD from a Forkcipher

PAEF is a forkcipher-mode of operation for nonce-based AEAD. It is parameterized by

- a forkcipher F (with key length k, tweak length τ and block length n),
- a nonce length ν such that $0 < \nu \leq \tau - 4$.

The following characteristics of PAEF$[\mathsf{F}, \nu]$ are determined by the parameters:

Key length	k bits
Maximal #of encryption queries	2^{ν}
Maximal length of a message	$2^{(\tau-\nu-3)} \cdot n/3$ bytes
Maximal length of AD	$2^{(\tau-\nu-3)} \cdot n/3$ bytes
Ciphertext expansion	n bits

The nonce-length ν is a parameter that allows to make a trade-off between the maximal message length and maximal number of queries with the same key. The encryption algorithm of PAEF is illustrated in Figure 7 and both the encryption and decryption algorithms are specified in Figure 6.

Encryption of PAEF. In an encryption query, AD and message are partitioned into blocks of n bits. Each block is processed with a single call to F using a tweak that is composed of: (1) the nonce N; (2) a three-bit flag $f_0 \| f_1 \| f_2$; (3) a $(\tau - \nu - 3)$-bit encoding of the block index i (unique for both AD and message). The flag bits are set as follows:

$\boldsymbol{f_0}$ distinguishes input: 0 for AD and 1 for message,

$\boldsymbol{f_1}$ captures block completeness: 0 for n-bit blocks and 1 for shorter blocks,

$\boldsymbol{f_2}$ marks end of input: 0 for non-final block of an input and 1 for the final block.

The ciphertext blocks are the “left” output blocks of F computed from message blocks, and a xor-sum of all-but-last “right” output blocks of F is xored to the “left” output block of the call to F that processes the final message block. The final “right” block is truncated according to the length of the final message block.

Decryption of PAEF. The decryption proceeds similarly as the encryption, except that the plaintext blocks and “right” output blocks are computed from ciphertext blocks (using the inverse algorithm F^{-1}), the xor sum is recomputed and xored to the same ciphertext block as in the encryption query. Then, the decryption of the final block and the integrity check are performed. When the final message block is complete, the check is done solely by comparing the tag. If the final message block is $m < n$ bits long, then the integrity check is done with m bits of the tag, and verifying that the inverse of the (unmasked) final ciphertext block has correct padding in final $n - m$ bits.

4.3 Sequential AEAD from a Forkcipher

SAEF is a mode of operation of a forkcipher for nonce-based AEAD. It is parameterized by

- a forkcipher F (with key length k, tweak length τ and block length n),
- a nonce length ν such that $0 < \nu \leq \tau - 4$.

The following characteristics of SAEF$[\mathsf{F}, \nu]$ are determined by the parameters:

Key length	k bits
Maximal #of encryption queries	2^ν
Maximal length of a message	not limited by the algorithm
Maximal length of AD	not limited by the algorithm
Ciphertext expansion	n bits

The encryption algorithm of SAEF is illustrated in Figure 9 and both the encryption and decryption algorithms are specified in Figure 8.

Encryption of SAEF. In an encryption query, first AD and then message are processed in blocks of n bits. Each block is processed with a single call to F, using a tweak that is composed of: (1) the nonce followed by a 1-bit in the initial F call, and the string $0^{\tau-3}$ otherwise, (2) three-bit flag f. The binary flag f takes different values for processing of different types of blocks in the encryption algorithm:

value of f	type of block processed by the encryption algorithm
000	processing non-final AD block
010	processing final complete AD block
011	processing final incomplete AD block
110	processing final complete AD block to produce tag
111	processing final incomplete AD block to produce tag
001	processing non-final message block
100	processing final complete message block
101	processing final incomplete message block

The right-hand output of every F call is used as a whitening mask for the following F call, masking either the input (in AD processing) or both the input and the output (in message processing) of this subsequent call. The initial F call in the query is unmasked. The "right" output block of the final call is concatenated to the ciphertext and truncated according to the length of the final message block.

Decryption of SAEF. The decryption proceeds similarly to the encryption, except that the plaintext blocks and the right-hand output blocks in the message processing part are computed with the inverse F^{-1} algorithm. Along with the decryption of the final

block, the integrity check is performed. When the final message block is complete, the check is done solely by comparing the tag. If the final message block is $m < n$ bits long, then the integrity check is done with m bits of the tag, and verifying that the inverse of the (unmasked) final ciphertext block has correct padding in final $n - m$ bits.

5 Security Claims

In this section we specify the security levels of all ForkAE members in the single key model (SK). In Table 7 we derive the bit level authenticated encryption (AE) security comprising of *confidentiality of plaintexts* and*integrity of ciphertexts, associated data, and nonce* and express it in $\log_2$ of number of ForkSkinny evaluations. Throughout this section we denote the AE security by $\mathsf{Sec}^{\mathrm{ae}}_{\mathrm{Mode}}$ (for Mode$\in \{\mathrm{PAEF}, \mathrm{SAEF}\}$) and the bit level security against key recovery of F (ForkSkinny) by $\mathsf{Sec}_{\mathsf{F}}$. Our AE security claims are supported by security proofs [9] and show full n-bit security for ForkSkinny in PAEF and birthday bound $n/2$-bit security on the block size n for SAEF. These security results are applicable under the provision that the nonce N is a unique value for every distinct message processed with the fixed key K. We conjecture that the bit level security against key recovery $\mathsf{Sec}_{\mathsf{F}}$ is at least 112 bits for our choice of key size $k = 128$ (for all instances). Our conjectured security is supported by extensive cryptanalysis in Section 6.

Mode-ForkSkinny	$\mathsf{Sec}^{\mathrm{ae}}$	$\mathsf{Sec}_{\mathsf{F}}$
PAEF-ForkSkinny	n	112
SAEF-ForkSkinny	$n/2$	112

Table 7: ForkAE general security claims.

5.1 NIST security requirement

Our primary recommendation is PAEF-ForkSkinny-128-288. According to the *main security requirement* of NIST, for *all* family members *"cryptanalytic attacks on the AEAD algorithm shall require at least* 2^{112} *computations on a classical computer in a single-key setting"*. NIST also mandates that the primary recommendation additionally comes with *"a nonce of at least* 96 *bits"* and *"limits on the input sizes (plaintext, associated data, and the amount of data that can be processed under one key) for this member shall not be smaller than* $2^{50} - 1$ *bytes."*. Both PAEF-ForkSkinny-128-288 and SAEF-ForkSkinny-128-256 comply with all of the above NIST requirements. Our two instances also support secure processing of $2^{50} - 1$ bytes in a *single query*.

In Table 8 we define the AE security $\mathsf{Sec}^{\mathrm{ae}}_{\mathrm{Mode}}$ for the PAEF-ForkSkinny-128-288 and SAEF-ForkSkinny-128-256 instances, respectively.

Mode	Primitive	$\mathsf{Sec}^{\mathrm{ae}}_{\mathrm{Mode}}$	$\mathsf{Sec}_{\mathsf{F}}$	Max Data	Nonce
PAEF	**ForkSkinny-128-288**	**128**	**112**	$\mathbf{2^{112+4}}$	**104**
SAEF	ForkSkinny-128-256	64	112	2^{64+4}	120

Table 8: PAEF-ForkSkinny-128-288 and SAEF-ForkSkinny-128-256 NIST compliant security claims for nonce N of at least 96 bits.

The max data (Column 5) is a measure of the total data complexity determined as the number of *bytes* of AD, messages, and ciphertexts that can be securely evaluated (before rekeying), and is determined as

$$\min\{2^4 \cdot 2^{\mathsf{Sec}_{\mathsf{F}}}, 2^4 \cdot 2^{\mathsf{Sec}^{\mathrm{ae}}_{\mathrm{Mode}}}, 2^{\nu} \cdot \text{bytes/message}\},$$

where the first two terms are upper bounds on data complexity in *bytes* (hence multiplying by the number of bytes per block $2^{(\log_2(n)-3)}$) w.r.t. the security analysis or the modes and ForkSkinny, respectively, and the third term is an operational limit (the total number of bytes processed in AEAD evaluations/calls with *all* possible nonce (of size ν) values, each with maximal number of bytes in a single message determined by Column 5 in Table 3).

5.2 Security for our primary and targeted use cases.

Our ForkAE family targets applications dealing primarily with **short messages**. All our members comply with the main security requirements of NIST as indicated in Table 7. Below in Table 9, we outline and emphasize that ForkAE instances achieve high security levels and max data processing abilities for our main target use case when predominantly data of up to 4 **blocks** is processed.

Mode	ForkSkinny-n-t	$\mathsf{Sec}^{\mathrm{ae}}_{\mathrm{Mode}}$	$\mathsf{Sec}_{\mathsf{F}}$	Max data	Nonce
PAEF	64-192	64	112	2^{48+5} bytes	48
	128-192	128	112	2^{48+6} bytes	48
	128-256	128	112	2^{112+4} bytes	112
	128-288	128	112	2^{104+6} bytes	104
SAEF	128-192	64	112	2^{56+6} bytes	56
	128-256	64	112	2^{64+4} bytes	120

Table 9: ForkAE security claims for short messages of up to 4 blocks.

The max data (Column 5) here is the maximal data complexity as the total number of *bytes* of AD, messages and ciphertexts that can be securely (before rekeying) evaluated by the AEAD algorithm if all messages have at most 4 blocks, determined as

$$\min\{2^{(\log_2(n)-3)} \cdot 2^{\mathsf{Sec}_{\mathsf{F}}}, 2^{(\log_2(n)-3)} \cdot 2^{\mathsf{Sec}^{\mathrm{ae}}_{\mathrm{Mode}}}, 2^{\nu} \cdot 2^{2+(\log_2(n)-3)}\},$$

where the first two terms are upper bounds on data complexity in *bytes* (hence multiplying by the number of bytes per block $2^{(\log_2(n)-3)}$) w.r.t. the security analysis or the modes and ForkSkinny, respectively, and the third term is an operational limit of the setting (the total number of bytes processed in messages with all possible nonce values, each of 4 blocks of $2^{(\log_2(n)-3)}$ bits).

6 Security Analysis

The design of the AEAS schemes proposed in this document allows for a modular security analysis: the modes of operation can be analyzed separately from the primitive (ForkSkinny), and then be easily combined. More precisely, we:

- Analyze the security of ForkSkinny to the state-of-the-art cryptanalysis techniques for tweakable block ciphers. We additionally analyze the resistance of ForkSkinny against adaptations of these attacks that exploit the structure of a forkcipher.
- Prove in the most widely used security model for AEAD that when instantiated with any secure forkcipher, PAEF and SAEF are respectively optimally and birthday secure. Informally, our provable analysis implies that when the modes are used correctly, attacking the resulting AEAD scheme will have little-or-no more success than an attack on ForkSkinny.

6.1 Cryptanalysis of ForkSkinny

The security arguments provided in the SKINNY specification [14] – as for instance the bounds on the number of active Sboxes – directly transfer to ForkSkinny in the setting where an attacker has access to only the plaintext M and to C_1 (such access translates to breaking $r_{\mathsf{init}} + r_1$-round SKINNY or full SKINNY for our choice of the parameters). In that case, the security arguments devised in [14] and in the numerous third-party analyses that followed (see for instance [10, 30, 37, 40, 42]) support the choice of our parameters. In a similar vein, finding an attack when only M and C_0 are known to the attacker is equivalent to breaking $r_{\mathsf{init}} + r_0$-round SKINNY with a slightly modified tweakey schedule.

A different type of analysis is required for what we call *reconstruction* type of attacks due to the specific forking structure of ForkSkinny. Such attacks correspond to a setting where the attacker has access to both C_0 and C_1, and has the freedom to obtain the value of C_1 corresponding to a chosen C_0 (or vice versa). In comparison to the previous scenarios (where M and either of C_1 or C_0 are provided), the most notable change comes from the fact that C_0 is related to C_1 by a series of decryption rounds followed by an equally long sequence of encryption rounds. In the reconstruction attacks analysis, we focus on the central rounds, when the operations switch from decryption to encryption. As shown in the study [12] of ForkAES with $r_{\mathsf{init}} = r_0 = r_1 = 5$, distinguishers may be possible when there is insufficient diffusion in the middle rounds.

Our analysis, detailed in Appendix D, showed that the cryptanalytic properties around the forking point could help build a distinguisher for more rounds than in only encryption or decryption. However, this effect is rather limited (to less than 5 rounds).

It is customary to evaluate any new symmetric-key function against known cryptanalysis techniques to demonstrate the resilience of the function. In addition to the cryptanalysis techniques which can exploit the special structure of ForkSkinny, in Appendix D we have evaluated ForkSkinny using state-of-the-art cryptanalysis techniques such as *integral attack*, *meet-in-the-middle attack*, *impossible differential attack*, and *algebraic attack*. We have considered both single key and related-tweakey cryptanalysis to evaluate the security of ForkSkinny. The detailed analysis in Appendix D and the conservative security margins of SKINNY give us confidence that the chosen ForkSkinny parameters provide the claimed security for ForkSkinny. Note that for ForkAE it is not essential to have related-key security of ForkSkinny and we do not claim any RK security of ForkSkinny in this article.

6.2 Modes

Based on the results of cryptanalysis, we assume ForkSkinny to be a pseudo-random tweakable forked permutation or indistinguishable from a pair of tweaked permutations drawn uniformly at random. We show that under this assumption (ForkSkinny is an n-bit pseudo-random tweakable forked permutation PRTFP), PAEF instantiated with ForkSkinny cannot be distinguished from an ideal authenticated encryption scheme in up to about 2^n ForkSkinny calls, and SAEF cannot be distinguished from an ideal authenticated encryption scheme in up to about $2^{n/2}$ ForkSkinny calls [9] . That is, we achieve confidentiality against chosen-plaintext (CPA) attacks and integrity against forgery up to approximately 2^n ForkSkinny calls for PAEF and $2^{n/2}$ ForkSkinny calls for SAEF. This result means that under the same key the total amount of associated data and plaintext/ciphertext should not exceed 2^n and $2^{n/2}$ blocks for PAEF and SAEF, respectively.

In more detail, we prove the security of PAEF in the standard model under the PRTFP assumption on ForkSkinny against an AE distinguisher up to the bound $\frac{q_v \cdot 2^n}{(2^n-1)^2}$, where q_v is the number of allowed decryption queries with authentication error for PAEF [9]. For SAEF we similarly prove AE security up to the bound $2 \cdot \frac{(\sigma-q+1)^2}{2^n} + \frac{\sigma(\sigma-q)}{2^n} + \frac{q_v(q+2)}{2^n}$, where q is the number of encryption queries, q_v is the number of allowed decryption queries with authentication error, and σ denotes the maximal total data complexity measured in the total number of blocks of AD, messages, and ciphertexts in all adversarial queries.

We clarify that our security model does not encompass timing and power consumption attacks.

7 Efficiency/Implementation

SKINNY: EFFICIENT AND FLEXIBLE BUILDING BLOCK. Our designs are based on the SKINNY tweakable block cipher, whose efficiency and versatility is well documented in the existing literature where it was compared to other (lightweight) block ciphers [14], such

as PRESENT, SIMON, and the block cipher AES. Suitable SKINNY implementation strategies range from 1-bit serial and word-based implementations, which yield a very small hardware footprint, to round-based and fully unrolled implementations. More specifically, the following highly attractive features should be highlighted:

- SKINNY excels in *round-based* implementations, which constitute a common implementation strategy due to their attractive latency-area characteristics. When implemented in this fashion, SKINNY outperforms the SIMON contribution [14], which has impressive performance figures.
- When implementation size or power consumption are of concern, serialized SKINNY implementations are among the smallest in the literature, while still offering good throughput.
- The SKINNY rounds can be efficiently fully or partially unrolled in hardware, resulting in low-latency/high-throughput characteristics with a comparatively small area footprint.
- When deployed in a potentially hostile environment, cryptographic devices are susceptible to *side-channel analysis* attacks. Fortunately, introducing standard side-channel countermeasures is very cheap for SKINNY compared to other primitives.
- Software implementations in microcontrollers perform very well too; SKINNY has excellent performance on *embedded microcontrollers*, which represent the low-end of computing devices and are widely deployed in the wild [14].

A primitive that is as well-rounded as SKINNY is very advantageous for our purposes: it satisfies a wide range of implementation constraints, and covers the needs of a high number of lightweight cryptography use cases. In addition, adaptability to different sets of constraints can also be beneficial within a single application of the IoT domain, where heterogeneous devices with varying optimization goals will routinely have to be interpretable.

TAILORED FOR SHORT AE QUERIES. The ForkSkinny primitive is purposely designed as a simple forked execution of the SKINNY tweakable block cipher. As a result, ForkSkinny directly inherits all the attractive implementation properties and trade-offs of SKINNY. ForkSkinny additionally adapts to short AE queries, achieving lower latency with less area because the whole ciphertext is produced immediately, within a single primitive call. Other AE modes fail to achieve this because either they contain a sequential component (e.g. CCM), or face a significant area cost (e.g. OCB). ForkSkinny hardware implementations can achieve authenticated encryption of very short messages in *a single clock cycle*.

FORKSKINNY IS VERSATILE. In **performance-oriented implementations**, the post-fork execution benefits from parallelism, essentially allowing to execute a forkcipher call with *zero latency overhead* compared to a block cipher call. In **resource-optimized implementations**, the forkcipher can be implemented *without area overhead* compared

to a block cipher call; the forking state can be recomputed instead of stored, significantly reducing the state size and allowing very compact implementations.

Application potential. In light of the previous discussion, ForkSkinny will be an excellent fit for a wide range of lightweight and IoT applications where short messages prevail, e.g. automotive and industrial controllers (**low latency budget**); medical implants and wireless sensor nodes (**low power and low energy requirements**, achieved by compact yet fast implementations); single use IoT objects (such as smart medication dispensers) and RFID tags (cheap and disposable, hence extremely **area-constrained**).

8 Design Rationale

Highly efficient encryption and authentication of *short messages* has been identified as an essential requirement for enabling security in constrained computation and communication scenarios such as the CAN FD in automotive systems (with *maximum* message length of 64 bytes or 4 blocks of 128 bits), massive IoT and critical communication domains of 5G, and Narrowband IoT (NB-IoT), or low energy protocols, such as Bluetooth with a maximum packet size length of 47 bytes. We believe this type of usage will continue to grow and modern lightweight AEAD schemes should optimally support it. This is the design philosophy that underlies the ForkAE family.

The designs of both the ForkSkinny primitive family and of the PAEF and SAEF modes of operation are primarily driven by this main optimization goal: optimal efficiency when predominantly processing very short inputs (up to 4 blocks). In the following we explain the design decisions made in the primitive level, modes of operations and different parameters used for instantiation of proposed schemes.

8.1 Design Decisions in the Primitive Level

Our underlying building block is a tweakable primitive built upon a tweakable block cipher. Tweakable block ciphers significantly improve and aid simpler security and design of cryptographic schemes (built upon them) but require careful analysis on the level of the tweakable block cipher primitive as they open to a class of so-called related-tweakey attacks. The TWEAKEY framework [25] analyzes secure ways to instantiate tweakable block ciphers and is our choice for constructing a secure tweakable forkcipher[1] together with the iterate-fork-iterate forkcipher design framework of [9]. We use SKINNY [14], a recently proposed family of tweakable block ciphers, as our starting primitive and transform it to the ForkSkinny family of forkciphers.

The SKINNY choice. Our choice is based on the following observations:

- SKINNY has comfortable security margins in view of the current best attacks: despite the fact that many cryptanalysts tried to break it, none of the published results pose

[1]The tweakey framework is used in many ciphers such as Deoxys-BC [22, 26], Joltic-BC [23], Kiasu-BC [24], Skinny and Mantis [14].

a serious threat to its security. The fraction of rounds of SKINNY-64 that can be broken is below 70%, while for SKINNY-128 this fraction drops below 50% (this can be observed from the review of the existing attacks done in Appendix D). As we detail in this document, many parts of the security analysis of SKINNY directly transfer to ForkSkinny.

- SKINNY excels at throughput per area in hardware with a round-based implementation.
- Offers possibilities for many trade-offs in the speed-resources design space, both in hardware and software.
- SKINNY is proven to be faster than other algorithms such as SIMON on various embedded systems [14].
- Protection against side-channel analysis which can be added efficiently to ForkSkinny owing to the decomposability of the SBox. Our forkcipher construction introduces no additional problems; standard side-channel countermeasures like threshold implementations [34] apply equally well to it.

All of these SKINNY advantages carry over to ForkSkinny.

DESIGN DECISIONS IN FORKSKINNY. In the following we detail the design decisions made in the construction of the ForkSkinny family of forkciphers.

Number of rounds. The number of rounds used before and after the forking step depends on the desired security of the forkcipher. Let us denote by r_{init} the number of rounds before forking, and by r_0 and r_1 the number of rounds leading to C_0 and C_1 after forking, respectively. Since we can imagine that an attacker has access to both outputs of the construction (C_0 and C_1) we must consider 3 cases: attacks that target the input message and one of the ciphertext, reconstruction attacks that target the relation between C_0 and C_1 and finally, attacks that try to benefit from the knowledge of all the input and of the two outputs together. Naturally, we aim for the same security levels for attacks targeting either M and C_0 or M and C_1, which is why we fix $r_0 = r_1$. To facilitate security analysis the parameters are also chosen so that $r_{\mathsf{init}} + r_0$ is at least equal to the number of rounds of the corresponding SKINNY cipher. Since the security margins of SKINNY are proved sufficient after numerous cryptanalysis results [10,14,30,37,40,42], we are confident that the desired security is met.

Intuitively, one expects that in the reconstruction scenario security is ensured by choosing r_0+r_1 equal to the number of rounds of the corresponding version of SKINNY. However, as shown in the cryptanalysis section, certain cryptanalytic properties of the cipher (for instance the diffusion) are weaker at the forking point than for other choices for sequences of rounds. Because of this effect, we decided not to fix $r_{\mathsf{init}} = r_0 = r_1$ but to set r_0 and r_1 larger than r_{init}. For all these reasons we fix the number of rounds as detailed in Section 4.

Forking step. An important parameter to consider in the forkcipher construction is the position of the forking step, which is determined by the parameter r_{init}. It must be chosen such that the reconstruction is secure, so we fix its value based on the cryptanalysis we perform in the reconstruction scenario. The position of the forking step also has an effect on the efficiency of ForkSkinny. If $r_{\mathsf{init}} + r_0$ is fixed, choosing a value of r_{init} such that $r_{\mathsf{init}} > r_0 = r_1$ makes the forkcipher more efficient. On the other hand, choosing $r_{\mathsf{init}} > r_0 = r_1$ can make the forkcipher insecure if $r_0 = r_1$ is too small. Indeed, an attack such as a related tweakey boomerang could possibly be found by exploiting the existence of high probability differential on the reduced r_0 rounds of SKINNY.

Round constants. As ForkSkinny iterates more rounds than SKINNY and in order to avoid the repetition of round constants, we made a simple tweak of the round constant generation of SKINNY. Taking the required number of iterations into account we decided to use 7-bit round constants generated by an LFSR (original SKINNY uses a 6-bit LFSR).

Branch constants. If we make a straightforward implementation of the forkcipher structure with the cipher SKINNY, we run into a problem that comes from the order in which the round operations are processed. More precisely, the fact that the round tweakey is added after SubCells (and not before as it is done in the AES for instance) implies that in the reconstruction scenario two SubCells operations cancel each others.

Indeed, the series of operations linking C_1 to C_0 (we have something similar for the link between C_0 and C_1) around the forking point would be:

$$MC \circ SR \circ ART_{r_{\mathsf{init}}+r_1+1} \circ AC_{r_{\mathsf{init}}+r_1+1} \circ SC \circ SC^{-1} \circ AC^{-1}_{r_{\mathsf{init}}+1} \circ ART^{-1}_{r_{\mathsf{init}}+1} \circ SR^{-1} \circ MC^{-1}.$$

Since the two non-linear operations SC cancel out in the middle, this would imply that these two rounds would boil down to only linear operations. Furthermore, since the constant and tweakey additions can switch position with the other linear operations up to a linear modification of their value, the two rounds can be simplified further and seen as simply the addition of a value that depends on the constants and tweakeys of these two rounds.

To avoid this cancellation, we decided to add a so-called branch constant (denoted BC) right after forking, on the branch leading to C_0. Each of the 16 cells of the internal state is modified. Note that the constants can either be stored or generated on the fly by running the LFSRs, allowing for more implementation trade-offs.

Additional round tweakey. The two branches in the ForkSkinny together require more round tweakeys than SKINNY. In order to produce these values we simply iterate further the SKINNY tweakey schedule. We make this design decision to benefit from the hardware-friendly SKINNY lightweight tweakey schedule.

8.2 Design Decisions in the Modes of Operation

The designs of both PAEF and SAEF are optimized the processing of short messages based on the following principles.

Lowest possible cost for shortest messages. Both PAEF and SAEF are designed so that complete data processing in AEAD encryption/decryption requires only a single call to the forkcipher per single block of data. To achieve a rate of 1 (1 forkcipher call per 1 data block) processing the nonce needs be integrated exclusively in the tweak inputs of the forkcipher.

The average performance of our instances is about 1 SKINNY equivalent evaluation per block of AD, and approximately 1.6 SKINNY evaluations per block of message. Naturally, the factor of 1.6 becomes restrictive for processing predominantly long plaintexts, but for *our main target category of use cases where inputs are short*, e.g. data (AD and messages) up to 4 blocks, our instances beat in performance the schemes obtained by applying the standard GCM [20,32], CCM [44] and OCB [29] modes instantiated with the block cipher SKINNY (matched to our ForkSkinny block and tweakey sizes).

Simplicity. Both PAEF and SAEF use just a single primitive, the forkcipher, as their main building block. All remaining operations are simple, and do not add substantial overhead.

Domain separation flags. Both PAEF and SAEF require constant binary flags to be included in each tweak. The security of the modes requires that these flags take distinct values in certain stages of processing. For the purposes of the specification we fix the particular assignment of values used in the two modes, but a change of that assignment will not have any particular impact on the security.

Parallelizability and optimal security. PAEF is designed to be fully parallel, allowing for even more efficient HW and SW implementations in when there are available resources (area resp. multiple computation cores). The parallel nature of PAEF requires the nonce and a counter to be included in every tweak, which while increasing the footprint of the implementation, allows to achieve optimal full n-bit security.

On-the-fly processing. PAEF and SAEF process data blocks “on-the-fly” as they arrive (with no delay) in encryption. PAEF processes data “on-the-fly” also in decryption.

Low overhead implementations. SAEF was designed to reduce the memory footprint of the implementation: the nonce is only processed in the first forkcipher call, and the chaining structure allows to dispose of the block counters as well. The same SAEF structure achieves birthday bound $n/2$-bit security which is typical for the majority of the existing standard AEAD modes, such as GCM, CCM and OCB.

Alignment of tweak components. In a hardware implementation, the organisation of the tweak components does not impact the performance, but in software it does. The tweak is organized in a software-friendly way. In particular, the byte-string nonce is left-aligned and the block counter (if any) is right-aligned. As such, for the majority of the counter updates in PAEF, there is no interference with the binary flags.

8.3 Distinct Instances, Advantages and Limitations

Our 6 ForkAE members come each with specific advantages and a range of security and performance tradeoffs, as schematically represented in Table 10.

Mode	ForkSkinny-n-t	parallel	data	short N	short T	latency	resources
PAEF	64-192	✓	tiny	✓	✓	lowest	+
	128-192	✓	short	✓	✗	low	++
	128-256	✓	short	✗	✗	low	++
	128-288	✓	short	✗	✗	highest	+++
SAEF	128-192	✗	short	✓	✗	low	+
	128-256	✗	short	✗	✗	low	++

Table 10: Lightweight advantages of our ForkAE members. The entries are indicative of the *relative* (to each other) characteristics of each member. Here tiny data refers ≤ 64 bits sizes of message and associated data and short to $\leq x \times 128$ for $x = 1$ to 4.

As an important efficiency metric for lightweight applications, the *resources* column denotes resource utilization for both software and hardware implementations. In software this is reflected by ROM and RAM size, whereas in hardware it captures implementation area (which has a large influence on the power consumption). On the one hand, the resource utilization is influenced by the block size and tweak size of each family member. On the other hand, it is influenced by the mode. Comparing with PAEF, SAEF comes with the advantage that the resource utilization is lower. Indeed, the nonce can be discarded after the first use, and there is no block counter that needs to be stored.

We define *latency* as the time between taking an input block and producing the corresponding output block. A small block size and tweakey size contribute positively to the latency. Given that the design target is efficiency for short messages, latency is a very important performance metric in the context of this proposal. Indeed, as the message size shrinks, the set-up time of a deep pipeline or a bitsliced implementation can no longer be amortized.

While the latency as we define it is independent of the mode, the overall *throughput* depends on whether the mode is parallel (as indicated in the *parallel* Column). Note that a fast implementation is not only desirable by itself, but also contributes to a low energy footprint of the device it will be embedded in. While SAEF only exhibits parallelism at the level of the forkcipher primitive, PAEF can additionally be parallellised and pipelined

at the level of the mode. As a result, PAEF is always potentially faster than SAEF, except for single-block messages, where both are equal (cf. the earlier latency argument).

Members of SAEF can be recommended for constrained applications where minimizing the footprint is critical (as there is no nonce or counters in the state) and settings where predominantly short messages (up to a few blocks) occur with an overwhelming probability. Regarding AE security, SAEF offers birthday type of security, whereas PAEF achieves full security of n bits (cf. Table 7).

Nonce Lengths. For PAEF, the choice of the nonce length sets an upper limit on both the number of AEAD evaluations that can be securely made with a single key, and the amount of data in a individual evaluation/call. The choices for the instances were made such that:

- the nonce is a byte string
- the maximal byte length of a message (or AD) that can be processed in a single encryption call shall be lest 1 kB (to allow for occasional large messages) for each instance except the primary recommendation,
- the maximal byte length of a message (or AD) that can be processed in a single encryption call shall be lest 2^{50} bytes for the primary recommendation.

For SAEF the choice of the nonce length was simple: the longest byte string that leaves the final 4 bits of tweak for the binary flags.

As can be seen in Table 3, this leaves long-enough nonces for sufficiently many encryption queries with a single key if the nonce is used in a stateful manner (e.g. counter per message), and for three instances also supports the use of random nonces.

Limitations. Below we discuss some of the limitations of our design.

- As ForkAE is particularly optimized for short message sizes, our ForkAE members come at a performance disadvantage when *the majority* of messages are long.
- Our algorithms give provable security guarantees when the nonces are unique values. We clarify that we purposefully forgot the strong nonce misuse resistance, due to the unavoidable doubling of computational cost it incurs for the shortest queries (due to two-pass processing).
- At present our ForkAE members do not support parametrizable tag sizes. This is a technical artifact of keeping the ciphertext expansion *constant* while at the same time processing single block queries with a single primitive call.
- As the `SKINNY` encryption and decryption process are not exactly the same, the forkcipher primitive has a non-negligible overhead with respect to a block cipher encryption-only module.

- The forkcipher primitive requires the internal cipher state at the fork to be available for both branches. Storing this state does introduce some overhead with respect to the traditional block cipher primitive. However, for many implementation strategies and use cases, this overhead is small enough and is clearly outweighed by the increased performance. Furthermore, for use cases where low resource utilization is essential, the forking state can be recomputed from the output of the C_1 branch instead of being stored, eliminating the storage overhead altogether.

References

[1] 3GPP TS 22.261: Service requirements for next generation new services and markets. https://portal.3gpp.org/desktopmodules/Specifications/SpecificationDetails.aspx?specificationId=3107.

[2] 3GPP TS 36.213: Evolved Universal Terrestrial Radio Access (E-UTRA); Physical layer procedures. https://portal.3gpp.org/desktopmodules/Specifications/SpecificationDetails.aspx?specificationId=2427.

[3] CAN FD Standards and Recommendations. https://www.can-cia.org/news/cia-in-action/view/can-fd-standards-and-recommendations/2016/9/30/.

[4] ISO 11898-1:2015: Road vehicles – Controller area network (CAN) – Part 1: Data link layer and physical signalling. https://www.iso.org/standard/63648.html.

[5] NB-IoT: Enabling New Business Opportunities. http://www.huawei.com/minisite/iot/img/nb_iot_whitepaper_en.pdf.

[6] Specification of Secure Onboard Communication. https://www.autosar.org/fileadmin/user_upload/standards/classic/4-3/AUTOSAR_SWS_SecureOnboardCommunication.pdf.

[7] Ahmed Abdelkhalek, Yu Sasaki, Yosuke Todo, Mohamed Tolba, and Amr M. Youssef. MILP modeling for (large) s-boxes to optimize probability of differential characteristics. *IACR Trans. Symm. Cryptol.*, 2017(4):99–129, 2017.

[8] Advanced Encryption Standard (AES). National Institute of Standards and Technology (NIST), FIPS PUB 197, U.S. Department of Commerce, November 2001.

[9] Elena Andreeva, Reza Reyhanitabar, Kerem Varici, and Damian Vizár. Forking a blockcipher for authenticated encryption of very short messages. Cryptology ePrint Archive, Report 2018/916, 2018. https://eprint.iacr.org/2018/916.

[10] Ralph Ankele, Subhadeep Banik, Avik Chakraborti, Eik List, Florian Mendel, Siang Meng Sim, and Gaoli Wang. Related-key impossible-differential attack on

reduced-round skinny. In Dieter Gollmann, Atsuko Miyaji, and Hiroaki Kikuchi, editors, *ACNS 17*, volume 10355 of *LNCS*, pages 208–228. Springer, Heidelberg, July 2017.

[11] Ralph Ankele and Stefan Kölbl. Mind the gap - A closer look at the security of block ciphers against differential cryptanalysis. In Carlos Cid and Michael J. Jacobson Jr:, editors, *SAC 2018*, volume 11349 of *LNCS*, pages 163–190. Springer, Heidelberg, August 2019.

[12] Subhadeep Banik, Jannis Bossert, Amit Jana, Eik List, Stefan Lucks, Willi Meier, Mostafizar Rahman, Dhiman Saha, and Yu Sasaki. Cryptanalysis of forkaes. Cryptology ePrint Archive, Report 2019/289, 2019. `https://eprint.iacr.org/2019/289`.

[13] Ray Beaulieu, Douglas Shors, Jason Smith, Stefan Treatman-Clark, Bryan Weeks, and Louis Wingers. SIMON and SPECK: Block ciphers for the internet of things. Cryptology ePrint Archive, Report 2015/585, 2015. `http://eprint.iacr.org/2015/585`.

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

[15] Mihir Bellare, Oded Goldreich, and Anton Mityagin. The Power of Verification Queries in Message Authentication and Authenticated Encryption. *IACR Cryptology ePrint Archive*, 2004:309, 2004.

[16] Mihir Bellare and Phillip Rogaway. The security of triple encryption and a framework for code-based game-playing proofs. 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 *Lecture Notes in Computer Science*, pages 409–426. Springer, 2006.

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

[18] Eli Biham and Adi Shamir. Differential cryptanalysis of DES-like cryptosystems. *Journal of Cryptology*, 4(1):3–72, January 1991.

[19] Jannis Bossert, Eik List, and Stefan Lucks. Boomerang and rectangle attacks on forkaes. personal correspondence, 2018.

[20] M. Dworkin. Recommendation for block cipher modes of operation: Galois/Counter Mode (GCM) and GMAC. NIST Special Publication 800-38D, November 2007.

[21] Jérémy Jean. TikZ for Cryptographers. `https://www.iacr.org/authors/tikz/`, 2016.

[22] Jérémy Jean, Ivica Nikolić, and Thomas Peyrin. Deoxys v1. *Submitted to the CAESAR competition*, 2014.

[23] Jérémy Jean, Ivica Nikolić, and Thomas Peyrin. Joltik v1. *Submitted to the CAESAR competition*, 2014.

[24] Jérémy Jean, Ivica Nikolić, and Thomas Peyrin. Kiasu v1. *Submitted to the CAESAR competition*, 2014.

[25] Jérémy Jean, Ivica Nikolic, and Thomas Peyrin. Tweaks and keys for block ciphers: The TWEAKEY framework. In Palash Sarkar and Tetsu Iwata, editors, *ASIACRYPT 2014, Part II*, volume 8874 of *LNCS*, pages 274–288. Springer, Heidelberg, December 2014.

[26] Jérémy Jean, Ivica Nikolić, Thomas Peyrin, and Yannick Seurin. Deoxys v1. 41 (2016).

[27] Lars Knudsen. Deal-a 128-bit block cipher. *complexity*, 258(2):216, 1998.

[28] Thorsten Kranz, Gregor Leander, and Friedrich Wiemer. Linear cryptanalysis: Key schedules and tweakable block ciphers. *IACR Trans. Symm. Cryptol.*, 2017(1):474–505, 2017.

[29] Ted Krovetz and Phillip Rogaway. The Software Performance of Authenticated-Encryption Modes. In Antoine Joux, editor, *FSE 2011*, volume 6733 of *LNCS*, pages 306–327. Springer, 2011.

[30] Guozhen Liu, Mohona Ghosh, and Ling Song. Security analysis of SKINNY under related-tweakey settings (long paper). *IACR Trans. Symm. Cryptol.*, 2017(3):37–72, 2017.

[31] Mitsuru Matsui. Linear cryptanalysis method for DES cipher. In Tor Helleseth, editor, *EUROCRYPT'93*, volume 765 of *LNCS*, pages 386–397. Springer, Heidelberg, May 1994.

[32] David A. McGrew and John Viega. The security and performance of the galois/counter mode (GCM) of operation. In Anne Canteaut and Kapalee Viswanathan, editors, *Progress in Cryptology - INDOCRYPT 2004, 5th International Conference on Cryptology in India, Chennai, India, December 20-22, 2004, Proceedings*, volume 3348 of *Lecture Notes in Computer Science*, pages 343–355. Springer, 2004.

[33] AmirHossein E. Moghaddam and Zahra Ahmadian. New automatic search method for truncated-differential characteristics: Application to midori and skinny. Cryptology ePrint Archive, Report 2019/126, 2019. `https://eprint.iacr.org/2019/126`.

[34] Svetla Nikova, Christian Rechberger, and Vincent Rijmen. Threshold implementations against side-channel attacks and glitches. In *Information and Communications Security, 8th International Conference, ICICS 2006, Raleigh, NC, USA, December 4-7, 2006, Proceedings*, pages 529–545, 2006.

[35] NIST. DRAFT Submission Requirements and Evaluation Criteria for the Lightweight Cryptography Standardization Process. `https://csrc.nist.gov/Projects/Lightweight-Cryptography`, 2018.

[36] Phillip Rogaway. Authenticated-Encryption with Associated-Data. In *ACM CCS 2002*, pages 98–107, 2002.

[37] Sadegh Sadeghi, Tahereh Mohammadi, and Nasour Bagheri. Cryptanalysis of reduced round SKINNY block cipher. *IACR Trans. Symm. Cryptol.*, 2018(3):124–162, 2018.

[38] Danping Shi, Siwei Sun, Patrick Derbez, Yosuke Todo, Bing Sun, and Lei Hu. Programming the Demirci-Selçuk meet-in-the-middle attack with constraints. In Thomas Peyrin and Steven Galbraith, editors, *ASIACRYPT 2018, Part II*, volume 11273 of *LNCS*, pages 3–34. Springer, Heidelberg, December 2018.

[39] Ling Song, Xianrui Qin, and Lei Hu. Boomerang connectivity table revisited. Cryptology ePrint Archive, Report 2019/146, 2019. `https://eprint.iacr.org/2019/146`.

[40] Siwei Sun, David Gerault, Pascal Lafourcade, Qianqian Yang, Yosuke Todo, Kexin Qiao, and Lei Hu. Analysis of AES, SKINNY, and others with constraint programming. *IACR Trans. Symm. Cryptol.*, 2017(1):281–306, 2017.

[41] Yosuke Todo. Structural evaluation by generalized integral property. In Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT 2015, Part I*, volume 9056 of *LNCS*, pages 287–314. Springer, Heidelberg, April 2015.

[42] Mohamed Tolba, Ahmed Abdelkhalek, and Amr M. Youssef. Impossible differential cryptanalysis of reduced-round SKINNY. In Marc Joye and Abderrahmane Nitaj, editors, *AFRICACRYPT 17*, volume 10239 of *LNCS*, pages 117–134. Springer, Heidelberg, May 2017.

[43] David Wagner. The boomerang attack. In Lars R. Knudsen, editor, *FSE'99*, volume 1636 of *LNCS*, pages 156–170. Springer, Heidelberg, March 1999.

[44] D. Whiting, R. Housley, and N. Ferguson. Counter with CBC-MAC (CCM). IETF RFC 3610 (Informational), September 2003.

[45] Pei Zhang and Wenying Zhang. Differential cryptanalysis on block cipher skinny with MILP program. *Security and Communication Networks*, 2018:3780407:1–3780407:11, 2018.

[46] Wenying Zhang and Vincent Rijmen. Division cryptanalysis of block ciphers with a binary diffusion layer. Cryptology ePrint Archive, Report 2017/188, 2017. `http://eprint.iacr.org/2017/188`.

A Formal Definitions of Forkcipher and Authenticated Encryption

The following formalization of the forkcipher is taken from [9]

A.1 Syntax

A forkcipher is a pair of deterministic algorithms, the encryption algorithm

$$\mathsf{F} : \{0,1\}^k \times \mathcal{T} \times \{0,1\}^n \times \{0,1,\mathsf{b}\} \rightarrow \{0,1\}^n \cup \{0,1\}^n \times \{0,1\}^n$$

and the inversion algorithm

$$\mathsf{F}^{-1}\{0,1\}^k \times \mathcal{T} \times \{0,1\}^n \times \{0,1\} \times \{\mathsf{i},\mathsf{o},\mathsf{b}\} \rightarrow \{0,1\}^n \cup \{0,1\}^n \times \{0,1\}^n.$$

The encryption algorithm takes a key K, a tweak T, a plaintext block X and an output selector s, and outputs the "left" n-bit ciphertext block C_0 if $s = 0$, the "right" n-bit ciphertext block C_1 if $s = 1$, and a **b**oth blocks C_0, C_1 if $s = \mathsf{b}$. We write $\mathsf{F}(K, \mathsf{T}, M, s) = \mathsf{F}_K(\mathsf{T}, M, s) = \mathsf{F}_K^{\mathsf{T}}(M, s) = \mathsf{F}_K^{\mathsf{T},s}(M)$ interchangeably. The decryption algorithm takes a key K, a tweak T, a ciphertext block C, an indicator b of whether this is the left or the right ciphertext block and an output selector s, and outputs the plaintext (or **i**nverse) block X if $s = \mathsf{i}$, the **o**ther ciphertext block C' if $s = \mathsf{o}$, and **b**oth blocks X, C' if $s = \mathsf{b}$. We write $\mathsf{F}^{-1}(K, \mathsf{T}, M, b, s) = \mathsf{F}^{-1}{}_K(\mathsf{T}, M, b, s) = \mathsf{F}^{-1}{}_K^{\mathsf{T}}(M, b, s) = \mathsf{F}_K^{\mathsf{T},b,s}(M)$ interchangeably. We call k, n and $\mathcal{T}$ the keysize, blocksize and tweak space of F, respectively.

A tweakable forkcipher F meets the *correctness condition*, if for every $K \in \{0,1\}^k, \mathsf{T} \in \mathcal{T}, X \in \{0,1\}^n$ and $\beta \in \{0,1\}$ all of the following conditions are met:

1. $\mathsf{F}^{-1}(K, \mathsf{T}, \mathsf{F}(K, \mathsf{T}, X, \beta), \beta, \mathsf{i}) = X$
2. $\mathsf{F}^{-1}(K, \mathsf{T}, \mathsf{F}(K, \mathsf{T}, X, \beta), \beta, \mathsf{o}) = \mathsf{F}(K, \mathsf{T}, X, \beta \oplus 1)$
3. $(\mathsf{F}(K, \mathsf{T}, X, 0), \mathsf{F}(K, \mathsf{T}, X, 1)) = \mathsf{F}(K, \mathsf{T}, X, \mathsf{b})$
4. $(\mathsf{F}^{-1}(K, \mathsf{T}, X, \beta, \mathsf{i}), \mathsf{F}^{-1}(K, \mathsf{T}, X, \beta, \mathsf{o})) = \mathsf{F}^{-1}(K, \mathsf{T}, X, \beta, \mathsf{b})$

In other words, for each pair of key and tweak, the forkcipher applies two independent permutations to the input to produce the two output blocks.

A.2 Security Definition of Forkcipher

An adversary $\mathcal{A}$ that aims at breaking a tweakable forkcipher F plays the games **prtfp-real** and **prtfp-ideal** and define the advantage of $\mathcal{A}$ at distinguishing F from a pair of random tweakable permutations in a *chosen ciphertext attack* as

$$\mathbf{Adv}_{\mathsf{F}}^{\mathrm{prtfp}}(\mathcal{A}) = \Pr[\mathcal{A}^{\mathbf{prtfp\text{-}real}_{\mathsf{F}}} \Rightarrow 1] - \Pr[\mathcal{A}^{\mathbf{prtfp\text{-}ideal}_{\mathsf{F}}} \Rightarrow 1].$$

A.3 Security Definition of Authenticated Encryption

We use the two-requirement definition of AE security. We model a chosen plaintext attack of an adversary $\mathcal{A}$ against the confidentiality of a nonce-based AE scheme Π with the help of the security games **priv-real** and **priv-ideal** in Figure 11. We define the advantage of $\mathcal{A}$ in breaking the confidentiality of Π as

$$\mathbf{Adv}_{\Pi}^{\mathbf{priv}}(\mathcal{A}) = \Pr[\mathcal{A}^{\mathbf{priv\text{-}real}_{\Pi}} \Rightarrow 1] - \Pr[\mathcal{A}^{\mathbf{priv\text{-}ideal}_{\Pi}} \Rightarrow 1].$$

We model a chosen ciphertext attack against the integrity of Π with help of the game **auth** in Figure 11. We define the advantage of $\mathcal{A}$ in breaking the integrity of Π as

$$\mathbf{Adv}_{\Pi}^{\mathbf{priv}}(\mathcal{A}) = \Pr[\mathcal{A}^{\mathbf{auth}_{\Pi}} \text{forges}]$$

where "$\mathcal{A}$ forges" denotes a decryption query that returns a value $\neq \bot$.

B Security Analysis of PAEF

Theorem 1. *Let F be a tweakable forkcipher with $\mathcal{T} = \{0,1\}^\tau$, and let $0 < \nu \leq \tau - 4$. Then for any nonce-respecting adversary $\mathcal{A}$ whose queries lie in the proper domains of the encryption and decryption algorithms and who makes at most q_v decryption queries, we have*

$$\mathbf{Adv}_{\mathrm{PAEF}[\mathsf{F},\nu]}^{\mathbf{priv}}(\mathcal{A}) \leq \mathbf{Adv}_{\mathsf{F}}^{\mathbf{prtfp}}(\mathcal{B})$$

and

$$\mathbf{Adv}_{\mathrm{PAEF}[\mathsf{F},\nu]}^{\mathbf{auth}}(\mathcal{A}) \leq \mathbf{Adv}_{\mathsf{F}}^{\mathbf{prfp}}(\mathcal{C}) + \frac{q_v \cdot 2^n}{(2^n - 1)^2}$$

for some adversaries $\mathcal{B}$ and $\mathcal{C}$ who make at most twice as many queries in total as is the total number of blocks in all encryption, respectively all encryption and decryption queries made by $\mathcal{A}$, and who run in time given by the running time of $\mathcal{A}$ plus an overhead that is linear in the total number of blocks in all $\mathcal{A}$'s queries.

Proof. Below we prove the confidentiality and authenticity of the PAEF mode. For both confidentiality and authenticity, we first replace F with a pair of independent random

tweakable permutations π_0, π_1, i.e. $\pi_0 = (\pi_{\mathsf{T},0} \leftarrow\$ \text{Perm}n)_{\mathsf{T}\in\{0,1\}^\tau}$ is a collection of independent uniform elements of Perm$n$ indexed by the elements of $\mathsf{T} \in \{0,1\}^\tau$ (and similarly $\pi_1 = (\pi_{\mathsf{T},1} \leftarrow\$ \text{Perm}n)_{\mathsf{T}\in\{0,1\}^\tau}$). We let PAEF$[(\pi_0,\pi_1),\nu]$ denote the PAEF mode that uses π_0, π_1 instead of F. We have that

$$\mathbf{Adv}^{\mathbf{priv}}_{\text{PAEF}[\mathsf{F},\nu]}(\mathcal{A}) \leq \mathbf{Adv}^{\mathbf{prtfp}}_{\mathsf{F}}(\mathcal{B}) + \mathbf{Adv}^{\mathbf{priv}}_{\text{PAEF}[(\pi_0,\pi_1),\nu]}(\mathcal{A})$$

because a distinguisher $\mathcal{B}$ for F can perfectly simulate the games $\mathbf{priv\text{-}real}_{PAEF[\mathsf{F},\nu]}$ and $\mathbf{priv\text{-}real}_{PAEF[(\pi_0,\pi_1),\nu]}$ for $\mathcal{A}$ using its own oracles. In place of any F^ρ call, $\mathcal{B}$ has to make a decryption query followed by an encryption query. By copying $\mathcal{A}$'s output, $\mathcal{B}$ can achieve the same advantage as $\mathcal{A}$ does, with the same data complexity as $\mathcal{A}$ and a very similar running time. This implies that the gap between these games is bounded by $\mathbf{Adv}^{\mathbf{prtfp}}_{\mathsf{F}}(\mathcal{B})$. By a similar argument, we have that

$$\mathbf{Adv}^{\mathbf{auth}}_{\text{PAEF}[\mathsf{F},\nu]}(\mathcal{A}) \leq \mathbf{Adv}^{\mathbf{prtfp}}_{\mathsf{F}}(\mathcal{C}) + \mathbf{Adv}^{\mathbf{priv}}_{\text{PAEF}[(\pi_0,\pi_1),\nu]}(\mathcal{A}).$$

For confidentiality, it is easy to see that in a nonce-respecting attack, every message block is processed with a unique tweak. Every ciphertext block and every tag is produced as the only image under and independent random permutation (or a substring of such), and thus uniformly distributed. The final block C_* of every ciphertext is produced as a xor-sum of outputs of π_0 and π_1, each produced with a unique tweak, and thus uniformly distributed. Since all ciphertexts are uniformly distributed we get perfect confidentiality and hence our result.

For authenticity, we analyse the probability of forgery for an adversary that makes a single decryption query against PAEF$[(\pi_0,\pi_1),\nu]$ and then use a result of Bellare [15] to extend our result to multiple queries (still against PAEF$[(\pi_0,\pi_1),\nu]$).

We will denote the encryption queries of $\mathcal{A}$ and the corresponding replies as (N^i, A^i, M^i) and C^i for $i = 1,\ldots,q$, where q is the number of encryption queries made by $\mathcal{A}$. For each i we let $C^i_1,\ldots,C^i_m,C^i_*,T = \mathsf{csplit\text{-}b}_n(C^i)$. We let (N,A,C) denote the only decryption query of $\mathcal{A}$ and we let $C_1,\ldots,C_m,C_*,T = \mathsf{csplit\text{-}b}_n(C)$. When the forgery (N,A,C) is made, we have two base cases. If the nonce N is fresh, then the forgery attempt is equivalent to guessing the value of a uniform string of n bits, and thus succeeds with probability 2^{-n}. This holds even if $|T| < n$, because the rightmost $(n - |T|)$ bits of the preimage of C_* under π_0 must have a specific value.

If N is reused, i.e. if $N = N^i$ for some $N^i \in \{N^1,\ldots,N^q\}$, then we perform a case analysis. Note that we can disregard all encryption queries except the ith, because their ciphertetxts are computed using independent random permutations. Every case assumes the negation of all previous case-conditions.

Case 1, $|C|_n \neq |C^i|_n$: We have several subcases.

- If $|C| = n$, then C is equal to a xor-sum of $\pi_{\mathsf{T},1}$ images from the associated data (denoted as T_A in Figure 7), such that we can possibly have $A^i = A$. However, due to the assumption in this case, we must have $|M^i| > 0$, so the xor-sum T_{A^i}

computed in the ith encryption query is xor-masked with uniform bits produced by the processing of M_*^i. Therefore T_{A^i} is statistically independent of C^i, and the adversary has no information when trying to guess the value of the T_A sum. The probability of a successful forgery is 2^{-n}.

- When $|C| > n$, regardless if C has more or less blocks than C^i, the successful forgery is equivalent to guessing the value of an image under π_1 (respectively the value of n out of $2n$ bits produced by $\pi_{\mathsf{T},0}^{-1}(\mathsf{left}_n(T_*))$ and $\pi_{\mathsf{T},1}(\pi_{\mathsf{T},0}^{-1}(\mathsf{left}_n(T_*))))$ such that the tweak $\mathsf{T} = N\|110\|\langle m+1\rangle_{\tau-\nu-3}$ (respectively $\mathsf{T} = N\|111\|\langle m+1\rangle_{\tau-\nu-3}$) was not used before. The probability of this event is 2^{-n}.

The probability of a successful forgery in **Case 1** is at most 2^{-n}. In the following cases, $|C|_n = |C^i|_n$.

Case 2, $|A|_n \neq |A^i|_n$: Again, we have a few subcases to consider.

- If $|A|_n > |A^i|_n$, a successful forgery is equivalent to guessing an output value of $\pi_{\mathsf{T},1}$ with a previously unused tweak ($\mathsf{T} = N\|0b1\|\langle a+1\rangle_{\tau-\nu-3}$ for $b \in \{0,1\}$) thanks to $a > a_i$, succeeding with probability of 2^{-n}.
- If $0 < |A|_n < |A^i|_n$, then a successful forgery is still equivalent to guessing an output value of $\pi_{\mathsf{T},1}$ with a previously unused tweak ($\mathsf{T} = N\|0b1\|\langle a+1\rangle_{\tau-\nu-3}$ for $b \in \{0,1\}$), thanks to the three-bit domain-separation flag (which was set to 000 in the ith encryption query). This succeeds with probability 2^{-n}.
- Finally if $|A| = 0$, then $|A|_n \neq |A^i|_n$ implies that $|A^i|_n > 0$. Forging in this case is either equivalent to guessing the image $\pi_{(N\|011\|1),1}(10^{n-1})$ such that the random permutation $\pi_{(N\|011\|1),1}$ was evaluated on no more than a single other input $A_*^i\|10^* \neq 10^{n-1}$ in the whole game (if $|C| = n$), or to guessing the correct value for C_*. The former succeeds with probability at most $1/(2^n-1)$, and the latter with probability at most 2^{-n} (because the corresponding output of $\pi_{\mathsf{T},0}$ was masked by T_{A^i}).

Thus the probability of a successful forgery in this case is at most $1/(2^n-1)$. In the remaining cases, we have $|C|_n = |C^i|_n > 1$ and $|A|_n = |A^i|_n > 0$.

Case 3, $|C| \neq |C^i|$ and $|T| = n$ or $|T^i| = n$: In this case, the forgery verification will use $\pi_{\mathsf{T},1}$ with a fresh tweak T because the "incomplete-block" bit of the three-bit flag will have different values in the processing of the decryption query, and in the processing of the ith encryption query. The forgery succeeds with probability 2^{-n}.

Case 4, $|A| \neq |A^i|$ and $|A_*| = n$ or $|A_*^i| = n$: This is analogous with the previous case; the probability of forgery is 2^{-n}. In the remaining cases, we have $|C|_n = |C^i|_n > 1$, $|A|_n = |A^i|_n > 0$ and $|T| > 0, |T^i| > 0, |A_*| > 0, |A_*^i| > 0$.

Case 5, $|C| \neq |C^i|$ and $|T| < n$ and $|T^i| < n$: In this case, both the encryption query and the decryption query use the same tweak T to process M_*^i and C_*, T, respectively. There are two conditions for the forgery to succeed. First, the preimage

$X = \pi^{-1}_{\mathsf{T},0}(C_* \oplus S)$ (as per line 29 in Figure 6) must be equal to $W\|10^{n-|T|-1} \neq M^i_*\|10^{n-|T^i|-1}$ (noting that the case condition implies $|T| \neq |T^i|$) for some $W \in \{0,1\}^{|T|}$. This is no easier than finding a fresh value whose preimage falls into a set of size $2^{|T|}$. With a single image of $\pi^{-1}_{\mathsf{T},0}$ already used, this succeeds with probability bounded by $(2^{|T|})/(2^n-1)$. *Secondly*, the image $Y = \pi_{\mathsf{T},1}(X)$ must be equal to $T\|Z$ for some $Z \in \{0,1\}^{n-|T|}$, *conditioned on* X *having the correct format.* This is equivalent to guessing a fresh image under $\pi_{\mathsf{T},1}$ with $(n-|T|)$ free bits. As a single image of $\pi_{\mathsf{T},1}$ has been used already, this happens with probability at most $(2^{n-|T|})/(2^n-1)$. The probability of a successful forgery in this case is therefore bounded by $(2^{|T|})/(2^n-1) \cdot (2^{n-|T|})/(2^n-1) = 2^n/(2^n-1)^2$.

Case 6, $|A| \neq |A^i|$ **and** $|A_a| < n$ **and** $|A^i_a| < n$**:** In this case, the final blocks A_* and A^i_* are processed by the same random permutation $\pi_{\mathsf{T},1}$, but as $A_*\|10^{n-|A_*|} \neq A^i_*\|10^{n-|A^i_*|}$, successfully forging in this case is equivalent to guessing the yet unsampled image $\pi_{\mathsf{T},1}(A_*\|10^{n-|A_*|})$. With a single image of $\pi_{\mathsf{T},1}$ used before, this happens with probability at most $1/(2^n-1)$.

Case 7, $|C| = |C^i|$ **and** $|A| = |A^i|$**:** In this case, there must be at least a single block of either AD or ciphertext where the two queries differ. We investigate the following subcases.

- If the forgery N, A, C differs from N, A^i, C^i only in $C_*\|T$, then , if we ran the decryption algorithm on N, A^i, C^i and N, A, C in parallel, the values S^i and S used on the line 29 of the decryption algorithm in Figure 6 would be the same, and thus necessarily $(C_* \oplus S)\|T \neq (C^i_* \oplus S^i)\|T^i$. The probability of a successful forgery is at $(2^n-1)^{-1}$ if $|T| = n$ (inverse of $C_* \oplus S$ has not yet been sampled) and at most $2^n/(2^n-1)^2$ otherwise (by a similar argument as in **Case 5**).
- If $A, C\|T$ and $A^i, C^i\|T^i$ differ in a single block, such that $C_*\|T = C^i_*\|T^i$, a forgery is impossible (because $\pi_{\mathsf{T},0}$ and $\pi_{\mathsf{T},1}$ are all permutations).
- If there are at least two blocks in $A_1, \ldots A_a, A_*, C_1, \ldots, C_m, C_*, T$ that differ from the corresponding blocks in $A^i_1, \ldots A^i_a, A^i_*, C^i_1, \ldots, C^i_m, C^i_*, T^i$, then the forgery can succeed in two ways. The first is if $(C_* \oplus S)\|T = (C^i_* \oplus S^i)\|T^i$. This happens with probability at most $1/(2^n-1)$, as there will be at least one index j for which $A_j \neq A^i_j$ (or $C_j \neq C^i_j$), and for which $\pi_{\mathsf{T},1}(A_j) \oplus \pi_{\mathsf{T},1}(A^i_j)$ (respectively $\pi_{\mathsf{T},1}(\pi^{-1}_{\mathsf{T},0}(C_j)) \oplus \pi_{\mathsf{T},1}(\pi^{-1}_{\mathsf{T},1}(C^i_j)))$ would have to take a particular value. The probability follows from the fact that whatever T, the random permutations $\pi_{\mathsf{T},1}$ and $\pi^{-1}_{\mathsf{T},1}$ were sampled only once. The second way is if $(C_* \oplus S)\|T \neq (C^i_* \oplus S^i)\|T^i$ but the verification still succeeds. This is analogous to **Case 5**.

The probability of a successful forgery in this case is bounded by $2^n/(2^n-1)^2$.

Thus a single forgery succeeds with probability no greater than $2^n/(2^n-1)^2$. By applying the result of Bellare [15], we can bound the probability of a successful forgery among q_v decryption queries as $(q_v \cdot 2^n)/(2^n-1)^2$. □ □

C Security Analysis of SAEF

Theorem 2. *Let* F *be a tweakable forkcipher with* $\mathcal{T} = \{0,1\}^\tau$. *Then for any nonce-respecting adversary* $\mathcal{A}$ *whose makes at most* q *encryption queries, at most* q_v *decryption queries such that the total number of forkcipher calls induced by all the queries is at most* σ, *with* $\sigma \leq 2^n/2$, *we have*

$$\mathbf{Adv}^{\mathbf{priv}}_{\mathrm{SAEF}[\mathsf{F}]}(\mathcal{A}) \leq \mathbf{Adv}^{\mathbf{prtfp}}_{\mathsf{F}}(\mathcal{B}) + 2 \cdot \frac{(\sigma-q)^2}{2^n},$$
$$\mathbf{Adv}^{\mathbf{auth}}_{\mathrm{SAEF}[\mathsf{F}]}(\mathcal{A}) \leq \mathbf{Adv}^{\mathbf{prtfp}}_{\mathsf{F}}(\mathcal{C}) + \frac{(\sigma-q+1)^2}{2^n} + \frac{\sigma(\sigma-q)}{2^n} + \frac{q_v(q+2)}{2^n}$$

for some adversaries $\mathcal{B}$ *and* $\mathcal{C}$ *who make at most* 2σ *queries, and who run in time given by the running time of* $\mathcal{A}$ *plus* $\gamma \cdot \sigma$ *for some constant* γ.

Proof of SAEF. The security analysis of SAEF is slightly more involved than in the case of PAEF. We first tackle confidentiality and then integrity.

Confidentiality of SAEF. We first replace the forkcipher F with a pair of tweakable permutations π_0 and π_1. I.e. $\pi_0 = (\pi_{\mathsf{T},0} \leftarrow\$ \mathrm{Perm}n)_{\mathsf{T}\in\{0,1\}^\tau}$ is a collection of independent uniform elements of Perm$n$ indexed by the elements of $\mathsf{T} \in \{0,1\}^\tau$ (and similarly for $\pi_1 = (\pi_{\mathsf{T},1} \leftarrow\$ \mathrm{Perm}n)_{\mathsf{T}\in\{0,1\}^\tau}$). We let SAEF$[\pi_0,\pi_1]$ denote the SAEF mode that uses π_0, π_1 instead of F. This replacement implies the following inequality:

$$\mathbf{Adv}^{\mathbf{priv}}_{\mathrm{SAEF}[\mathsf{F}]}(\mathcal{A}) \leq \mathbf{Adv}^{\mathbf{prtfp}}_{\mathsf{F}}(\mathcal{B}) + \mathbf{Adv}^{\mathbf{priv}}_{\mathrm{SAEF}[\pi_0,\pi_1]}(\mathcal{A})$$

by a similar argument as in the proof of Theorem 1.

We now further replace the two families of random permutations π_0 and π_1 with families of *random functions* f_0 and f_1 with the same signature. I.e. $f_b = (f_{\mathsf{T},b} \leftarrow\$ \mathrm{Func}(n))_{\mathsf{T}\in\{0,1\}^\tau}$ for $b \in \{0,1\}$. Denoting the SAEF mode using these random functions by SAEF$[f_0, f_1]$, we have that

$$\mathbf{Adv}^{\mathbf{priv}}_{\mathrm{SAEF}[\pi_0,\pi_1]}(\mathcal{A}) \leq \mathbf{Adv}^{\mathbf{priv}}_{\mathrm{SAEF}[f_0,f_1]}(\mathcal{A}) + 2 \cdot \frac{(\sigma-q)^2}{2^{n+1}}$$

because all but the first block (be it AD or message) of each query are processed using a tweak of the form $0^{\tau-3}\|b_0b_1b_2$ with $b_0, b_1, b_2 \in \{0,1\}$. As there are no more than σ blocks of data in total, each of the permutations $\pi_{\mathsf{T},0}$ and $\pi_{\mathsf{T},1}$ processes σ_{T} blocks with $\sum_{\mathsf{T}\in\{0,1\}^\tau} \sigma_{\mathsf{T}} = \sigma$. Replacing each $\pi_{\mathsf{T},0}$ by $f_{\mathsf{T},0}$ augments the bound by at most $\sigma_{\mathsf{T}}(\sigma_{\mathsf{T}}-1) \cdot 2^{-n-1}$ by the RP-RF switching lemma [16] and a standard hybrid argument. A sum of all these augmentations is upper bounded by $(\sigma-q)^2/2^{n+1}$, noting that there

are at least q tweak values T for which $\pi_{\mathsf{T},0}$ is applied to at most a single block. Another term $(\sigma - q)^2/2^{n+1}$ needs to be added to account for the replacement of $\pi_{\mathsf{T},1}$ for all T.

We now bound $\mathbf{Adv}^{\mathbf{priv}}_{\mathrm{SAEF}[f_0,f_1]}(\mathcal{A})$. For this, we use the games G_0 and G_1 defined in Figure 12. In both games, the set $\mathcal{D}_{\mathsf{T}}$ collects the domain points, on which the functions $f_{\mathsf{T},0}$ and $f_{\mathsf{T},1}$ were already evaluated. It is easy to verify that G_0 actually implements $\mathbf{priv\text{-}real}_{\mathrm{SAEF}[f_0,f_1]}$, as the flag bad and the sets $\mathcal{D}_{\mathsf{T}}$ have no influence on the outputs of Enc. It is also possible to verify that $\Pr[\mathcal{A}^{\mathbf{priv\text{-}ideal}_{\mathrm{SAEF}[f_0,f_1]}} \Rightarrow 1] = \Pr[\mathcal{A}^{G_1} \Rightarrow 1]$: unless bad is set, every ciphertext block C_i is an xor of images of a distinct input to two random functions, and T is simply produced by applying a random function to a fresh input. Thus, all the output bits of Enc are uniform. Once bad is set, all the ciphertext blocks and each value of Δ is replaced by a uniform string, so the simulation is perfect. Thus we have $\mathbf{Adv}^{\mathbf{priv}}_{\mathrm{SAEF}[f_0,f_1]}(\mathcal{A}) \leq \Pr[\mathcal{A}^{G_0} \Rightarrow 1] - \Pr[\mathcal{A}^{G_1} \Rightarrow 1]$.

We also have that G_0 and G_1 are identical until bad, so by the Fundamental lemma of gameplaying [16] we have that $\mathbf{Adv}^{\mathbf{priv}}_{\mathrm{SAEF}[f_0,f_1]}(\mathcal{A}) \leq \Pr[\mathcal{A}^{G_0} \text{ sets } \mathsf{bad}]$, where $\mathcal{A}^{G_0}$ sets bad denotes the event that $\mathsf{bad} = \mathsf{true}$ when $\mathcal{A}$ issues its final output. We bound $\Pr[\mathcal{A}^{G_0} \text{ sets } \mathsf{bad}]$ by union bound, iterating over the probability that the ith query sets bad, given that bad was not set before.

For an encryption query (N, A, M), the initial block of that query is processed with a tweak $N\|1b_0b_1b_2$, with the corresponding set $\mathcal{D}_{N\|1b_0b_1b_2}$ empty, making it impossible to set bad. Each remaining block (be it AD or message) is masked with the Δ value before it is fed to $f_{\mathsf{T},b}$ (for $b \in \{0,1\}$ and some T). If bad has not been set before, Δ is a uniform n-bit string. Thus each such block can set bad with probability $|\mathcal{D}_{\mathsf{T},b}|/2^n$ for $b \in \{0,1\}$ and some T will be uniformly distributed due to the Δ mask produced by $f_{N\|1b_0b_1b_2,1}$. There are almost $(\sigma - q)$ blocks that can set bad when fed to $f_{\mathsf{T},b}$, and for each we have $|\mathcal{D}_{\mathsf{T},b}| \leq (\sigma - q)$. The total probability of setting bad is thus no more than $(\sigma - q)/2^n$, completing the proof of the confidentiality bound.

Integrity of SAEF. We again replace the forkcipher F with a pair of tweakable permutations $\pi_0 = (\pi_{\mathsf{T},0} \leftarrow\$ \operatorname{Perm}n)_{\mathsf{T}\in\{0,1\}^\tau}$ and $\pi_1 = (\pi_{\mathsf{T},1} \leftarrow\$ \operatorname{Perm}n)_{\mathsf{T}\in\{0,1\}^\tau}$, such that we have

$$\mathbf{Adv}^{\mathbf{auth}}_{\mathrm{SAEF}[\mathsf{F}]}(\mathcal{A}) \leq \mathbf{Adv}^{\mathbf{prtfp}}_{\mathsf{F}}(\mathcal{C}) + \mathbf{Adv}^{\mathbf{auth}}_{\mathrm{SAEF}[\pi_0,\pi_1]}(\mathcal{A})$$

by a similar argument as in the proof of Theorem 1.

We additionally replace the tweakable permutation π_1 by a tweakable function f_1 with the same signature, yielding

$$\mathbf{Adv}^{\mathbf{auth}}_{\mathrm{SAEF}[\pi_0,\pi_1]}(\mathcal{A}) \leq \mathbf{Adv}^{\mathbf{auth}}_{\mathrm{SAEF}[\pi_0,f_1]}(\mathcal{A}) + \frac{(\sigma - q + 1)^2}{2^{n+1}}$$

by a similar argument as in the proof of SAEF's confidentiality; the difference here is that $\mathcal{A}$ may force a permutation $\pi_{N\|b_0b_1b_2,1}$ to be used $\sigma - q + 1$ by making all decryption queries with N.

To bound $\mathbf{Adv}^{\mathbf{auth}}_{\mathrm{SAEF}[\pi_0,f_1]}(\mathcal{A})$, we consider the games G_2 and G_3 in Figures 13 and 14. It is easy to see that the game G_2 actually implements the game $\mathbf{auth}_{\mathrm{SAEF}[\pi_0,f_1]}$, because the

sets $\mathcal{D}_\mathsf{T}$ for $\mathsf{T} \in \{0,1\}^\tau$ and the flag bad have no effect on the outputs of the game. Moreover, unless bad is set to true, the games G_2 and G_3 execute the same code. Thus, by the Fundamental lemma of gameplaying [16], we have that $\Pr[\mathcal{A}^{G_2} \text{ forges}] - \Pr[\mathcal{A}^{G_3} \text{ forges}] \leq \Pr[\mathcal{A}^{G_2} \text{ sets } \mathsf{bad}]$ and consequently $\mathbf{Adv}^{\mathbf{auth}}_{\mathrm{SAEF}[\pi_0, f_1]}(\mathcal{A}) \leq \Pr[\mathcal{A}^{G_2} \text{ sets } \mathsf{bad}] + \Pr[\mathcal{A}^{G_3} \text{ forges}]$.

Transition from G_2 to G_3. The flag bad being set means that for some $\mathsf{T} \in \{0,1\}^\tau$, the permutation $\pi_{\mathsf{T},0}$ and the function $f_{\mathsf{T},1}$ were used twice on the same input in an encryption query, beyond a trivial prefix of the two queries. Informally speaking, this event may allow the adversary to forge trivially by simply truncating the ciphertext, or the associated data used in an encryption query with such a collision. We disallow this kind of victory in the game G_3.

Some of the conditions that can set bad use predicates $P^i_\mathrm{A}(W,Q)$, $P^*_\mathrm{A}(W,Q)$ and $P^i_\mathrm{M}(W,Q)$. These predicates return true if the current query is, up to the currently processed block, **not** a blockwise prefix of some previous query. More precisely, the predicate $P^i_\mathrm{A}(W,Q)$ (with $W = (N,A,C)$ or (N,A)) returns *false* if and only if(1) $Q(N) \neq \emptyset$ and (2) there is a tuple (N,A') or (N,A',C') such that $A_j = A'_j$ for $j = 1,\ldots,i$. The predicate $P^*_\mathrm{A}(W,Q)$ is the same as the predicate $P^i_\mathrm{A}(W,Q)$ except condition (2) becomes that there is a tuple (N,A') or (N,A',C') such that $A_j = A'_j$ for $j = 1,\ldots,a$ and $A_* = A'_*$. Finally the predicate $P^i_\mathrm{A}(W,Q)$ returns *false* if and only if $P^*_\mathrm{A}(W,Q)$ is false, and if additionally $C_j = C'_j$ for $j = 1,\ldots,i$. Note that the three predicates generate a monotonic sequence when a query is processed; once one predicate returns true, all will return true in the same query. Note also that in the decrytpion queries, checking the collisions in the domain of any $\pi_{\mathsf{T},0}$ is equivalent with checking the collisions in the range, as each $\pi_{\mathsf{T},0}$ is a permutation. Similarly as in the proof of confidentiality bound, we bound $\Pr[\mathcal{A}^{G_2} \text{ sets } \mathsf{bad}]$ by the union bound, iterating over the probability that the ith query sets bad, given that bad was not set before.

In an encryption query (N,A,M), the flag bad can be set during AD processing only after $P^i_\mathrm{A}(W,Q)$ (or $P^*_\mathrm{A}(W,Q)$) are returning true. The first block A_i (or A_*), for which the predicate is true comes right after the longest blockwise prefix with previous queries, so the current mask $\Delta = \Delta'$ for the corresponding Δ' in the previous query (N,A') that yields the common prefix, but $A_i \neq A'_i$ (or $A_* \neq A'_*$). The value of Δ' is statistically independent of the ciphertexts returned to $\mathcal{A}$, and so $A_i \oplus \Delta \in \mathcal{D}_\mathsf{T}$ (or $\mathsf{pad10}(A_*) \oplus \Delta \in \mathcal{D}_\mathsf{T}$) falls into $\mathcal{D}_\mathsf{T}$ with probability $|\mathcal{D}_\mathsf{T}|/2^n \leq (\sigma - q)/2^n$ by a similar argument as in the confidentiality proof of SAEF. For all the consequent blocks B of AD or message, if bad is not set before B is being processed, the Δ value that is used to mask B is a uniformly distributed string, so $B \oplus \Delta \in \mathcal{D}_\mathsf{T}$ with probability $|\mathcal{D}_\mathsf{T}|/2^n \leq (\sigma - q)/2^n$ as well.

In a decryption query (N,A,C), bad can only be set after the first time $P^i_\mathrm{A}(W,Q)$, $P^*_\mathrm{A}(W,Q)$, or $P^i_\mathrm{M}(W,Q)$ return true. Similarly as in an encryption query, the first block B for which this occurs will be masked by a reused Δ, but this Δ will be independent of the observed ciphertexts (even if B is a message block, because $\neg\mathsf{bad}$ implies that each ciphertext block was computed with a fresh uniform mask). For the consequent blocks, $\neg\mathsf{bad}$ implies that Δ is fresh and uniformly distributed. Thus $B \oplus \Delta \in \mathcal{D}_\mathsf{T}$ with probability $|\mathcal{D}_\mathsf{T}|/2^n \leq (\sigma - q)/2^n$.

By summing over all σ blocks, we get $\Pr[\mathcal{A}^{G_2}$ sets $\mathsf{bad}] \leq \sigma(\sigma - q)/2^n$.

FORGERY IN G_3. We proceed to bounding $\Pr[\mathcal{A}^{G_3}$ forges]. We carry out the analysis for an adversary $\mathcal{A}'$ that makes a single verification query, and then obtain $\Pr[\mathcal{A}^{G_3}$ forges] $\leq q_v \cdot \Pr[\mathcal{A}'^{G_3}$ forges], referring to a result by Bellare to support the claim [15]. We establish the bound by the means of a case analysis.

In what follows, we let $(N^i, A^i, M^i), C^i$ denote the ith encryption query made by $\mathcal{A}'$, and (N, A, C) denote the only decryption query. For each i, we let $C_1^i, \ldots, C_m^i, C_*^i, T^i \leftarrow \mathsf{csplit\text{-}b}_n(C^i)$ and we let $C_1, \ldots, C_m, C_*, T \leftarrow \mathsf{csplit\text{-}b}_n(C)$. Additionally, we will refer to the values of the Δ variable. We will indicate by $\Delta_{A,j}$ the jth value that the variable Δ takes when processing the jth block of A from the decryption query (N, A, C), and by $\Delta_{M,j}$ the jth value that the variable Δ takes when processing the jth block of the ciphertext C. We note that we can have $j = *$ and that $\Delta_{A,1} = 0^n$. We define $\Delta_{A,j}^i$ and $\Delta_{M,j}^i$ in a similar way for (N^i, A^i, M^i).

Case 1, $A = \varepsilon$ and $|C|_n \leq 2$, or $|A|_n = 1$ and $|C|_n = 1$: We have two sub-cases.

Case 1.1, $\nexists N^i$ such that $N = N^i$: In this case, the forgery equals to guessing n random bits, as the verification uses $\pi_{N\|b_0b_1b_2,0}$ and $f_{N\|b_0b_1b_2,0}$, which have not been sampled before because of the freshness of the nonce.

Case 1.2, $\exists N^i$ such that $N = N^i$ but $|A|_n + |C|_n > 2$: Also in this case, the forgery equals to guessing n random bits, as the verification uses $\pi_{N\|b_0b_1b_2,0}$ and $f_{N\|b_0b_1b_2,0}$, which have not been sampled before because the N was not used with the binary flags $b_0b_1b_2$.

Case 1.3, $\exists N^i$ such that $N = N^i$ and $|A|_n + |C|_n \leq 2$: $\mathcal{A}'$ knows a at most a single image under each $\pi_{N\|b_0b_1b_2,0}$ (C_*) and $f_{N\|b_0b_1b_2,0}$ (T). If the forgery attempt is with an AD block, or a ciphertext corresponding to a complete message block, the adversary has to guess a fresh image under $f_{N\|b_0b_1b_2,0}$, succeeding with probability 2^{-n}. If $\mathcal{A}'$ tries to forge with a ciphertext corresponding to an incomplete message block, the freshly sampled preimage $M_* = \pi^{-1}_{N\|101,0}(C_* \oplus \Delta_{M,*})$ will need to be of the form $X = Z\|10^*$ for some $Z \in \{0,1\}^{|T|}$ *and* simultaneously, the first $|T|$ bits of the freshly sampled image $Y = f_{N\|101,1}(M_*)$ will need to be equal to T. This happens with probability no greater than $((2^{|T|} - 1) \cdot (2^{n-|T|}))/((2^n - 1) \cdot 2^n) \leq 1/(2^n - 1)$.

The probability of forgery in this case is no more than $1/(2^n - 1)$.

The following cases assume the negation of the condition in **Case 1** (i.e., the forgery attempt consist of more than a single block in total).

Case 2: The tag computation is not done right after the trivial prefix with (N, A^i, C^i). More formally, we have the following subcases:

Case 2.1, $|C|_n = 1$ and $P_{\mathsf{A}}^a(W, Q) =$ true: In this case, the tag is verified in AD processing using A_* and a mask $\Delta_{A,*}$. Due to the condition in this case (and the fact that a domain collision on $f_{\mathsf{T},1}$ sets bad and ends the game), Δ is computed as an

image of $f_{\mathsf{T},1}$ evaluated on a fresh input, and thus uniform. The forgery can either succeed if $A_* \oplus \Delta_{A,*}$ equals to a value $A_*^j \oplus \Delta_{A,*}^j$ that has already been fed to $f_{\mathsf{T},1}$ in the jth encryption query (then $\mathcal{A}'$ can reuse C_*^j). As $j \in \{1, \ldots, q\}$ (T is used at most once per query), this happens with probability at most $q/2^n$. If this collision does not succeed, then the adversary must guess a fresh image under $f_{\mathsf{T},1}$, which succeeds with probability 2^{-n}. The total forgery probability in this case is bounded by $(q+1)/2^n$.

Case 2.2, $|C|_n > 1$ and $P_{\mathsf{M}}^m(W, Q) =$ true: In this case, the tag is verified in message processing using C_*, tweak $\mathsf{T} \in \{0^{\tau-3}\|100, 0^{\tau-3}\|101\}$ and a mask $\Delta_{M,*}$. Similarly as in **Case 2.1**, the Δ mask is a uniform string, and the forgery can either succeed if $C_* \oplus \Delta_{M,*}$ is equal to an already-used range point $C_*^j \oplus \Delta_{M,*}^j$ of $\pi_{\mathsf{T},0}$ (allowing $\mathcal{A}'$ to reuse the corresponding tag), or by guessing a correct value and length of the tag. The former succeeds with probability at most $q/2^n$. For the latter, we explore two brief subcases.

Case 2.2.1, $|T| = n$. In this case, the fact that $C_* \oplus \Delta_{M,*}$ is fresh implies that $M_* \oplus \Delta_{M,*}$ has not been fed to $f_{\mathsf{T},1}$ before, and a successful forgery equals to guessing a value of a uniform n-bit string. This happens with probability at most 2^{-n}.

Case 2.2.2, $|T| < n$. In this case, the yet unknown preimage $M_* = \pi_{\mathsf{T},0}^{-1}(C_* \oplus \Delta_{M,*})$ must have the form $M_* = Z\|10^{n-|T|-1}$ for some $Z \in \{0,1\}^{|T|}$, and the yet unknown image $f_{\mathsf{T},1}(M_* \oplus \Delta_{M,*})$ has to be equal to $T\|Y$ for some $Y \in \{0,1\}^{n-|T|}$. This happens with probability at most $(2^{|T|}/(2^n - \sigma)) \cdot (2^{n-|T|}/2^n) \leq 2/2^n$.

The total probability of forgery in **Case 2.2** is bounded by $(q+2)/2^n$.

The probability of forgery in **Case 2** is at most $(q+2)/2^n$.

Case 3: In the final case, the tag verification is done right after the trivial prefix with (N, A^i, C^i). More formally, we have the following subcases:

Case 3.1, $|C|_n = 1$ and $P_{\mathsf{A}}^a(W, Q) =$ false: In this case, the tag is verified in AD processing using A_*, right after the trivial prefix with the ith encryption query, using a tweak $\mathsf{T} \in \{0^{\tau-3}\|110, 0^{\tau-3}\|111\}$ and a mask $\Delta_{A,*} = \Delta_{A,*}^i$ (for the corresponding mask in the ith encryption query). We must have that $A_* \neq A_*^i$ (otherwise the forgery attempt would be invalid), so C_*^i can't be reused (as necessarily $C_* \neq C_*^i$). $\mathcal{A}'$ may attempt to force $A_* \oplus \Delta_{A,*} = A_*^j \oplus \Delta_{A,*}$ and reuse C_*^j for $j \neq i$, but this happens with probability at most $q/2^n$, similarly as in **Case 2.1**. This is because $\Delta_{A,*} = \Delta_{A,*}^j$ is statistically independent of the ciphertexts observed by the adversary. Otherwise $\mathcal{A}'$ can forge by guessing the correct value for C_* succeeding with probability 2^{-n}. The total probability of forgery in this case is no more than $(q+1)/2^n$.

Case 3.2, $|C|_n > 1$ and $P_{\mathsf{M}}^m(W, Q) =$ false: In this case, the tag is verified in message processing right after the trivial prefix with the ith encryption query, using C_*, tweak $\mathsf{T} \in \{0^{\tau-3}\|100, 0^{\tau-3}\|101\}$ and a mask $\Delta_{M,*}$. This case is analogous to **Case 2.2**, except that $\Delta_{M,*} = \Delta_{M,*}^i$ has already been used before. Yet, $\Delta_{M,*} = \Delta_{M,*}^i$

is statistically independent from the observed ciphertexts (if bad is not set, every ciphertext block is equal to an image of $\pi_{\mathsf{T},0}$ masked with an independent uniform string). Thus the argumentation of **Case 2.2** carries over, and the probability of forgery in **Case 3.2** is no more than $(q+2)/2^n$.

By taking the maximum over all cases, the probability that a single-decryption-query adversary $\mathcal{A}'$ forgers in the game G_3 is at most $(q+2)/2^n$. The adversary $\mathcal{A}$ making q_v decryption queries thus forges with probability bounded by $q_v \cdot (q+2)/2^n$. By back-substituting all the previous equalities, we obtain the claimed result. □

D Security Analysis of ForkSkinny

D.1 Arguments deduced from the Security of SKINNY

As noted previously, the security analyses of SKINNY directly transfer to ForkSkinny in the scenario where an attacker try to attack the cipher from the knowledge of both M and C_1. Consequently, to justify the security of our construction we give an overview of the main attacks published so far: Table 11 details how many rounds can be reached together with the complexities of the attacks (note that we focus our review on the versions of SKINNY with the same parameters as in our ForkSkinny candidates).

Table 11: Complexities of the main previous cryptanalyses of SKINNY-64-192, SKINNY-128-256 and SKINNY-128-384. The letters indicate if it is in the Related (R) or Single (S) tweakey scenario.

Version	Technique	Rounds	Time	Data	Memory	ref.
SKINNY-64-192	Rect.(R)	27/40	$2^{165.5}$	$2^{63.5}$	2^{80}	[30]
SKINNY-64-192	Impossib.(S)	22/40	$2^{183.97}$	$2^{47.84}$	$2^{74.84}$	[42]
SKINNY-128-256	Impossib.(R)	23/48	$2^{251.47}$	$2^{124.47}$	2^{248}	[30]
SKINNY-128-256	Impossib.(S)	20/48	$2^{245.72}$	$2^{92.1}$	$2^{147.1}$	[42]
SKINNY-128-384	Rect.(R)	27/56	2^{331}	2^{123}	2^{155}	[30]
SKINNY-128-384	Impossib.(S)	22/56	$2^{373.48}$	$2^{92.22}$	$2^{147.22}$	[42]
SKINNY-128-384	DS-MITM.(S)	22/56	$2^{382.46}$	2^{96}	$2^{330.99}$	[38]

Other previous works discussed distinguishers only, without converting them into attacks. We summarize them in Table 12.

Table 12: Probabilities of the main previous distinguishers of SKINNY-64-192, SKINNY-128-256 and SKINNY-128-384. The letters indicate if it is in the Related (R) or Single (S) tweakey scenario.

Version	Type of distinguisher	Rounds	Probability	ref.
SKINNY-64-192	Boomerang (R)	22/40	$2^{-42.98}$	[39]
SKINNY-64-192	Differential (S)	20/40	$2^{-176.74}$	[11]
SKINNY-64	Truncated (S)	10/40	2^{-40}	[33]
SKINNY-64	Integral (S)	10/40	n/a	[46]
SKINNY-64	zero-correlation (S)	10/40	n/a	[37]
SKINNY-128-256	Boomerang (R)	18/48	$2^{-77.83}$	[39]
SKINNY-128	zero-correlation (S)	10/48	n/a	[37]
SKINNY-128-384	Boomerang (R)	22/56	$2^{-48.30}$	[39]
SKINNY-128	zero-correlation (S)	10/56	n/a	[37]

D.2 Differential and Linear analysis

Arguments in favor of the resistance of ForkSkinny to differential [18] and linear [31] crypanalysis can easily be deduced from the analysis that has been done on SKINNY. First, we recall in Table 13 the bounds on the number of active Sboxes that were provided in SKINNY specification document. These bounds were later refined, and for instance Abdelkhalek et al. [7] showed that in the single key scenario there are no differential characteristics of probability higher than 2^{-128} for 14 rounds or more of SKINNY-128.

These previous results transfer to the case where we look at a trail covering the path from the input message up to C_1. Due to the change in the tweakey schedule we expect different bounds in the related-tweakey for the path from the input message up to C_0. A rough estimate of the minimal number of active Sboxes on this trail can be obtained by summing the bound on r_{init} rounds and the bound on r_0 rounds. For ForkSkinny-64-192 in the TK3 setting, the 17 rounds of r_{init} ensure 31 active Sboxes, so already before the forking point the probability of the characteristic is close to 2^{-64}. For ForkSkinny-128-192 and ForkSkinny-128-256 (both in TK2 model), 21 rounds activate at least 59 Sboxes. If we consider that the branch starting from the forking point is independent and can start from any internal state difference and tweakey difference (this is the very pessimistic case), only 8 rounds after forking are necessary to go below the characteristic probability of 2^{-128}. In a similar way 25 rounds in the TK3 model ensure 60 Sboxes, and already 10 rounds after forking give a characteristic probability under 2^{-128}.

The last case that needs to be evaluated is the reconstruction scenario. An estimate can be computed following the same idea as before: the number of active Sboxes can be upper bounded by the bound obtained by summing the one for r_0 rounds and the one for

Table 13: Lower bounds on the number of active Sboxes in SKINNY, in the single key (SK) and Related-tweakey (TK1, TK2 and TK3) models, as given in [14].

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
SK	1	2	5	8	12	16	26	36	41	46	51	55	58	61	66
TK1	0	0	1	2	3	6	10	13	16	23	32	38	41	45	49
TK2	0	0	0	0	1	2	3	6	9	12	16	21	25	31	35
TK3	0	0	0	0	0	0	1	2	3	6	10	13	16	19	24

	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
SK	75	82	88	92	96	102	108	(114)	(116)	(124)	(132)	(138)	(136)	(148)	(158)
TK1	54	59	62	66	70	75	79	83	85	88	95	102	(108)	(112)	(120)
TK2	40	43	47	52	57	59	64	67	72	75	82	85	88	92	96
TK3	27	31	35	43	45	48	51	55	58	60	65	72	77	81	85

r_1 rounds. If we consider that $r_0 = r_1$ as for our concrete instances, we obtain that 32 Sboxes are active after 12 rounds ($r_0 = r_1 = 6$) in the single key setting, while 26 rounds are required in TK3 ($r_0 = r_1 = 13$) for ForkSkinny-64-192. For the 128-bit block versions, in the single key setting, 16 rounds are required to get more than 64 active Sboxes. For ForkSkinny-128-192 and ForkSkinny-128-256, in TK2 model, 30 rounds are required to get more than 64 active Sboxes. For ForkSkinny-128-288 we need 36 rounds (TK3).

With respect to the parameters we chose, these (optimistic for the attacker) evaluations make us believe that the differential attacks pose no threat to our proposal.

Similar arguments lead to the same conclusion for linear attacks. Also, we refer to the FSE 2017 paper [28] by Kranz et al. that looks at the linear hull of a tweakable block cipher and shows that the addition of a tweak does not introduce new linear characteristics, so that no additional precaution should be taken in comparison to a key-only cipher.

Observations on the differentials. One of the disadvantages of the forking structure comes from the improved differentials one can observe in the reconstruction scenario in comparison to the normal cipher. If there exists a differential going from a difference of δ_{in} to δ_{out} over s rounds of encryption, the probability of the differential going from δ_{out} to δ_{out} over $2s$ rounds in the reconstruction scenario is at least equal to the square of the probability (this is Proposition 2 in [12]), while a smallest probability would have been expected for a *"common"* cipher with a good diffusion.

We also recall here that the clustering effect is rather important in SKINNY [11] and that it might widen the gap between what is expected from the number of active Sboxes analysis and what is observed for ForkSkinny in the reconstruction scenario.

D.3 Impossible Differential

Impossible differential attacks [17,27] make use of a couple of differences (α, β) that verifies that for all possible keys two messages with a Xor difference equal to α cannot produce two messages that differ by β after a given number of rounds r of encryption.

To turn this distinguisher into a key recovery, an attacker appends some rounds before and after the impossible differential. She then makes a guess on the value of some key bits to check if the differences α and β are observed together. If this is the case, the guess is wrong for sure (since it leads to a situation that is impossible), so the corresponding keys are discarded. Once the search space has been sufficiently reduced, the attack is usually finalised with an exhaustive search.

In case the impossible differential is of the truncated type, we can easily give an upper bound on its number of rounds. This study was provided in the `SKINNY` specification, where it was shown that a miss-in-the-middle (in the special case where the contradiction is that one cell is active for sure from one direction but inactive from the other direction) can at most reach 11 rounds in the single-tweakey model.

In following works, the study was extended to the related-tweakey scenario, and for this the number of rounds covered by the distinguisher was extended to 12 rounds for TK1, 14 rounds for TK2 and 16 rounds for TK3 [30].

What remains to be done is the study of the case where the impossible differential is positioned around the forking point. A good first estimate consists in looking at the single key truncated impossible differential case, where the contradiction comes from one active cell obtained from one direction and one inactive cell coming from the other direction. We start by looking for the maximum number of rounds for which one word at least remains inactive or active, both for the cases:

1. decryption rounds only (corresponding to going from C_0 or C_1 up to before the forking point)

2. decryption rounds followed by encryption rounds (corresponding to going from C_0 or C_1 and decrypting and then continuing over the forking point with encryption.)

To evaluate the second case, we look at all the possibilities for the number of rounds before the forking point.

The results are provided in Table 14. If we leave out the necessary requirement that the position of the active cell of one path has to correspond to the position of the inactive path of the other cell, we obtain that no truncated impossible differential can cover more than $7 + 5 = 6 + 6 = 12$ rounds.

Table 14: Maximum number of rounds covered with a truncated differential path until we lose all information.

information	case 1	case 2
inactive	5	6
active	6	7

Since this approximation (that is optimistic for the attacker) is close to what was

obtained for SKINNY (and that SKINNY has comfortable security margins), we are confident about the resistance of ForkSkinny against this type of attacks.

In the related tweakey scenario, an attacker can easily increase the number of rounds of the distinguisher by creating blank rounds (that is with no differences at all), simply by choosing carefully the value of the tweakey difference. However, this trick is limited by the properties of SKINNY Tweakey Schedule, namely the $p-1$ cancellation property of [25]: only a single difference cancellation can happen every 15 rounds for TK2, and only two difference cancellations can happen for TK3. Since only half of the tweakey material is used every round this implies that at most 3 consecutive rounds with no tweakey differences can be constructed every 30 rounds for TK2, and 5 for TK3. Even in the case where these free rounds can be exploited both at the beginning and at the end, the securiy margins chosen in SKINNY are sufficient.

D.4 Boomerang Attack

In the classical boomerang attack [43] the adversary produces a quartet of plaintexts/ciphertexts $\{(P_i)\}_{i=0}^{4}$ such that $\bigoplus P_i = 0$, satisfying $\bigoplus E(P_i) = 0$, where E is typically a block cipher. Boomerang attack can also be adapted in the related-key model, which is known as the related-key boomerang attack. The success of classic boomerang attack depends on the probability of differential propagation in a block cipher. Usually a boomerang attack combines two high probability differentials which exist on reduced number of rounds. Suppose that in a block cipher two differentials exist with probabilities p and q on r_1 and r_2 round respectively. Then the probability of the boomerang distinguisher for $r_1 + r_2$ rounds of $E_{r_2} \circ E_{r_1}$ is p^2q^2, where E_r denotes the r round of the encryption function E. In ForkSkinny such attack can not be applied for the full round due to the large number of active Sboxes. For ForkSkinny the related-key boomerang attack is more relevant, since it may lead to a forgery attack against the AE scheme. In ForkSkinny, we can always find a difference between the round-tweakeys (immediately after the forking step) which are used in the two different branches of the forkcipher. Using such related round-tweakeys if an adversary can find RTK boomerang attack then it will lead to the forgery of the AE scheme. The idea of such attack is depicted in the Fig 15. Such an attack [19] was also found on an earlier forkcipher instantation. However, it is not possible to find a similar boomerang attack on ForkSkinny which may lead to forgery attack.

D.5 Meet-in-the-Middle Attack

In a (basic) Meet-in-the-Middle attack, the attacker looks for a decomposition of the cipher in two parts so that the computation of each part only requires a fraction of the master key. She then computes a part of the internal state from the plaintext up to the end of the first part of the cipher, and computes the same part from the ciphertext up to the beginning of the second part. The correct value for the guessed key bits is among the hypotheses that lead to a match.

A good starting point to obtain a first approximation of the resistance of a cipher to Meet-in-the-Middle attacks consists in looking at its diffusion.

The diffusion of a cipher corresponds to the number of rounds d that are required for any input bit to influence all the bits of the internal state. In case the key size corresponds to the block size and that all the key material is used in every rounds, having a cipher with diffusion equal to d means that any output bit after d rounds is an expression depending on all the key bits, which prevents the previous MitM attacks when more than $(d-1)+(d-1)$ rounds are used.

For SKINNY the diffusion delay is equal to 6, which would lead to a first estimate of 10 rounds for a partial matching. However, we must take into account the fact that only half of the tweakey material is used in each round, that the key addition is made after the non-linear operation and that the forking point in a reconstruction operation has a lower diffusion[2]), which adds some rounds to the first estimate.

In any case the obtained numbers are far from the chosen number of rounds. Moreover, recent results by Shi et al. [38] showed that with the improvements resulting from the Demirci-Selçuk techniques a total of 22 rounds out of the 56 of SKINNY-128-384 can be attacked. This supports that the number of rounds we chose are sufficient to thwart these types of attacks.

D.6 Integral Attack

ForkSkinny has two components ForkSkinny_0 and ForkSkinny_1 which produce C_0 and C_1, respectively from M. The security of these components follow directly from the analysis of SKINNY. The integral cryptanalysis against SKINNY can be directly applied to ForkSkinny_0 and ForkSkinny_1. SKINNY specification describes an integral distinguisher for 10 rounds. This can be applied to both reduced round ForkSkinny_0 and ForkSkinny_1. When applied to these components, the integral distinguisher can only cover less than r_{init} rounds prior to forking step. For the key recovery attack, it is possible to add 4 rounds to this integral distinguisher which allows an adversary to mount an attack against 14 rounds of SKINNY. Again, this key recovery attack can only cover less than r_{init} rounds, prior to forking in different ForkSkinny-n-t. In the reconstruction, an adversary has to cover at least 27 rounds in the encryption direction (following the forking point). Hence, it is not possible to use the integral attack against the full reconstruction in ForkSkinny. Complexities of the integral attacks against round reduced ForkSkinny remain the same as described in the specification of SKINNY [14].

Division Property The division property was introduced as a generalization of the integral property by Todo [41]. SKINNY specification analyses show that the division property has significant margin against an attack that uses it. The generic analysis of SPN ciphers described in [41] leads to only 6 round of division property. Taking the resistance

[2]The diffusion delay could also increase if the forking point chains two tweakeys that depend on the same half of the tweakey material. To avoid this we opted for values of r_1 that are odd.

of SKINNY against division property into account, we are confident that ForkSkinny has sufficient security margin against the same type of attacks.

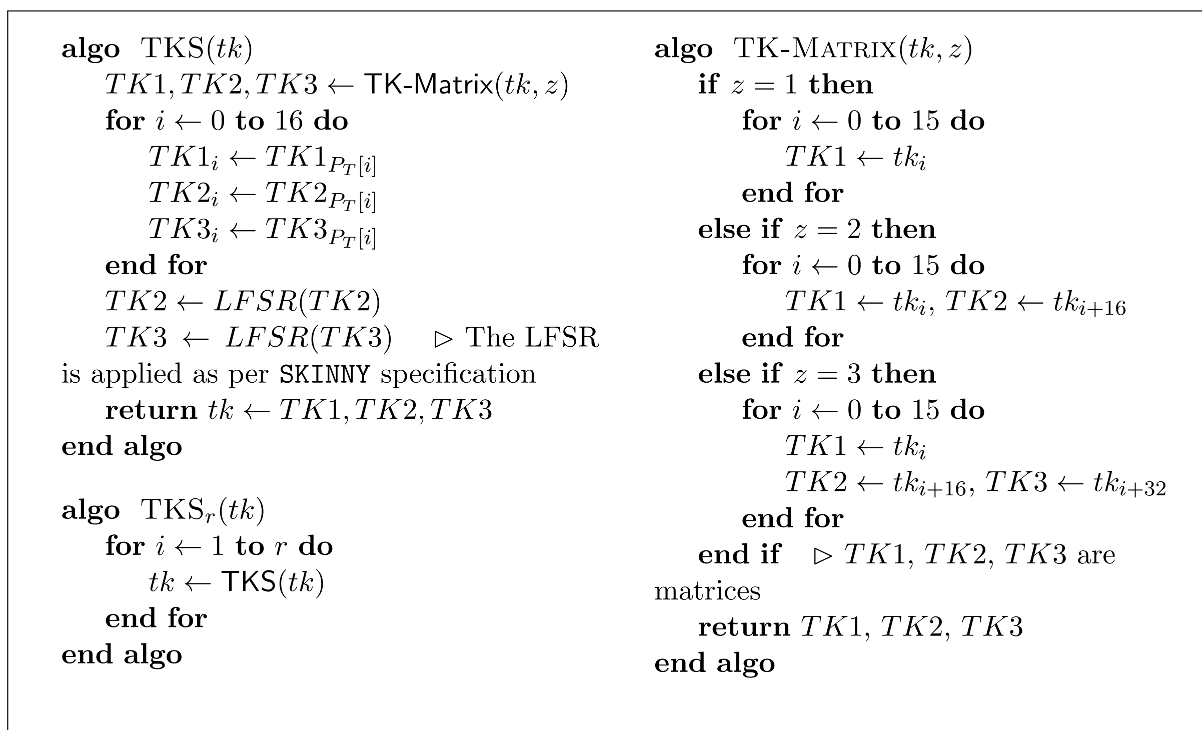
D.7 Algebraic Attack

ForkSkinny is resilient against algebraic attacks. Following the analysis of SKINNY we can reason that algebraic attacks do not apply against full ForkSkinny. ForkSkinny uses the same Sboxes of sizes 4 bits and 8 bits with algebraic degree $a = 3$ and $a = 6$, respectively, as in SKINNY. In a single key setting, consecutive 7 rounds of SKINNY encryption and decryption have 26 and 27 active Sboxes respectively. For all variants of r rounds ForkSkinny, we obtain $a \cdot 26 \cdot \lfloor \frac{r}{7} \rfloor \ggg n$. Like SKINNY, ForkSkinny is also described by a large number of quadratic equations which contains a large number of variables. Overall, we conclude that ForkSkinny offers the same security as SKINNY against algebraic attacks.

E The Sboxes of SKINNY

```
/* SKINNY-64 Sbox */
const unsigned char S4[16] = {12,6,9,0,1,10,2,11,3,8,5,13,4,14,7,15};

/* SKINNY-128 Sbox */
uint8_t S8 [256] = {
0x65, 0x4c, 0x6a, 0x42, 0x4b, 0x63, 0x43, 0x6b, 0x55, 0x75, 0x5a, 0x7a, 0x53, 0x73, 0x5b, 0x7b,
0x35, 0x8c, 0x3a, 0x81, 0x89, 0x33, 0x80, 0x3b, 0x95, 0x25, 0x98, 0x2a, 0x90, 0x23, 0x99, 0x2b,
0xe5, 0xcc, 0xe8, 0xc1, 0xc9, 0xe0, 0xc0, 0xe9, 0xd5, 0xf5, 0xd8, 0xf8, 0xd0, 0xf0, 0xd9, 0xf9,
0xa5, 0x1c, 0xa8, 0x12, 0x1b, 0xa0, 0x13, 0xa9, 0x05, 0xb5, 0x0a, 0xb8, 0x03, 0xb0, 0x0b, 0xb9,
0x32, 0x88, 0x3c, 0x85, 0x8d, 0x34, 0x84, 0x3d, 0x91, 0x22, 0x9c, 0x2c, 0x94, 0x24, 0x9d, 0x2d,
0x62, 0x4a, 0x6c, 0x45, 0x4d, 0x64, 0x44, 0x6d, 0x52, 0x72, 0x5c, 0x7c, 0x54, 0x74, 0x5d, 0x7d,
0xa1, 0x1a, 0xac, 0x15, 0x1d, 0xa4, 0x14, 0xad, 0x02, 0xb1, 0x0c, 0xbc, 0x04, 0xb4, 0x0d, 0xbd,
0xe1, 0xc8, 0xec, 0xc5, 0xcd, 0xe4, 0xc4, 0xed, 0xd1, 0xf1, 0xdc, 0xfc, 0xd4, 0xf4, 0xdd, 0xfd,
0x36, 0x8e, 0x38, 0x82, 0x8b, 0x30, 0x83, 0x39, 0x96, 0x26, 0x9a, 0x28, 0x93, 0x20, 0x9b, 0x29,
0x66, 0x4e, 0x68, 0x41, 0x49, 0x60, 0x40, 0x69, 0x56, 0x76, 0x58, 0x78, 0x50, 0x70, 0x59, 0x79,
0xa6, 0x1e, 0xaa, 0x11, 0x19, 0xa3, 0x10, 0xab, 0x06, 0xb6, 0x08, 0xba, 0x00, 0xb3, 0x09, 0xbb,
0xe6, 0xce, 0xea, 0xc2, 0xcb, 0xe3, 0xc3, 0xeb, 0xd6, 0xf6, 0xda, 0xfa, 0xd3, 0xf3, 0xdb, 0xfb,
0x31, 0x8a, 0x3e, 0x86, 0x8f, 0x37, 0x87, 0x3f, 0x92, 0x21, 0x9e, 0x2e, 0x97, 0x27, 0x9f, 0x2f,
0x61, 0x48, 0x6e, 0x46, 0x4f, 0x67, 0x47, 0x6f, 0x51, 0x71, 0x5e, 0x7e, 0x57, 0x77, 0x5f, 0x7f,
0xa2, 0x18, 0xae, 0x16, 0x1f, 0xa7, 0x17, 0xaf, 0x01, 0xb2, 0x0e, 0xbe, 0x07, 0xb7, 0x0f, 0xbf,
0xe2, 0xca, 0xee, 0xc6, 0xcf, 0xe7, 0xc7, 0xef, 0xd2, 0xf2, 0xde, 0xfe, 0xd7, 0xf7, 0xdf, 0xff
};
```



algo TKS(tk)
  $TK1, TK2, TK3 \leftarrow$ TK-Matrix(tk, z)
  for $i \leftarrow 0$ **to** 16 **do**
    $TK1_i \leftarrow TK1_{P_T[i]}$
    $TK2_i \leftarrow TK2_{P_T[i]}$
    $TK3_i \leftarrow TK3_{P_T[i]}$
  end for
  $TK2 \leftarrow LFSR(TK2)$
  $TK3 \leftarrow LFSR(TK3)$ ▷ The LFSR is applied as per SKINNY specification
  return $tk \leftarrow TK1, TK2, TK3$
end algo

algo TKS$_r$(tk)
  for $i \leftarrow 1$ **to** r **do**
    $tk \leftarrow$ TKS(tk)
  end for
end algo

algo TK-Matrix(tk, z)
  if $z = 1$ **then**
    for $i \leftarrow 0$ **to** 15 **do**
      $TK1 \leftarrow tk_i$
    end for
  else if $z = 2$ **then**
    for $i \leftarrow 0$ **to** 15 **do**
      $TK1 \leftarrow tk_i$, $TK2 \leftarrow tk_{i+16}$
    end for
  else if $z = 3$ **then**
    for $i \leftarrow 0$ **to** 15 **do**
      $TK1 \leftarrow tk_i$
      $TK2 \leftarrow tk_{i+16}$, $TK3 \leftarrow tk_{i+32}$
    end for
  end if   ▷ $TK1$, $TK2$, $TK3$ are matrices
  return $TK1$, $TK2$, $TK3$
end algo

Figure 3: The tweakey scheduling algorithms for ForkSkinny.

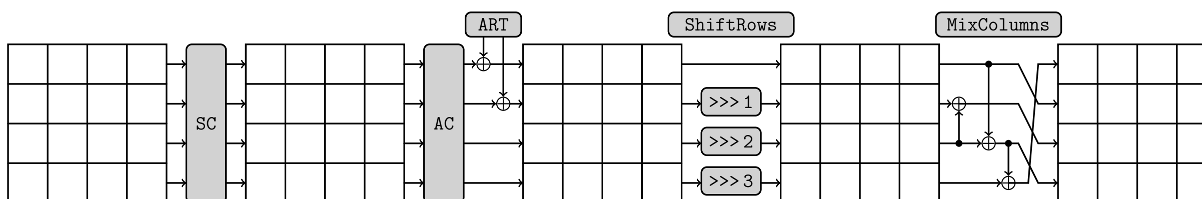


Figure 4: Structure of every round in ForkSkinny, made of the five operations SubCells (SC), AddConstants (AC), AddRoundTweakey (ART), ShiftRows (SR) and MixColumns (MC), as it is done in SKINNY. (Figure credits: [21]).

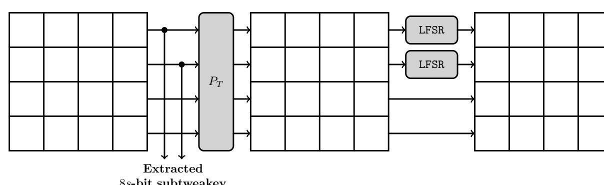


Figure 5: Tweakey schedule of ForkSkinny, replicating the one of SKINNY. (Figure credits: [21])

1: **algo** $\mathcal{E}(K, N, A, M)$
2: $A_1, \ldots, A_a, A_* \xleftarrow{n} A$
3: $M_1, \ldots, M_m, M_* \xleftarrow{n} M$
4: $S \leftarrow 0^n$; $c \leftarrow (\tau - \nu - 3)$
5: **for** $i \leftarrow 1$ **to** a **do**
6: $\mathsf{T} \leftarrow N \| 000 \| \langle i \rangle$
7: $S \leftarrow S \oplus \mathsf{F}_K^{\mathsf{T},1}(A_i)$
8: **end for**
9: **if** $|A_*| = n$ **then**
10: $\mathsf{T} \leftarrow N \| 001 \| \langle a+1 \rangle_c$
11: $S \leftarrow S \oplus \mathsf{F}_K^{\mathsf{T},1}(A_*)$
12: **else if** $|A_*| > 0$ **or** $|M| = 0$ **then**
13: $\mathsf{T} \leftarrow N \| 011 \| \langle a+1 \rangle_c$
14: $S \leftarrow S \oplus \mathsf{F}_K^{\mathsf{T},1}(A_* \| 10^*)$
15: **end if** ▷ Do nothing if $A = \varepsilon, M \neq \varepsilon$
16: **for** $i \leftarrow 1$ **to** m **do**
17: $\mathsf{T} \leftarrow N \| 100 \| \langle i \rangle_c$
18: $C_i, S' \leftarrow \mathsf{F}_K^{\mathsf{T},\mathsf{b}}(M_i)$
19: $S \leftarrow S \oplus S'$
20: **end for**
21: **if** $|M_*| = n$ **then**
22: $\mathsf{T} \leftarrow N \| 101 \| \langle m+1 \rangle_c$
23: **else if** $|M_*| > 0$ **then**
24: $\mathsf{T} \leftarrow N \| 111 \| \langle m+1 \rangle_c$
25: **else**
26: **return** S
27: **end if**
28: $C_*, T \leftarrow \mathsf{F}_K^{\mathsf{T},\mathsf{b}}(\mathsf{pad10}(M_*))$
29: $C_* \leftarrow C_* \oplus S$
30: **return** $C_1 \| \ldots \| C_m \| C_* \| \mathsf{left}_{|M_*|}(T)$
31: **end algo**

1: **algo** $\mathcal{D}(K, N, A, C)$
2: $A_1, \ldots, A_a, A_* \xleftarrow{n} A$
3: $C_1, \ldots, C_m, C_*, T \leftarrow \mathsf{csplit\text{-}b}_n(C)$
4: $S \leftarrow 0^n$; $c \leftarrow (\tau - \nu - 3)$
5: **for** $i \leftarrow 1$ **to** a **do**
6: $\mathsf{T} \leftarrow N \| 000 \| \langle i \rangle_c$
7: $S \leftarrow S \oplus \mathsf{F}_K^{\mathsf{T},1}(A_i)$
8: **end for**
9: **if** $|A_*| = n$ **then**
10: $\mathsf{T} \leftarrow N \| 001 \| \langle a+1 \rangle_c$
11: $S \leftarrow S \oplus \mathsf{F}_K^{\mathsf{T},1}(A_*)$
12: **else if** $|A_*| > 0$ **or** $|T| = 0$ **then**
13: $\mathsf{T} \leftarrow N \| 011 \| \langle a+1 \rangle_c$
14: $S \leftarrow S \oplus \mathsf{F}_K^{\mathsf{T},1}(A_* \| 10^*)$
15: **end if** ▷ Do nothing if $A = \varepsilon, M \neq \varepsilon$
16: **for** $i \leftarrow 1$ **to** m **do**
17: $\mathsf{T} \leftarrow N \| 100 \| \langle i \rangle_c$
18: $M_i, S' \leftarrow \mathsf{F^{-1}}_K^{\mathsf{T},0,\mathsf{b}}(C_i)$
19: $S \leftarrow S \oplus S'$
20: **end for**
21: **if** $|T| = n$ **then**
22: $\mathsf{T} \leftarrow N \| 101 \| \langle m+1 \rangle_c$
23: **else if** $|T| > 0$ **then**
24: $\mathsf{T} \leftarrow N \| 111 \| \langle m+1 \rangle_c$
25: **else**
26: **if** $C_* \neq S$ **then return** $\perp$
27: **return** ε
28: **end if**
29: $M_*, T' \leftarrow \mathsf{F^{-1}}_K^{\mathsf{T},0,\mathsf{b}}(C_* \oplus S)$
30: $T' \leftarrow \mathsf{left}_{|T|}(T')$; $P \leftarrow \mathsf{right}_{n-|T|}(M_*)$
31: **if** $T' \neq T$ **return** $\perp$
32: **if** $P \neq \mathsf{left}_{n-|T|}(10^{n-1})$ **return** $\perp$
33: **return** $M_1 \| \ldots \| M_m \| \mathsf{left}_{|T|}(M_*)$
34: **end algo**

Figure 6: The PAEF[F, ν] AEAD scheme. Here $\langle i \rangle_\ell$ is the cannonical encoding of an integer i as an ℓ-bit string.

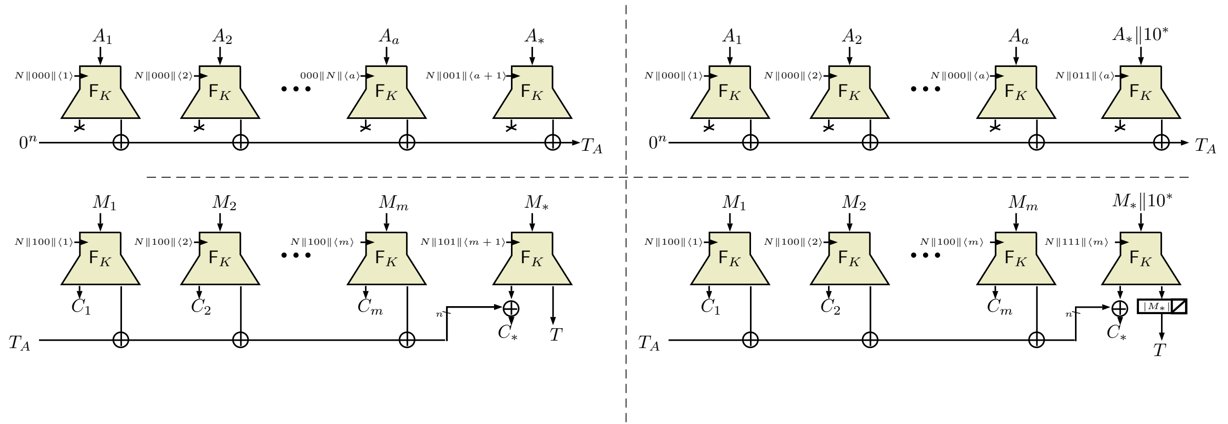


Figure 7: The encryption algorithm of PAEF[F] mode. The picture illustrates the processing of AD when length of AD is a multiple of n (**top left**) and when the length of AD is not a multiple of n (**top right**), and the processing of the message when length of the message is a multiple of n (**bottom left**) and when the length of message is not a multiple of n (**bottom right**). The tweak inputs to the forkcipher F are of the form $\mathsf{T} = N\|f\|\mathsf{ctr}$, where $f \in \{0,1\}^3$ is a three-bit domain separation flag and $\mathsf{ctr} = \langle i \rangle$ is a $(\tau - |N| - 3)$-bit encoding of the block counter.

1: **algo** $\mathcal{E}(K, N, A, M)$
2: $A_1, \ldots, A_a, A_* \xleftarrow{n} A$
3: $M_1, \ldots, M_m, M_* \xleftarrow{n} M$
4: $\mathsf{noM} \leftarrow 0$
5: **if** $|M| = 0$ **then** $\mathsf{noM} \leftarrow 1$
6: $\Delta \leftarrow 0^n$; $\mathsf{T} \leftarrow N \| 0^{\tau-4-\nu} \| 1$
7: **for** $i \leftarrow 1$ **to** a **do**
8: $\mathsf{T} \leftarrow \mathsf{T} \| 000$
9: $\Delta \leftarrow \mathsf{F}_K^{\mathsf{T},1}(A_i \oplus \Delta)$
10: $\mathsf{T} \leftarrow 0^{\tau-3}$
11: **end for**
12: **if** $|A_*| = n$ **then**
13: $\mathsf{T} \leftarrow \mathsf{T} \| \mathsf{noM} \| 10$
14: $\Delta \leftarrow \mathsf{F}_K^{\mathsf{T},1}(A_* \oplus \Delta)$
15: $\mathsf{T} \leftarrow 0^{\tau-3}$
16: **else if** $|A_*| > 0$ **or** $|M| = 0$ **then**
17: $\mathsf{T} \leftarrow \mathsf{T} \| \mathsf{noM} \| 11$
18: $\Delta \leftarrow \mathsf{F}_K^{\mathsf{T},1}((A_* \| 10^*) \oplus \Delta)$
19: $\mathsf{T} \leftarrow 0^{\tau-3}$
20: **end if** ▷ Do nothing if $A = \varepsilon, M \neq \varepsilon$
21: **for** $i \leftarrow 1$ **to** m **do**
22: $\mathsf{T} \leftarrow \mathsf{T} \| 001$
23: $C_i, \Delta \leftarrow \mathsf{F}_K^{\mathsf{T},\mathsf{b}}(M_i \oplus \Delta) \oplus (\Delta, 0^n)$
24: $\mathsf{T} \leftarrow 0^{\tau-3}$
25: **end for**
26: **if** $|M_*| = n$ **then**
27: $\mathsf{T} \leftarrow \mathsf{T} \| 100$
28: **else if** $|M_*| > 0$ **then**
29: $\mathsf{T} \leftarrow \mathsf{T} \| 101$
30: **else**
31: **return** Δ
32: **end if**
33: $C_*, T \leftarrow \mathsf{F}_K^{\mathsf{T},\mathsf{b}}(\mathsf{pad10}(M_*) \oplus \Delta) \oplus (\Delta \| 0^n)$
34: **return** $C_1 \| \ldots \| C_m \| C_* \| \mathsf{left}_{|M_*|}(T)$
35: **end algo**

1: **algo** $\mathcal{D}(K, N, A, C)$
2: $A_1, \ldots, A_a, A_* \xleftarrow{n} A$
3: $C_1, \ldots, C_m, C_*, T \leftarrow \mathsf{csplit\text{-}b}_n C$
4: $\mathsf{noM} \leftarrow 0$
5: **if** $|C| = n$ **then** $\mathsf{noM} \leftarrow 1$
6: $\Delta \leftarrow 0^n$; $\mathsf{T} \leftarrow N \| 0^{\tau-4-\nu} \| 1$
7: **for** $i \leftarrow 1$ **to** a **do**
8: $\mathsf{T} \leftarrow \mathsf{T} \| 000$
9: $\Delta \leftarrow \mathsf{F}_K^{\mathsf{T},1}(A_i \oplus \Delta)$
10: $\mathsf{T} \leftarrow 0^{\tau-3}$
11: **end for**
12: **if** $|A_*| = n$ **then**
13: $\mathsf{T} \leftarrow \mathsf{T} \| \mathsf{noM} \| 10$
14: $\Delta \leftarrow \mathsf{F}_K^{\mathsf{T},1}(A_* \oplus \Delta)$
15: $\mathsf{T} \leftarrow 0^{\tau-3}$
16: **else if** $|A_*| > 0$ **or** $|T| = 0$ **then**
17: $\mathsf{T} \leftarrow \mathsf{T} \| \mathsf{noM} \| 11$
18: $\Delta \leftarrow \mathsf{F}_K^{\mathsf{T},1}((A_* \| 10^*) \oplus \Delta)$
19: $\mathsf{T} \leftarrow 0^{\tau-3}$
20: **end if** ▷ Do nothing if $A = \varepsilon, M \neq \varepsilon$
21: **for** $i \leftarrow 1$ **to** m **do**
22: $\mathsf{T} \leftarrow \mathsf{T} \| 001$
23: $M_i, \Delta \leftarrow \mathsf{F}^{-1}{}_K^{\mathsf{T},0,\mathsf{b}}(C_i \oplus \Delta) \oplus (\Delta, 0^n)$
24: $\mathsf{T} \leftarrow 0^{\tau-3}$
25: **end for**
26: **if** $|T| = n$ **then**
27: $\mathsf{T} \leftarrow \mathsf{T} \| 100$
28: **else if** $|T| > 0$ **then**
29: $\mathsf{T} \leftarrow \mathsf{T} \| 101$
30: **else**
31: **if** $C_* \neq \Delta$ **then return** $\perp$
32: **return** ε
33: **end if**
34: $M_*, T' \leftarrow \mathsf{F}^{-1}{}_K^{\mathsf{T},0,\mathsf{b}}(C_* \oplus \Delta) \oplus (\Delta, 0^n)$
35: $T' \leftarrow \mathsf{left}_{|T|}(T')$; $P \leftarrow \mathsf{right}_{n-|T|}(M_*)$
36: **if** $T' \neq T$ **return** $\perp$
37: **if** $P \neq \mathsf{left}_{n-|T|}(10^{n-1})$ **return** $\perp$
38: **return** $M_1 \| \ldots \| M_m \| \mathsf{left}_{|T|}(M_*)$
39: **end algo**

Figure 8: The SAEF[F] AEAD scheme.

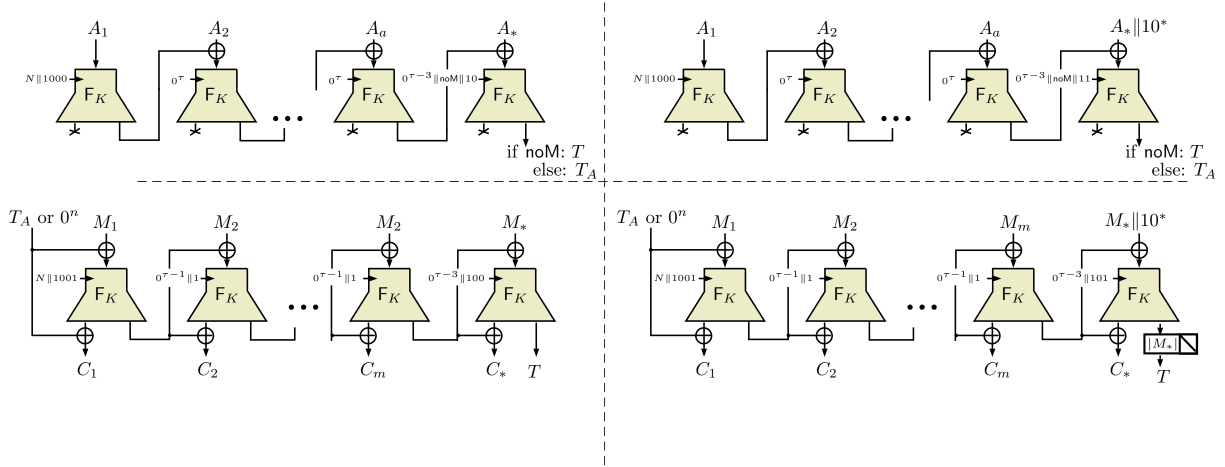


Figure 9: The encryption algorithm of SAEF[F] mode when $|N| = \tau - 4$. The bit $\mathsf{noM} = 1$ iff $|M| = 0$. The picture illustrates the processing of AD when length of AD is a multiple of n (**top left**) and when the length of AD is not a multiple of n (**top right**), and the processing of the message when length of the message is a multiple of n (**bottom left**) and when the length of message is not a multiple of n (**bottom right**). The tweak inputs to the forkcipher F are of the form $\mathsf{T} = N\|0^{\tau-|N|-4}\|1\|f$ or $\mathsf{T} = 0^{\tau-4}\|0\|f$, where $f \in \{0,1\}^3$ is a three-bit domain separation flag (The zero padding of the nonce is not present in the figure as $|N| = \tau - 4$).

Game **prtfp-real**$_{\mathsf{F}}$	Game **prtfp-ideal**$_{\mathsf{F}}$
$K \leftarrow_\$ \{0,1\}^k$ $b \leftarrow \mathcal{A}^{\text{Enc},\text{Dec}}$ **return** b	**for** $\mathsf{T} \in \mathcal{T}$ **do** $\pi_{\mathsf{T},0}, \pi_{\mathsf{T},1} \leftarrow_\$ \text{Perm}n$ $b \leftarrow \mathcal{A}^{\text{Enc},\text{Dec}}$ **return** b
Oracle $\text{Enc}(\mathsf{T}, M, s)$ **return** $\mathsf{F}(K, \mathsf{T}, M, s)$	**Oracle** $\text{Enc}(\mathsf{T}, M, s)$ **if** $s = 0$ **then return** $\pi_{\mathsf{T},0}(M)$ **if** $s = 1$ **then return** $\pi_{\mathsf{T},1}(M)$ **if** $s = \mathsf{b}$ **then return** $\pi_{\mathsf{T},0}(M), \pi_{\mathsf{T},1}(M)$
Oracle $\text{Dec}(\mathsf{T}, C, \beta, s)$ **return** $\mathsf{F}^{-1}(K, \mathsf{T}, C, \beta, s)$	**Oracle** $\text{Dec}(\mathsf{T}, C, \beta, s)$ **if** $s = \mathsf{i}$ **then return** $\pi^{-1}_{\mathsf{T},\beta}(C)$ **if** $s = \mathsf{o}$ **then return** $\pi_{\mathsf{T},(\beta\oplus 1)}(\pi^{-1}_{\mathsf{T},\beta}(C))$ **if** $s = \mathsf{b}$ **then return** $\pi^{-1}_{\mathsf{T},\beta}(C), \pi_{\mathsf{T},(\beta\oplus 1)}(\pi^{-1}_{\mathsf{T},\beta}(C))$

Figure 10: Games **prtfp-real** and **prtfp-ideal** defining the security of a (strong) forkcipher.

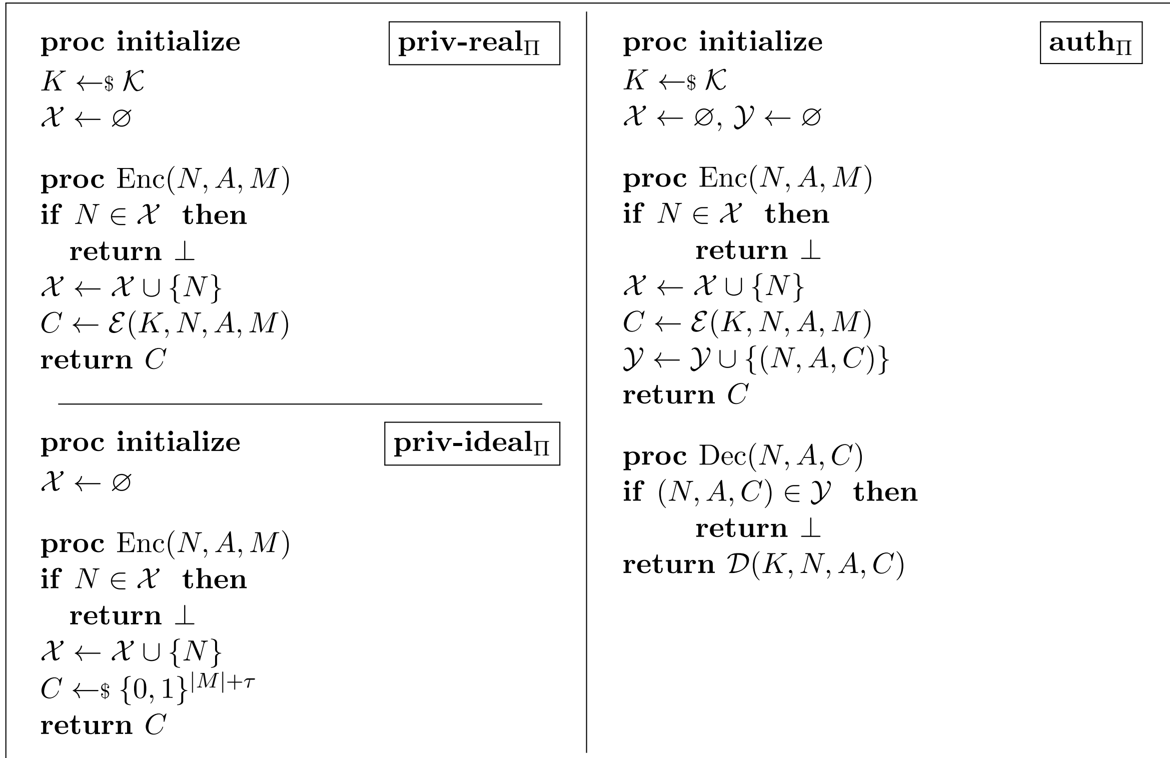


Figure 11: Security games for a nonce-based AE $\Pi = (\mathcal{K}, \mathcal{E}, \mathcal{D})$ with ciphertext expansion τ.

```
1:  proc initialize
2:      for T ∈ {0,1}^τ do
3:          f_{T,0} ←$ Func(n)
4:          f_{T,1} ←$ Func(n)
5:          D_T ← ∅
6:      end for
7:      bad ← false

1:  proc Enc(N, A, M)
2:      A_1, ..., A_a, A_* ←n− A
3:      M_1, ..., M_m, M_* ←n− M
4:      W ← (N, A)
5:      noM ← 0
6:      if |M| = 0 then noM ← 1
7:      Δ ← 0^n; T ← N‖0^{τ−ν−4}‖1
8:      for i ← 1 to a do
9:          T ← T‖000
10:         if A_i ⊕ Δ ∈ D_T then
11:             bad ← true
12:         end if
13:         D_T ← D_T ∪ (A_i ⊕ Δ)
14:         Δ ← f_{T,1}(A_i ⊕ Δ)
15:         if bad = true then
16:             [Δ ←$ {0,1}^n]
17:         end if
18:         T ← 0^{τ−3}
19:     end for
20:     if |A_*| = n then
21:         T ← T‖noM‖10
22:         if A_* ⊕ Δ ∈ D_T then
23:             bad ← true
24:         end if
25:         D_T ← D_T ∪ (A_* ⊕ Δ)
26:         Δ ← f_{T,1}(A_* ⊕ Δ)
27:         if bad = true then
28:             [Δ ←$ {0,1}^n]
29:         end if
30:         T ← 0^{τ−3}
31:     else if |A_*| > 0 or |M| = 0 then
32:         T ← T‖noM‖11
33:         if A_* ⊕ Δ ∈ D_T then
34:             bad ← true
35:         end if
36:         D_T ← D_T ∪ ((A_*‖10^*) ⊕ Δ)
37:         Δ ← f_{T,1}((A_*‖10^*) ⊕ Δ)
38:         if bad = true then
39:             [Δ ←$ {0,1}^n]
40:         end if
41:         T ← 0^{τ−3}
42:     end if
43:     for i ← 1 to m do
44:         T ← T‖001
45:         if M_i ⊕ Δ ∈ D_T then
46:             bad ← true
47:         end if
48:         D_T ← D_T ∪ (M_i ⊕ Δ)
49:         C_i ← f_{T,0}(M_i ⊕ Δ) ⊕ Δ
50:         Δ ← f_{T,1}(M_i ⊕ Δ)
51:         if bad = true then
52:             [C_i, Δ ←$ {0,1}^n × {0,1}^n]
53:         end if
54:         T ← 0^{τ−3}
55:     end for
56:     if |M_*| = n then
57:         T ← T‖100
58:     else if |M_*| > 0 then
59:         T ← T‖101
60:     else
61:         return Δ
62:     end if
63:     if M_* ⊕ Δ ∈ D_T then
64:         bad ← true
65:     end if
66:     C_* ← f_{T,0}(pad10(M_*) ⊕ Δ) ⊕ Δ
67:     T ← f_{T,1}(pad10(M_*) ⊕ Δ)
68:     if bad = true then
69:         [C_*, T ←$ {0,1}^n × {0,1}^n]
70:     end if
71:     return C_1‖...‖C_m‖C_*‖left_{|M_*|}(T)
```

Figure 12: The games G_0 and G_1 for bounding $\mathbf{Adv}^{\mathbf{priv}}_{\mathrm{SAEF}[f_0,f_1]}$. The game G_0 does *not* contain the boxed statement, while G_1 does.

```
 1: proc initialize
 2:     for T ∈ {0,1}^τ do
 3:         π_{T,0} ←$ Perm n
 4:         f_{T,1} ←$ Func(n)
 5:         D_T ← ∅
 6:     end for
 7:     for N ∈ {0,1}^ν do Q(N) ← ∅
 8:     bad ← false

 1: proc Enc(N, A, M)
 2:     A_1, ..., A_a, A_* ←n− A
 3:     M_1, ..., M_m, M_* ←n− M
 4:     noM ← 0
 5:     if |M| = 0 then noM ← 1
 6:     Δ ← 0^n; T ← N‖1
 7:     for i ← 1 to a do
 8:         T ← T‖000
 9:         if A_i ⊕ Δ ∈ D_T and P^i_a(W, Q) then
10:             bad ← true
11:         end if
12:         D_T ← D_T ∪ (A_i ⊕ Δ)
13:         Δ ← f_{T,1}(A_i ⊕ Δ)
14:         T ← 0^{τ−3}
15:     end for
16:     if |A_*| = n then
17:         T ← T‖noM‖10
18:         if A_* ⊕ Δ ∈ D_T and P^*_a(W, Q) then
19:             bad ← true
20:         end if
21:         D_T ← D_T ∪ (A_* ⊕ Δ)
22:         Δ ← f_{T,1}(A_* ⊕ Δ)
23:         T ← 0^{τ−3}
24:     else if |A_*| > 0 or |M| = 0 then
25:         T ← T‖noM‖11
26:         if (A_*‖10^*) ⊕ Δ ∈ D_T and P^*_a(W, Q) then
27:             bad ← true
28:         end if
29:         D_T ← D_T ∪ ((A_*‖10^*) ⊕ Δ)
30:         Δ ← f_{T,1}((A_*‖10^*) ⊕ Δ)
31:         T ← 0^{τ−3}
32:     end if
33:     for i ← 1 to m do
34:         T ← T‖001
35:         if M_i ⊕ Δ ∈ D_T then
36:             bad ← true
37:         end if
38:         D_T ← D_T ∪ (M_i ⊕ Δ)
39:         C_i ← π_{T,0}(M_i ⊕ Δ) ⊕ Δ
40:         Δ ← f_{T,1}(M_i ⊕ Δ)
41:         T ← 0^{τ−3}
42:     end for
43:     if |M_*| = n then
44:         T ← T‖100
45:     else if |M_*| > 0 then
46:         T ← T‖101
47:     else
48:         return Δ
49:     end if
50:     if pad10(M_i) ⊕ Δ ∈ D_T then
51:         bad ← true
52:     end if
53:     D_T ← D_T ∪ (pad10(M_i) ⊕ Δ)
54:     C_* ← π_{T,0}(pad10(M_*) ⊕ Δ) ⊕ Δ
55:     T ← f_{T,1}(pad10(M_*) ⊕ Δ)
56:     Q(N) ← Q(N) ∪ ((A_1, .., A_*), (C_1, .., C_m))
57:     return C_1‖...‖C_m‖C_*‖left_{|M_*|}(T)
```

Figure 13: The games G_2 and G_3 for bounding $\mathbf{Adv}^{\mathbf{auth}}_{\mathrm{SAEF}[\pi_0, f_1]}$ (continued in Figure 14). The game G_2 does *not* contain the boxed statements, while G_3 does.

```
1:  proc Dec(N, A, C)
2:      if bad = true then
3:          [return ⊥]
4:      end if
5:      W ← (N, A, C)
6:      Q(N) ← Q(N) ∪ ((A_1, .., A_*))
7:      A_1, ..., A_a, A_* ←^n A
8:      C_1, ..., C_m, C_*, T ← csplit-b_n(C)
9:      noM ← 0
10:     if |C| = n then noM ← 1
11:     Δ ← 0^n; T ← N‖1
12:     for i ← 1 to a do
13:         T ← T‖000
14:         if A_i ⊕ Δ ∈ D_T and P^i_a(W, Q) then
15:             bad ← true
16:             [return ⊥]
17:         end if
18:         D_T ← D_T ∪ (A_i ⊕ Δ)
19:         Δ ← f_{T,1}(A_i ⊕ Δ)
20:         T ← 0^{τ−3}
21:     end for
22:     if |A_*| = n then
23:         T ← T‖noM‖10
24:         if A_* ⊕ Δ ∈ D_T and P^*_a(W, Q) then
25:             bad ← true
26:             [return ⊥]
27:         end if
28:         D_T ← D_T ∪ (A_* ⊕ Δ)
29:         Δ ← f_{T,1}(A_* ⊕ Δ)
30:         T ← 0^{τ−3}
31:     end if
32:     if |A_*| > 0 or |T| = 0 then
33:         T ← T‖noM‖11 and P^*_a(W, Q)
34:         if (A_*‖10^*) ⊕ Δ ∈ D_T and P^*_a(W, Q)
            then
35:             bad ← true
36:             [return ⊥]
37:         end if
38:         D_T ← D_T ∪ ((A_*‖10^*) ⊕ Δ)
39:         Δ ← f_{T,1}((A_*‖10^*) ⊕ Δ)
40:         T ← 0^{τ−3}
41:     end if
42:     for i ← 1 to m do
43:         T ← T‖001
44:         M_i ← π^{−1}_{T,0}(C_i ⊕ Δ, 0) ⊕ Δ
45:         if M_i ⊕ Δ ∈ D_T and P^i_m(W, Q) then
46:             bad ← true
47:             [return ⊥]
48:         end if
49:         D_T ← D_T ∪ (M_i ⊕ Δ)
50:         Δ ← f_{T,1}(π^{−1}_{T,0}(C_i ⊕ Δ, 0))
51:         T ← 0^{τ−3}
52:     end for
53:     if |T| = n then
54:         T ← T‖100
55:     else if |T| > 0 then
56:         T ← T‖100
57:     else
58:         if C_* ≠ Δ then return ⊥
59:         return ε
60:     end if
61:     M_* ← π^{−1}_{T,0}(C_* ⊕ Δ) ⊕ Δ
62:     T' ← f_{T,1}(M_* ⊕ Δ)
63:     T' ← left_{|T|}(T'); P ← right_{n−|T|}(M_*)
64:     if T' ≠ T return ⊥
65:     if P ≠ left_{n−|T|}(10^{n−1}) return ⊥
66:     return M_1‖...‖M_m‖left_{|T|}(M_*)
```

Figure 14: The games G_2 and G_3 for bounding $\mathbf{Adv}^{\mathbf{auth}}_{\mathrm{SAEF}[\pi_0,f_1]}$ (continued from Figure 13). The game G_2 does *not* contain the boxed statements, while G_3 does. The predicates P_{A} and P_{M} are defined in Section C.

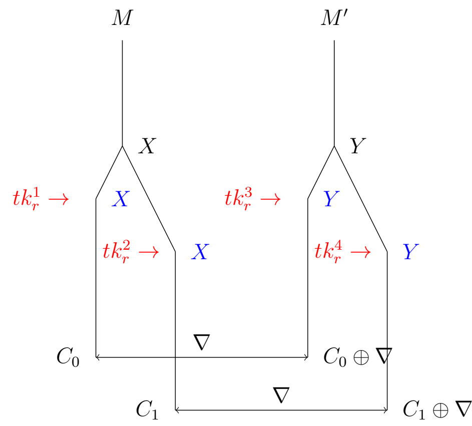


Figure 15: RTK boomerang attack against forkcipher producing forgery for single block. Here tk_r^1 and tk_r^2 are two round keys after forking which introduce the tweakey difference. X and Y are states of the ForkSkinny after forking.