Cryptanalysis of Forkciphers

Augustin Bariant, Nicolas David, Gaëtan Leurent

IACR Transactions on Symmetric Cryptology
ISSN 2519-173X, Vol. 2020, No. 1, pp. 233–265. DOI:10.13154/tosc.v2020.i1.233-265

Cryptanalysis of Forkciphers

Augustin Bariant[1], Nicolas David[1,2] and Gaëtan Leurent[1]

[1] Inria, Paris, France
augustin.bariant@polytechnique.edu, gaetan.leurent@inria.fr
[2] ENS Paris-Saclay, Gif-sur-Yvette, France
nicolas.david@ens-paris-saclay.fr

Abstract. The forkcipher framework was designed in 2018 by Andreeva et al. for authenticated encryption of short messages. Two dedicated ciphers were proposed in this framework: ForkAES based on the AES (and its tweakable variant Kiasu-BC), and ForkSkinny based on Skinny. The main motivation is that the forked ciphers should keep the same security as the underlying ciphers, but offer better performances thanks to the larger output. Recent cryptanalysis results at ACNS '19 have shown that ForkAES actually offers a reduced security margin compared to the AES with an 8-round attack, and this was taken into account in the design of ForkSkinny.

In this paper, we present new cryptanalysis results on forkciphers. First we improve the previous attack on ForkAES in order to attack the full 10 rounds. This is the first attack challenging the security of full ForkAES. Then we present the first analysis of ForkSkinny, showing that the best attacks on Skinny can be extended to one round for most ForkSkinny variants, and up to three rounds for ForkSkinny-128-256. This allows to evaluate the security degradation between ForkSkinny and the underlying block cipher.

Our analysis shows that all components of a forkcipher must be carefully designed: the attack against ForkAES uses the weak diffusion of the middle rounds in reconstruction queries (going from one ciphertext to the other), but the attack against ForkSkinny uses a weakness of the tweakey schedule in encryption queries (when one branch of the tweakey schedule is skipped).

Keywords: Forkciphers, TWEAKEY, ForkAES, ForkSkinny, Cryptanalysis, NIST Lightweight Standardisation

1 Introduction

Block ciphers are the main building block of symmetric cryptography, with the AES standard [DR13] being widely used and strongly believed to be secure. However, standard block ciphers and modes of operation (*e.g.* AES-GCM) might be too heavy for some constrained environments. Therefore there is an ongoing effort by academia to design *lightweight* cryptography algorithms with a smaller footprint, and NIST is currently running a standardization effort to standardize some of them.[1] One of the ideas recently proposed in this field is the forkcipher framework, designed in 2018 by Andreeva *et al.* [ARVV18, ALP+19a, ALP+19b] for authenticated encryption of very short messages, with a smaller overhead than the typical combination of an encryption mode and a MAC. Let us introduce formally the forkcipher notion:

A block cipher is a family of keyed permutation of the space of n-bit messages, indexed

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

Received: 2019-11-23, Accepted: 2020-01-23, Published: 2020-05-07

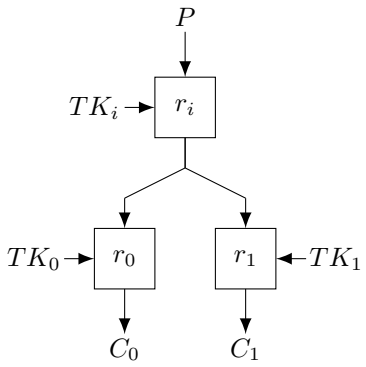


Figure 1: Illustration of an encryption by a forkcipher. Each box corresponds to several rounds of a block cipher round function.

by a k-bit key:

$$E : \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^n$$
$$K, P \mapsto C.$$

When the key is chosen randomly, a block cipher should behave like a pseudo-random permutation. Concrete block ciphers consist generally of a round function iterated a specific number of times r_{tot} with a subkey addition between each round.

A tweakable block cipher has an extra t-bit input called a tweak, generally assumed to be chosen by the adversary:

$$\widetilde{E} : \{0,1\}^k \times \{0,1\}^t \times \{0,1\}^n \to \{0,1\}^n$$
$$K, T, P \mapsto C.$$

When the key is chosen randomly, a tweakable block cipher should behave like a family of independent pseudo-random permutations indexed by the tweak, even if the tweak is public or chosen by the adversary. Several recent tweakable block ciphers follow the tweakey construction [JNP14], where the tweak and key are processed together to generate the subkeys for each round.

A (tweakable) fork cipher is a generalization that outputs two n-bit ciphertexts rather than one:

$$\widetilde{F} : \{0,1\}^k \times \{0,1\}^t \times \{0,1\}^n \to \{0,1\}^n \times \{0,1\}^n$$
$$K, T, P \mapsto C_0, C_1.$$

For each key and tweak, the functions from the input (P) to each half of the output ($P \mapsto C_0$ and $P \mapsto C_1$) should be a permutation, and the corresponding family should be a secure tweakable block cipher (See Figure 1). In addition to encryption queries ($P \mapsto C_0 \| C_1$) and decryption queries ($C_i \mapsto P$), the security model of forkciphers also considers so-called *reconstruction queries*, which take C_0 as input and C_1 as output (or vice-versa).

A forkcipher is built from an iterated block cipher by applying the round function r_i times to the plaintext (with subkeys $1, \ldots, r_i$), then forking the state, and computing independently both ciphertexts: applying r_0 rounds in the first branch (with subkeys $r_i{+}1, \ldots, r_i{+}r_0$) and r_1 rounds in the second branch (with subkeys $r_i{+}r_0{+}1, \ldots, r_i{+}r_0{+}r_1$).

The subkeys are generated by extending the tweakey schedule to produce $r_i + r_0 + r_1$ subkeys rather than r_{tot} in the original block cipher (depending on the design, some extra whitening keys might be used).

In most cases, $r_0 = r_1 \geq r_{\text{tot}}/2$ and $r_i + r_0 = r_{\text{tot}}$. This ensures that all queries go through at least r_{tot} rounds (including reconstruction queries), so that the forkcipher is expected to be as secure as the underlying (tweakable) block cipher. In particular encryption queries $P \mapsto C_0$ correspond exactly to the original block cipher.

The forkcipher can be used directly for authenticated encryption of short messages of less than one block, without needing a mode of operation (using the nonce as tweak). The cost to process the message will be roughly 1.5 block cipher calls, while block cipher-based modes require at least 2 block cipher calls.

Andreeva *et. al* have proposed two dedicated forkciphers using this framework: ForkAES [ARVV18] based on the AES round function [DR13], and ForkSkinny [ALP+19a, ALP+19b] based on the Skinny round function [BJK+16]. ForkSkinny has been submitted to the NIST Lightweight Cryptography standardization process and has been selected to the second round.

1.1 Our Results

In this paper we present new cryptanalysis results on both forkciphers primitives designed by Andreeva *et al.*

ForkAES. Previous results by Banik *et al.* [BBJ+19] have shown that ForkAES is weaker than AES, because reconstruction queries are easier to attack than encryption or decryption queries. Indeed, a reconstruction query goes through decryption rounds followed by encryption rounds, and this strongly limits the diffusion around the middle. This leads to a truncated differential attack against 8 rounds.

In Section 2, we extend this attack from 8 rounds to the full 10 rounds by introducing two new ideas. First, we consider weak keys such that the diffusion in the middle rounds is even weaker than usual. Second, instead of considering a single pair of texts with a particular difference, we consider two related pairs simultaneously, so that if one pair follows the characteristic, then the second pair will also follow it with high probability.

We give several attacks with a varying fraction of weak keys, as shown in Table 1; for each attack the expected complexity is lower than $2^{127} \times \epsilon$, where ϵ corresponds to the fraction of weak keys. In particular, the last attack can target the full keyspace with an expected complexity of 2^{125}.

Even though the attacks are not practical, these are the first that break the security claim of the full ForkAES. In particular, it shows that ForkAES offers a significantly lower security that the underlying block cipher AES, where the best attacks reach only 7 rounds.

ForkSkinny. In Section 3, we show attacks against several variants of ForkSkinny. ForkSkinny has more rounds than Skinny in reconstruction queries to limit the impact of the weaker diffusion, but the parameters used by the designers have a bad interaction with the tweakey schedule. We rely on two well-known properties of the Skinny tweakey schedule. First, each round uses only key material from one half of the key (depending on the parity of the round number). Second, some tweakey differences lead to inactive round keys every 30 rounds because differences cancel out (the tweakey schedule is linear).

In particular, when r_0 is odd, encryption queries for C_1 have two consecutive rounds that use key material from the same half of the master key. Moreover, when $r_0 = 27$, encryption queries for C_1 have 27 rounds of "blank" key schedule; this allows to have two consecutive cancellation events (cancellation at round i is followed by a round using the inactive half of the key, then 27 blank rounds, another round with the inactive half,

Table 1: Cryptanalysis results against AES, Kiasu-BC, and ForkAES (excluding optimized brute-force). The Time column corresponds to the expected time, and ϵ to the fraction of weak keys targeted by the attack.

Algorithm	Attack Type	Rds.	Data	Time	Mem.	ϵ	Reference
AES-128	Square	6	2^{32}	2^{71}	2^{32}	1	[DKR97]
AES-128	Impossible Diff.	7	$2^{106.2}$	$2^{110.2}$	$2^{90.2}$	1	[MDRMH10]
AES-128	Meet in the Middle	7	2^{97}	2^{99}	2^{98}	1	[DFJ13]
Kiasu-BC	Square	7	$2^{48.5}$	$2^{43.6}$	$2^{41.7}$	1	[DEM16]
Kiasu-BC	Meet in the Middle	8	2^{116}	2^{116}	2^{86}	1	[MAY16]
Kiasu-BC	Impossible Diff.	8	2^{118}	$2^{120.2}$	2^{102}	1	[DL17]
Kiasu-BC	Boomerang	8	2^{103}	2^{103}	2^{60}	1	[DL17]
ForkAES-$*$-4-4	Impossible Diff.	8	$2^{39.5}$	2^{47}	2^{35}	1	[BBJ+19]
ForkAES-$*$-4-4	Reflection Diff.	8	2^{35}	2^{35}	2^{33}	1	[BBJ+19]
ForkAES-$*$-5-5	Truncated Diff.	10	$2^{74.5}$	2^{75}	$2^{59.5}$	2^{-32}	Sect. 2.4
ForkAES-$*$-5-5	Truncated Diff.	10	$2^{100.5}$	2^{114}	$2^{80.5}$	2^{-4}	Sect. 2.5.2
ForkAES-$*$-5-5	Truncated Diff.	10	2^{119}	2^{125}	2^{83}	1	Sect. 2.5.3

Table 2: Cryptanalysis results against Skinny and ForkSkinny.

Algorithm	Attack Type	Rds.	Data	Time	Mem.	Reference
Skinny-128-256	RK Impossible Diff.	23	$2^{124.5}$	$2^{251.5}$	2^{248}	[LGS17]
(256-bit key)	RK Impossible Diff.	23	$2^{124.5}$	$2^{243.5}$	$2^{155.5}$	[SMB18]
ForkSkinny-128-256 (128-bit key + 128-bit tweak)	RK Impossible Diff.	24	$2^{122.5}$	$2^{124.5}$	$2^{97.5}$	Section 3.5
ForkSkinny-128-256	RK Impossible Diff.	26	2^{125}	$2^{254.6}$	2^{160}	Section 3.6
(256-bit key)	RK Impossible Diff.	26	2^{127}	$2^{250.3}$	2^{160}	Section 3.6

and a second cancellation event). Since ForkSkinny-128-256 uses $r_0 = 27$, this allows to extend the best attacks by three rounds: we have a related-key attack against 24-round ForkSkinny-128-256 with a 128-bit key (corresponding to the parameters used in the NIST submission) and a related-key attack against 26-round ForkSkinny-128-256 with a 256-bit key. These results do not affect the security of the full ForkSkinny because it was designed with a large security margin (the full version of ForkSkinny-128-256 has 48 rounds), but they show that bad parameter choices make it significantly weaker than Skinny.

1.2 Preliminaries

We will use some differential properties of S-Boxes in our attacks. In the following S denotes the S-Box of the cipher being analyzed, with b the size of the S-Box.

First we denote by $\mathcal{P}(\delta_i, \delta_o)$ the probability of having an output difference of δ_o if the input difference is δ_i through S:

$$\mathcal{P}(\delta_i, \delta_o) = |\{x \in \mathbb{F}_{2^b} : S(x) \oplus S(x \oplus \delta_i) = \delta_o\}| \times 2^{-b}.$$

For a fixed δ_i, there are at most 2^{b-1} values δ_o with $\mathcal{P}(\delta_i, \delta_o) \neq 0$.

Note that an important cryptographic property of the AES S-Box is that for $\delta_i \neq 0$, $\mathcal{P}(\delta_i, \delta_o)$ is either $2^{-7}, 2^{-6}$ or 0, and for every non-zero δ_i, there exists a unique $\delta_o \in \mathbb{F}_{2^8}$ such that $\mathcal{P}(\delta_i, \delta_o) = 2^{-6}$. The inverse AES S-Box also has this property.

We also use the following lemma:

Lemma 1. *For two random difference $\delta_{in} \neq 0, \delta_{out} \neq 0$, the equation $S(x \oplus \delta_{in}) \oplus S(x) = \delta_{out}$ has one solution x in average.*

2 Cryptanalysis of ForkAES

ForkAES [ARVV18] is based on the AES standard [DR13]; more precisely, it is a forked variant of the tweakable block cipher Kiasu-BC [JNP14], which reuses the AES round function.

2.1 Description of ForkAES

The AES round function. The AES cipher is the most widely used block cipher today, designed by Daemen and Rijmen in 1998, and selected in 2000 after a NIST competition [DR13]. We focus on AES-128, taking a 128-bit plaintext and a 128-bit key as input and returning a 128-bit ciphertext. The state is represented as a 4×4-byte array, and processed through 10 rounds with the following operations:

- SubBytes applies an S-Box on each byte of the state;
- ShiftRows shifts the second row of the state by 1 cell, the third row by 2 cells, and the last row by 3 cells;
- MixColumns multiplies each column of the state by an MDS matrix:

$$\mathcal{M} = \begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix}; \tag{1}$$

- AddRoundKey xors the state with the round key.

The MixColumns operation is defined over the field $\mathbb{F}_{2^8}$, constructed as $\mathbb{F}_2[X]/(X^8 + X^4 + X^3 + X + 1)$. For simplicity, we refer to elements of the field using the integer with the same bit representation, with an hexadecimal notation in fixed-with font. In particular, `2` represents the polynomial X, and its inverse is represented as `8d`.

The final round omits the MixColumns operation, and an initial round key is xored to the state before the first round. The best attack known against the AES can only break 7 of the 10 rounds (See Table 1).

The tweakable block cipher Kiasu-BC. Kiasu-BC is a tweakable variant of AES designed by Jean, Nikolic, and Peyrin [JNP14]. It has the same number of rounds as the AES and the round functions are very similar, but it has an additional 64-bit tweak input. The only change between the two primitives is the AddRoundTweak (AT) operation after the AddRoundKey operation of each round, in which the 64-bit tweak is xored to the first two rows of the state. Note that unlike the key of AES, the tweak does not go through a tweak schedule, and is the same on each round.

Since the attacker chooses the tweak, he can use it to inject differences in the state, and cancel a state difference with a tweak difference (similar to the related-key model). Despite the designers' initial claim, for most of the existing attacks on AES there exists an equivalent attack on Kiasu-BC reaching one more round by taking advantage of the tweak. Table 1 sums up the attacks and their complexities.

ForkAES. ForkAES is a forkcipher based on Kiasu-BC. It takes a 128-bit plaintext, a 128-bit key and a 64-bit tweak input and returns two 128-bit ciphertexts C_0 and C_1 derived from the same plaintext. The encryption of the data works almost exactly like Kiasu-BC. To compute C_0 and C_1, five Kiasu-BC rounds are applied to the plaintext. Then we duplicate the state and compute 5 more Kiasu-BC rounds with different round keys, obtained with an extension of the Kiasu-BC tweakey schedule. This way, we efficiently obtain two ciphertexts that both went through 10 rounds of Kiasu-BC, for a total cost of 15 Kiasu-BC round encryptions.

We denote ForkAES-r_i-r_0-r_1 the round reduced ForkAES version composed of r_i rounds before the forking point, and respectively r_0 and r_1 rounds in each branch. Full ForkAES is therefore ForkAES-5-5-5. For attacks that do not depend on r_i, we denote the round reduced version as ForkAES-$*$-r_0-r_1.

Previous results. Although the designers of ForkAES state "Since we do not introduce any novel design complexities, the security of our forkcipher design can be reduced to the security of the AES and Kiasu ciphers for further type of attacks", this type of encryption provides the attacker with a new oracle: the reconstruction oracle. In this model, the attacker can ask for the second ciphertext C_1 obtained from a chosen ciphertext C_0. Unlike Kiasu-BC, the path from C_0 to C_1 consists of 5 decryption rounds followed by 5 encryption rounds, which reduces the diffusion around the middle round.

Banik *et al.* exploited the lack of diffusion to revisit attacks against AES and Kiasu-BC and apply them to a larger number of rounds on ForkAES [BBJ+19]. In particular, they describe a truncated differential attack on ForkAES-$*$-4-4 with complexity only 2^{35}, much lower than the best attacks on Kiasu-BC with 8 rounds.

2.2 Notation

We denote x_i, y_i, z_i and w_i the states after respectively the AddRoundTweakey, SubBytes, ShiftRows and MixColumns operations of round i. For a 128-bit state s, we denote $s[j]$ the j-th byte of the state, following the AES byte ordering:[2]

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

ForkAES forks the state after 5 rounds and transforms it through two different branches into two ciphertexts. The first branch takes round keys k_5 to k_{10} and the second branch takes round keys k_{11} to k_{16}, derived from k with an extended key schedule. We denote the ciphertext obtained from the first branch as C_0 and the ciphertext obtained from the second branch as C_1.

Equivalent last rounds. For the following attacks, we denote $\widehat{C}_0$ the state after partial inversion of the tenth round (we invert AddRoundTweak, MixColumns and ShiftRows), and $\widehat{k}_{10}$ the equivalent round key of the tenth round (we point out that ForkAES does not omit the MixColumns operation in the last round). The equivalent operation is denoted by AK_{10}^{eq}. In other words,

$$\widehat{C}_0 = \mathsf{SR}^{-1}(\mathsf{MC}^{-1}(\mathsf{AT}(C_0))) \qquad \widehat{k}_{10} = \mathsf{SR}^{-1}(\mathsf{MC}^{-1}(k_{10})).$$

Similarly, we denote $\widehat{C}_1$ the equivalent output ciphertext and $\widehat{k}_{16}$ the equivalent ultimate round key. The equivalent operation is denoted by AK_{16}^{eq}. They verify:

$$\widehat{C}_1 = \mathsf{SR}^{-1}(\mathsf{MC}^{-1}(\mathsf{AT}(C_1))) \qquad \widehat{k}_{16} = \mathsf{SR}^{-1}(\mathsf{MC}^{-1}(k_{16})).$$

[2]Note that the standard byte ordering for AES and Skinny are not the same.

We denote by $(\widehat{C}_0, T)$ the ciphertext $\widehat{C}_0$ with the tweak T. We denote ATK_i the AddRoundTweakey operation, which combines the AddRoundKey and the AddRoundTweak operations of round i. We denote the associated tweakey as $tk_i = T \oplus k_i$ where T is the tweak and k_i is the round key of round i.

2.3 Previous Attack Against ForkAES-∗-4-4

We start by explaining the details of the previous reflection differential attack of Banik et al. [BBJ+19]. This attack is a truncated differential attack exploiting reconstruction query.

2.3.1 Truncated Differential Characteristic and Probability

The attack uses the truncated differential characteristic of Figure 2. For each byte of the input, output, and internal state, the characteristic specifies whether there should be a difference or not. We evaluate the probability of the characteristic as the product of the probability of each rounds. The attack has two phases: first we build a structure of C_0 values so that the corresponding pairs have a difference compatible with the characteristic, and we call the reconstruction oracle to get the corresponding C_1. Then, we look for pairs with the prescribed C_1 difference. If we identify a pair, we assume that the internal states follow the characteristic, and we use this assumption to recover part of the key.

Because we consider ForkAES-∗-4-4, round keys of the first branch are k_5,k_6,k_7,k_8 and k_9, and round keys of the second branch are k_{10},k_{11},k_{12},k_{13} and k_{14}. A reconstruction query from C_0 to C_1 starts with the whitening key k_9, goes through 4 inverse rounds with round keys k_8, k_7, k_6, then key $k_5 \oplus k_{10}$ is xored at the forking point followed by 4 forward rounds with keys k_{11}, k_{12}, k_{13} and k_{14} (final whitening key). We use $\widehat{k}_9$ and $\widehat{k}_{14}$ as the equivalent round keys, as defined in Section 2.2:

$$\widehat{k}_9 = \mathsf{SR}^{-1}(\mathsf{MC}^{-1}(k_9)) \qquad \widehat{k}_{14} = \mathsf{SR}^{-1}(\mathsf{MC}^{-1}(k_{14})).$$

The input difference of the characteristic is only active on bytes 0,1,2,3, and the tweak difference is active only on the first byte. We can compute the probability of the characteristic as follows:

- $\Pr(w_8 \to z_8) = 2^{-24}$ since three bytes become inactive after inverse MixColumns.
- $\Pr(x_8 \to w_7) = 2^{-8}$ since the difference in byte 0 cancels out the tweak difference.
- $\Pr(z_{10} \to w_{10}) = 2^{-24}$ since three bytes become inactive after MixColumns.
- $\Pr(w_{10} \to x_{11}) = 2^{-8}$ since the difference in byte 0 cancels out the tweak difference.

In total, the probability of the characteristic is 2^{-64}.

2.3.2 Attack Procedure

While the description given in [BBJ+19] uses a fixed tweak difference, we will not fix it in advance, but we use structures on the tweak to further reduce the complexity of the attack. Moreover, we will show in Section 2.5.1 that fixing the tweak does not ensure the above probability of going through the characteristic, as the middle rounds can only be satisfied if the tweak difference and the key are compatible.

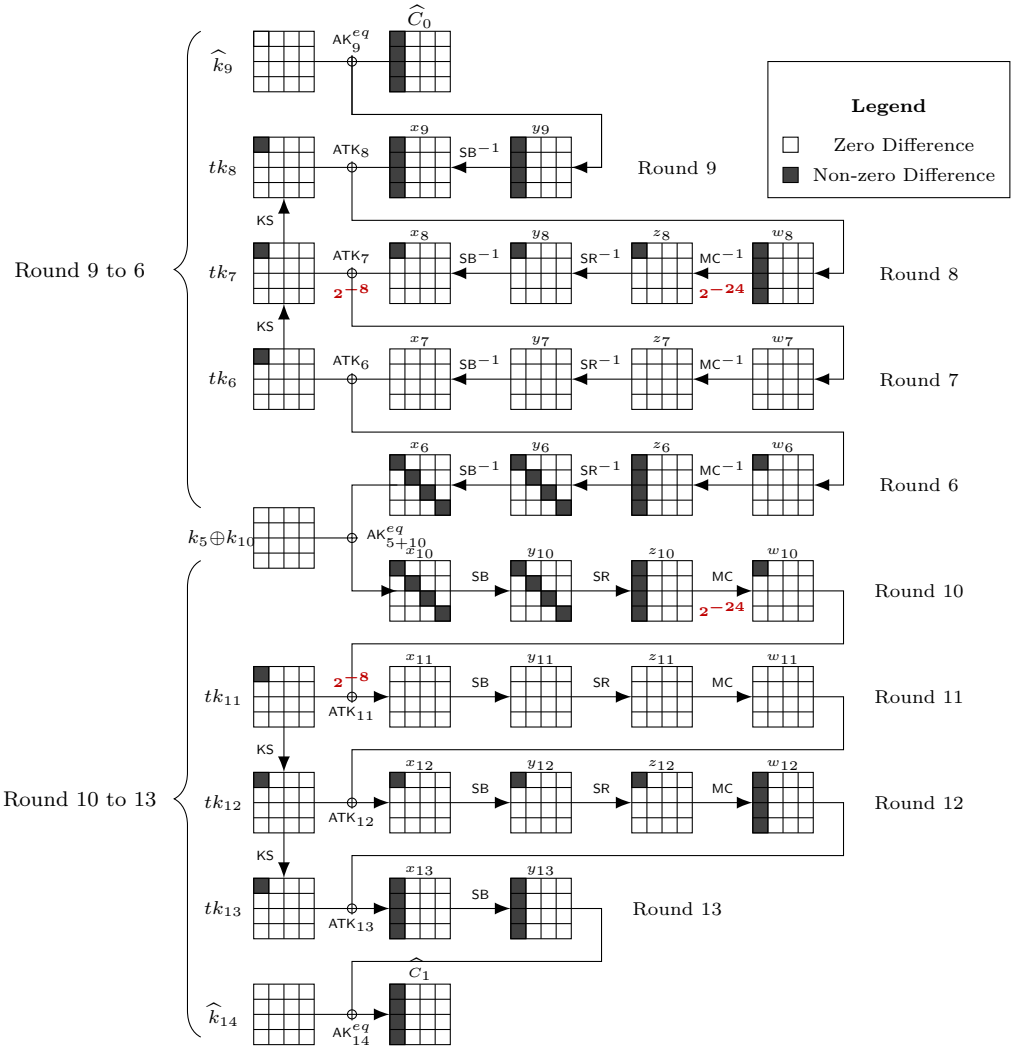


Figure 2: Truncated differential characteristic for ForkAES-∗-4-4.

Construction of the pairs. First, we choose arbitrary values for the last three columns of C_0, along with arbitrary values for all bytes of the tweak except the first one. Iterating over the first byte of the tweak and the values of the first column of C_0 gives us a set of 2^{40} (T, C_0) values. Among them, we query $2^{32.5}$ randomly chosen elements. This gives us $2^{32.5} \times (2^{32.5} - 1)/2 \approx 2^{64}$ pairs of inputs verifying the input difference of the characteristic. In average, one pair satisfies the characteristic.

Filtering the pairs. We can distinguish the right pair from the wrong ones by storing the corresponding C_1 ciphertexts into a hash table, indexed by the values of the three last columns. If a pair of inputs satisfies the characteristic, we detect it as a collision in the hash table. For random pairs of outputs, there is a collision with probability 2^{-96}. As we have 2^{64} pairs stored in the hash table, a random collision occurs with probability 2^{-32}, which can be neglected. The only collision is therefore the right pair.

Reducing the key space. The pair went through the characteristic so the difference in state x_{12} is exactly the tweak difference. As mentioned in Section 1.2, there are only 2^7 possible output differences through the S-Box if the input difference is fixed. We can compute these differences from the known tweak difference, and propagate the differences through SR, MC and ATK_{13} since they are linear operations, in order to get 2^7 candidates for the x_{13} difference. We also know the difference in $\widehat{C}_1$, therefore the difference in y_{13}. Using Lemma 1, we recover the value of the first column of y_{13} for each of the 2^7 potential differences. Then we can deduce 2^7 candidates for the first column of the key $\widehat{k}_{14}$, from the value of y_{13} and $\widehat{C}_1$.

Full key recovery. No operation depends specifically on the column position in AES, Kiasu-BC and ForkAES. Consequently, we can iterate this attack by shifting the entire characteristic to another column. We can therefore recover 2^7 candidates for each column of $\widehat{k}_{14}$. We can invert the key schedule to recover the master key from $\widehat{k}_{14}$. The 2^{28} candidates are ultimately tested exhaustively.

2.3.3 Complexity Evaluation

We require $2^{32.5}$ queries per column, which induces a data complexity of $2^{34.5}$. The memory complexity is equivalent to $2^{32.5}$ AES states to store $2^{32.5}$ values of $\widehat{C}_0$ in the hash table. The time complexity is 2^{28} encryptions for the final exhaustive search and $2^{34.5}$ memory accesses for the data processing. Eventually the (Data,Time,Memory) complexity will be:

$$(D, T, M) = (2^{34.5}, 2^{34.5}, 2^{32.5}).$$

2.4 Attack Against Full ForkAES for 2^{96} Weak Keys

We now describe new attacks against ForkAES-$*$-5-5, by improving the previous techniques. A reconstruction query from C_0 to C_1 now starts with the whitening key k_{10}, goes through 5 inverse rounds with round keys k_9, k_8, k_7, k_6, then key $k_5 \oplus k_{11}$ is xored at the forking point followed by 5 forward rounds with keys k_{12}, k_{13}, k_{14}, k_{15} and k_{16} (final whitening key).

Our first attack targets weak keys such that $k_5 \oplus k_{11}$ only has zero values on the diagonal; this happens with probability 2^{-32}. Our attack uses a differential characteristic with high probability, shown in Figure 3. An important particularity of this characteristic is that if one pair satisfies this characteristic, then it is easy to construct another pair that has a very high chance of satisfying the characteristic as well.

2.4.1 Notation

To describe our attacks, we use θ to denote a non-zero byte difference such that $\mathcal{P}(\theta, \theta/2) = 2^{-7}$, and λ to denote the unique value such that $\mathcal{P}(\theta, \lambda) = 2^{-6}$. There are 123 values of θ satisfying this condition. We use Θ to denote the state difference active only on the first byte with $\Theta[0] = \theta$. For instance, we can use $\theta = \mathtt{10}$, $\theta/2 = \mathtt{08}$, $\lambda = \mathtt{a9}$.

2.4.2 Differential Characteristic for the First Pair

First, we guess the first byte of the equivalent key $\widehat{k}_{10}$ and denote it K, and we consider input pairs $p = ((C_0, T), (C_0', T'))$ of the form:

$$\widehat{C}_0 = \left(\begin{array}{c|c} S(0) \oplus K & \\ 0 & \\ 0 & u \\ 0 & \end{array}\right), \qquad T = 0, \qquad \widehat{C}_0' = \left(\begin{array}{c|c} S(\theta) \oplus K & \\ 0 & \\ 0 & v \\ 0 & \end{array}\right), \qquad T' = \Theta$$

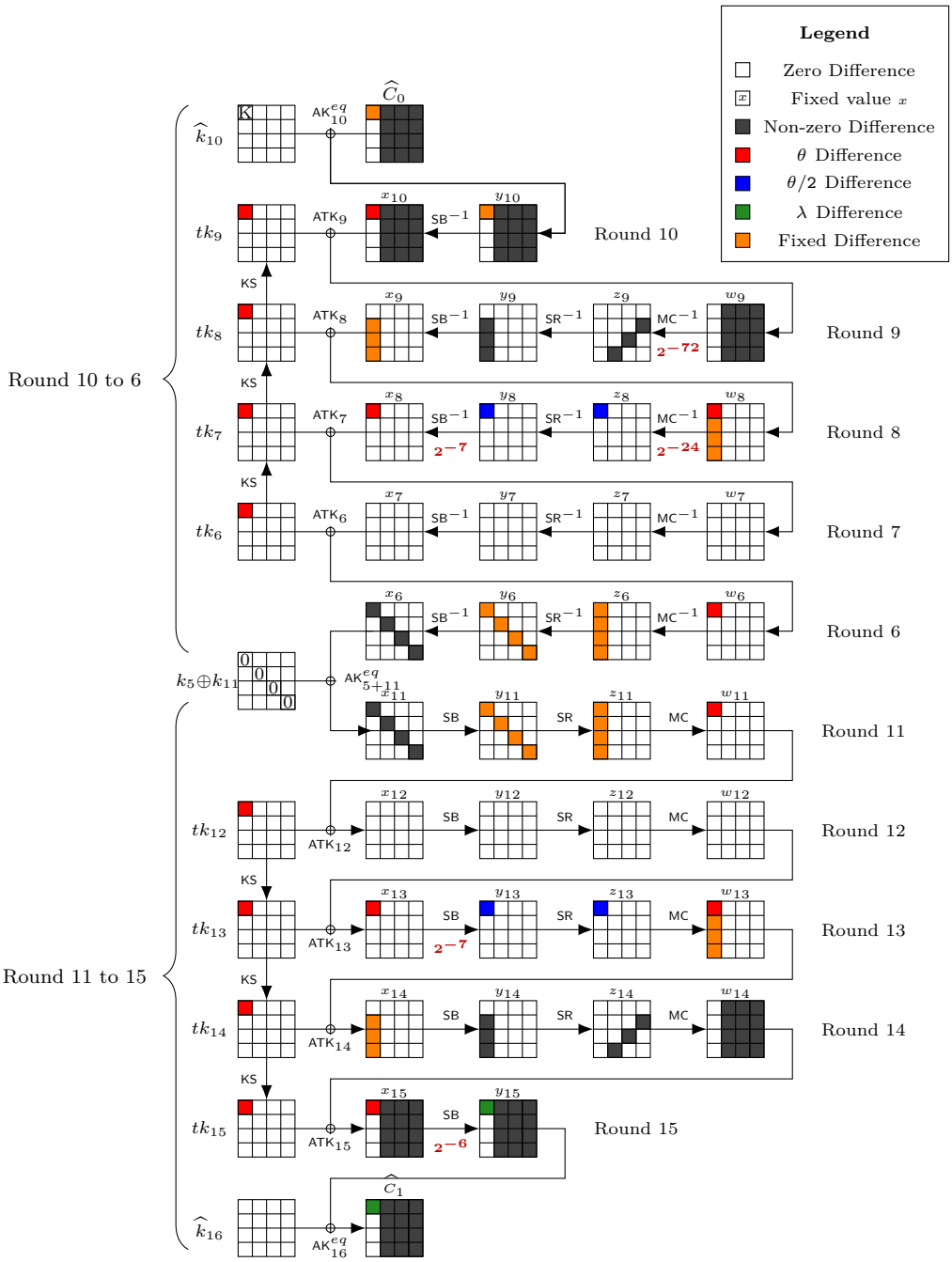


Figure 3: Differential characteristic for very weak keys.

for any 96-bit vectors u and v (seen as 4×3 matrices). By construction, the first byte difference at state x_{10} is exactly the tweak difference, so that the state w_9 has an inactive first column. Therefore, the probability of the characteristic can be computed as

- $\Pr(w_9 \to z_9) = 2^{-72}$ since 9 bytes become inactive after the MixColumns operation.
- $\Pr(w_8 \to z_8) = 2^{-24}$ since 3 bytes become inactive after the MixColumns operation. Moreover, since the difference in $w_8[0]$ is θ, cancelling three byte implies that the difference in the first column of w_8 must be $[\theta, \theta/2, \theta/2, \theta \oplus \theta/2]$. Alternatively, we can consider that $\Pr(y_9 \to x_9) = 2^{-24}$ if the difference in the first column of x_9 is fixed in advance to $[0, \theta/2, \theta/2, \theta \oplus \theta/2]$.
- $\Pr(y_8 \to x_8) = \mathcal{P}(\theta, \theta/2) = 2^{-7}$ because of the choice of θ.
- Round 7 is then inactive.
- Values in the diagonals of both elements of the pair in state x_6 do not change with the AK^{eq}_{5+11} operation, because of the hypothesis we made on the key. Consequently, both elements of the pair have the same diagonal values in state y_6 and in state y_{11}. The difference in state w_{11} is therefore the tweak difference with probability 1.
- Round 12 is then inactive.
- $\Pr(x_{13} \to y_{13}) = 2^{-7}$ because of the choice of θ.
- $\Pr(x_{15} \to y_{15}) = 2^{-6}$ because we filter the pair on the first byte difference and we want a λ difference.

In summary, for the first pair, $\Pr(\widehat{C}_0 \to \widehat{C}_1) = 2^{-72-24-7-7-6} = 2^{-116}$. Since a random pair has the prescribed output difference with probability 2^{-32}, this is too low for a direct attack, but we use an amplification technique using twin pairs.

2.4.3 Construction of Twin Pairs

We build a twin pair $\bar{p}$ from the pair p so that $\bar{p}$ follows the first three rounds of the characteristic with probability 1 assuming that p follows it. The probability of the total characteristic for $\bar{p}$ will therefore be 2^{-13}:

$$\widehat{\overline{C}}_0 = \left(\begin{array}{c|c} S(\tau[0]) \oplus K & \\ 0 & \\ 0 & u \\ 0 & \end{array}\right), \quad \overline{T} = \tau, \quad \widehat{\overline{C}}'_0 = \left(\begin{array}{c|c} S(\tau[0] \oplus \theta) \oplus K & \\ 0 & \\ 0 & v \\ 0 & \end{array}\right), \quad \overline{T}' = \tau \oplus \Theta$$

where τ is the tweak active only on the first byte, such that $\tau[0] = \theta/\mathtt{1c}$. By construction, the twin pair satisfies:

$$\overline{w}_9[0] = \mathsf{ATK}_9(\mathsf{SB}^{-1}(\mathsf{AK}^{eq}_{10}(\mathsf{SB}(\tau[0] \quad) \oplus K))) = 0 = w_9[0]$$
$$\overline{w}'_9[0] = \mathsf{ATK}_9(\mathsf{SB}^{-1}(\mathsf{AK}^{eq}_{10}(\mathsf{SB}(\tau[0] \oplus \theta) \oplus K))) = 0 = w'_9[0].$$

Up to this point, neither MixColumns nor ShiftRows have been applied on the states, therefore each byte of the state has a specific value depending only on the corresponding byte of the input. This implies $\overline{w}_9 = w_9$ and $\overline{w}'_9 = w'_9$. This equality of states is satisfied for the whole round 9, up to x_9. After the next tweak addition, we have the following property:

$$\overline{w}_8 = w_8 \oplus \tau \qquad\qquad \overline{w}'_8 = w'_8 \oplus \tau.$$

In particular, the differences of both pairs in state w_8 are equal. Because $w_8 \to y_8$ consists only in linear operations, the differences in state y_8 of both pairs are equal, in particular the difference $\overline{y}_8 \oplus \overline{y}'_8$ is only active on the first byte, with difference $\theta/2$.

More precisely, we have:

$$\overline{y}_8 = y_8 \oplus \mathsf{SR}^{-1}(\mathsf{MC}^{-1}(\tau)) \qquad \overline{y}'_8 = y'_8 \oplus \mathsf{SR}^{-1}(\mathsf{MC}^{-1}(\tau)).$$

Note that τ was chosen so that the first byte of $\mathsf{SR}^{-1}(\mathsf{MC}^{-1}(\tau))$ is $\theta/2$. Therefore the pair $(\overline{y}_8[0], \overline{y}'_8[0])$ is the same as the pair $(y_8[0], y'_8[0])$ and follows the S-Box transition with probability 1:

$$\overline{y}_8[0] = y_8[0] \oplus \theta/2 = y'_8[0] \qquad \overline{y}'_8[0] = y'_8[0] \oplus \theta/2 = y_8[0].$$

2.4.4 Filtering Twin Pairs

The main step of the attack will be to build quadruples of oracle queries corresponding to a pair and its twin, and to filter the candidates that follow the truncated characteristic. The differential characteristic for both pairs has a total probability of 2^{-129}, and we have a filter on the output satisfied with probability 2^{-64} (32 bits of fixed difference in each pair).

We strengthen the filter using the fact that both pairs follow the characteristic with the same key: for each pair, we can deduce four candidates for $\widehat{k}_{16}[0]$ so that the transition $\theta \to \lambda$ is followed in the last round, and the two sets should have a common value. More precisely, we know that there are exactly four possible values for $y_{15}[0]$, corresponding to the four values following the S-Box transition $\theta \to \lambda$; we denote them as a_0, a_1, a_2 and a_3. If both pairs went through the characteristic, the four values $\widehat{C}_1[0]$, $\widehat{C}'_1[0]$, $\widehat{\overline{C}}_1[0]$ and $\widehat{\overline{C}}'_1[0]$ must be in the same coset $\mathsf{AK}^{eq}_{16}[0] \oplus \{a_0, a_1, a_2, a_3\}$. We can build a linear function φ that identifies the coset uniquely,[3] and we have an additional 6-bit filter $\varphi(\widehat{C}_1[0]) = \varphi(\widehat{\overline{C}}_1[0])$. Note that we also have $\varphi(\widehat{C}_1[0]) = \varphi(\widehat{C}'_1[0])$ and $\varphi(\widehat{\overline{C}}_1[0]) = \varphi(\widehat{\overline{C}}'_1[0])$ by definition if the pairs satisfy the output difference.

Random twin pairs pass the complete filter with probability 2^{-70}.

However, a good quadruple satisfies much more restrictive conditions: the difference at the end of round 14 is in a space of 2^{24} output differences. The rest of this attack is devoted to efficiently finding these quadruples. Alternatively, this can be described as an attack with a characteristic going only until the end of round 14, and using partial decryption in the last round to detect good pairs and recover the key.

2.4.5 Using Structures

We now explain how to use a structure of C_0 values and identify candidates pairs efficiently. Following the previous notations, for 96-bit vectors u and v, we make four reconstruction queries:

$$\widehat{C}_0(u), 0 \mapsto \widehat{C}_1(u) \qquad \widehat{\overline{C}}_0(u), \tau \mapsto \widehat{\overline{C}}_1(u)$$
$$\widehat{C}'_0(v), \Theta \mapsto \widehat{C}'_1(v) \qquad \widehat{\overline{C}}'_0(v), \Theta \oplus \tau \mapsto \widehat{\overline{C}}'_1(v)$$

and we want to identify u and v such that the output satisfy a 70-bit condition:

$$\widehat{C}_1(u)[0,1,2,3] = \widehat{C}'_1(v)[0,1,2,3] \oplus [\lambda, 0, 0, 0] \tag{2}$$
$$\widehat{\overline{C}}_1(u)[0,1,2,3] = \widehat{\overline{C}}'_1(v)[0,1,2,3] \oplus [\lambda, 0, 0, 0] \tag{3}$$
$$\varphi(\widehat{C}_1(u)[0]) = \varphi(\widehat{\overline{C}}_1(u)[0]) \tag{4}$$

[3] Such as the orthogonal projection with the dimension 2 kernel $\langle a_0 \oplus a_1, a_0 \oplus a_2, a_0 \oplus a_3 \rangle$.

The last condition depends only on u and we can filter out values that do not satisfy it. The three conditions imply another 6-bit condition that can be used to filter v values:

$$\varphi(\widehat{C}'_1(v)[0]) = \varphi(\widehat{\overline{C}}'_1(v)[0]). \tag{5}$$

We store the remaining values in two hash tables:

- H indexed by $\widehat{C}_1(u)[0,1,2,3] \| \widehat{\overline{C}}_1(u)[0,1,2,3]$;
- H' indexed by $\widehat{C}'_1(v)[0,1,2,3] \| \widehat{\overline{C}}'_1(v)[0,1,2,3] \oplus [\lambda,0,0,0,\lambda,0,0,0]$.

Let's assume we have a match between an element of the first table and an element of the second one, corresponding to vectors u and v. Our twin pairs will be

$$p = [\widehat{C}_0(u), \widehat{C}'_0(v)] \qquad\qquad \bar{p} = [\widehat{\overline{C}}_0(u), \widehat{\overline{C}}'_0(v)].$$

A match in the table ensures that conditions (2) and (3) are satisfied, and condition (4) was checked before filling the table.

Both hash tables have 64-bit hash keys. However, a pair (u, v) matches with probability 2^{-58} due to redundancy: conditions (2), (3) and (4) imply condition (5).

We finally have our 70-bit filter implemented with a 6-bit pre-filter and two 64-bit hash tables. The filter has a memory complexity of only $2 \times D \times 2^{-6}$ 96-bit blocks if D is the number of candidate vectors.

2.4.6 Extended Filtering

The probability that twin pairs satisfy the characteristic is 2^{-129}. Therefore, we need two sets of $2^{64.5}$ 96-bit vectors to form 2^{129} twin pairs in order to expect one right pair. After the 70-bit filter, we will still have $2^{129-70} = 2^{59}$ pairs, and we need to distinguish the right pair from the wrong ones.

The difference in z_{13} is $\theta/2$, and $z_{13} \to x_{14}$ is composed of linear operations, so we can compute the fixed difference in state x_{14}. Then, we preventively compute the 2^7 possible difference values of a byte difference in state y_{14}, say $y_{14}[1]$. We then compute the possible differences of the last column of the state x_{15}. Knowing the differences in $\widehat{C}_1$, we can deduce 2^7 possible values for the last column of $\widehat{k}_{16}$. Doing that for pairs p and $\bar{p}$, we can check whether the intersection of these possible values is empty or not. If it is, the pairs are incompatible. This costs 2^8 operations at most, and the probability of having a non-empty intersection is approximately $2^7 \times 2^7 \times 2^{-32} = 2^{-18}$. When this happens, there is in average one element in the intersection. Doing this with the last column, then the two other columns, we have another filter, which a random pair passes with probability $2^{3\times(-18)} = 2^{-54}$.

Using precomputed tables. In some attacks, this filtering will be most expensive part of the attack, with a complexity of 2^8 elementary operations per candidates. Instead of computing the sets of $\widehat{k}_{16}$ candidates on the fly, we can build a table indexed by bytes of $\widehat{C}_1$, $\widehat{C}'_1$, $\widehat{\overline{C}}_1$ and $\widehat{\overline{C}}'_1$ that contains a boolean value indicating whether the pairs are compatible or not. This implementation of the filter requires a single table access instead of 2^8 operations.

More precisely, we can deduce 2^7 candidates for $\widehat{k}_{16}[12,13]$ from $\widehat{C}_1[12,13]$ and $\widehat{C}'_1[12,13]$. Therefore, we can check if the candidates for $\widehat{k}_{16}[12,13]$ have a non-empty intersection using only the values of $\widehat{C}_1[12,13]$, $\widehat{C}'_1[12,13]$, $\widehat{\overline{C}}_1[12,13]$ and $\widehat{\overline{C}}'_1[12,13]$. After pre-computing a table indexed by these 8 bytes (of size 2^{64}), we filter candidates with complexity 1, keeping only a fraction $2^{7+7-16} = 2^{-2}$. We can actually reduce the table size to 2^{48} because the

valid values of $\widehat{C}_1[12,13]$, $\widehat{C}'_1[12,13]$, $\widehat{\widehat{C}}_1[12,13]$ and $\widehat{\widehat{C}}'_1[12,13]$ are invariant by translation. Therefore, we create a table indexed by $\widehat{\widehat{C}}_1[12,13] \oplus \widehat{C}_1[12,13]$, $\widehat{\widehat{C}}'_1[12,13] \oplus \widehat{C}'_1[12,13]$, $\widehat{C}_1[12,13] \oplus \widehat{C}'_1[12,13]$.

We generate 4 such tables checking for intersection of different key bytes, in order to keep only a fraction 2^{-8} of the candidates, and then apply the full filter to the remaining candidates. Therefore, in the description of our attacks, we consider that this filter costs only one operation per candidate.

After this extended filter, we only have $2^{59-54} = 2^5$ quadruples left. For each of them, there are in average one key guess for the three last columns and four key guesses for the first byte. We then iterate over the bytes $\widehat{k}_{16}[1,2,3]$ to proceed in an exhaustive search for the key. This has a complexity of $2^{5+2+24} = 2^{31}$ which is clearly lower than the complexities of other steps of this attack.

2.4.7 Complexity of the Attack

First, we guessed the value K of the first byte of the equivalent round key $\widehat{k}_{10}$. For each guess we query two structures of size $2^{64.5}$, corresponding to $2^{66.5}$ encryptions, which makes a total data complexity of $2^{74.5}$. The memory complexity is $2^{59.5}$, because the average number of vectors written in each hash table is $2^{58.5}$. The time complexity is at most $2^8 \times 2^{59}$ simple operations for the pair processing of Section 2.4.6 for each guess of K, hence a total of $2^8 \times 2^8 \times 2^{59} = 2^{75}$.

Therefore, the (Data,Time,Memory) complexity is: $(D, T, M) = (2^{74.5}, 2^{75}, 2^{59.5})$.

2.5 Larger Classes of Weak Keys

We now describe attacks with larger classes of weak keys, by relaxing the constraint that $k_5 \oplus k_{11}$ should have a diagonal of zeroes. However, without this constraint the path becomes impossible to satisfy for some tweak differences.

2.5.1 Incompatibility between the Tweak Difference and the Key

We focus on the middle rounds of the previous characteristic, with a tweak difference Θ active only in the first cell. As explained earlier, we want to have rounds 7 and 12 inactive, so that the difference in y_6 and y_{11} is completely determined:

$$(y_{11} \oplus y'_{11})[0,5,10,15] = \mathsf{MC}^{-1}([\theta,0,0,0]) = (y_6 \oplus y'_6)[0,5,10,15]. \tag{6}$$

Moreover, we have the following relation between y_6 and y_{11}:

$$x_6[i] = x_{11}[i] \oplus k_5[i] \oplus k_{11}[i]$$
$$S^{-1}(y_6[i]) = S^{-1}(y_{11}[i]) \oplus k_5[i] \oplus k_{11}[i]$$

Therefore, for each $i \in \{0,5,10,15\}$, the value of $y_{11}[i]$ must satisfy an equation of the form:

$$S(S^{-1}(t) \oplus \kappa) = S(S^{-1}(t \oplus \alpha) \oplus \kappa) \oplus \alpha, \tag{7}$$

with unknown t, where κ and α are parameters (corresponding respectively to $k_5[i] \oplus k_{11}[i]$ and $\mathsf{MC}^{-1}([\theta,0,0,0])[i/5]$). This equation admits solutions if and only if the coefficient (α, κ) of the Boomerang Connectivity Table of S^{-1} is non-zero [CHP+18]. Following the analysis of [BC18], there are exactly 128 such values of κ for each $\alpha \neq 0$ when S is the AES S-Box.

That implies that for each choice of θ there is a probability of 2^{-4} that the key is compatible with the tweak difference (2^{-1} for each diagonal byte). When the key is compatible we have $\Pr(x_7 \to x_{12}) \geq 2^{-28}$, because equation (7) has at least 2 solution. In other cases, the probability of passing the total characteristic is zero.

2.5.2 A Class of 2^{124} Weak Keys

We now assume that the key $k_5 \oplus k_{11}$ has at least one diagonal byte equal to zero. This happens with probability 2^{-6}. Without loss of generality, we consider that the first byte of the key is zero. We also assume that the tweak difference θ is compatible with the key, which happens with probability 2^{-3} (2^{-1} for each non-zero diagonal key byte).

This variant of the attack is very similar to the previous one. The main difference is that the characteristic (Figure 10) now has a probability 2^{-21} of passing the middle rounds (2^{-7} for each of the three rightmost columns).

As in the previous attack, we start by guessing the first byte of $\widehat{k}_{10}$, denoted by K. The probability of passing the characteristic is $2^{-116} \cdot 2^{-21} = 2^{-137}$ for the first pair and $2^{-13} \cdot 2^{-21} = 2^{-34}$ for its twin pair, which induces a total probability of 2^{-171}. We therefore use structures of size $2^{85.5}$. After the 70-bit filter we are left with 2^{101} pairs. The extended filtering costs a single operation per pair (using tables) and keeps a fraction 2^{-54} of the pairs, which gives us 2^{47} remaining twin pairs. In average, each remaining twin pair has one key guess for the three last columns, four key guesses for the first byte, and 2^{24} key guesses for bytes 1,2,3. Exhaustively testing these candidates has a complexity of $2^{47+2+24} = 2^{73}$.

This attack covers 2^{119} keys and has complexity $(D, T, M) = (2^{95.5}, 2^{109}, 2^{80.5})$ after iterating over K, with success probability $1 - 1/e \approx 0.63$.

In order to cover more keys, we can repeat the attack with different choices of θ, to cover other keys with a zero diagonal byte. We can also modify the characteristic by using a tweak active in a different column and rotating the whole characteristic. For instance, let us assume that we repeat the attack with 32 characteristics, 8 active in each column. If the key has at least one byte of $k_5 \oplus k_{11}$ equal to zero, then 8 of these characteristics correspond to the correct column and succeed with probability $2^{-3} \cdot (1 - 1/e)$. Therefore the success probability is $1 - \left(1 - 2^{-3} \cdot (1 - 1/e)\right)^8 \approx 0.48$ for this class of $2^{123.95}$ keys, with an attack complexity $(D, T, M) = (2^{100.5}, 2^{114}, 2^{80.5})$.

2.5.3 Attacking All Keys

In order to further extend the weak key class, we only require that the key is compatible with the tweak difference. This has a probability of 2^{-1} on each diagonal byte, which makes a total probability of 2^{-4}. In order to apply the attack to all keys, we repeat the attack with different choices of θ and different structures, until it succeeds.

The characteristic is still essentially the same, but the middle rounds now have a probability of 2^{-28} (Figure 11). Again, we start by guessing the first byte of $\widehat{k}_{10}$, denoted by K. The first pair satisfies the total characteristic with probability $2^{-116} \cdot 2^{-28} = 2^{-144}$, and the twin pair with probability $2^{-13} \cdot 2^{-28} = 2^{-41}$, so that the quadruple satisfies the characteristic with probability 2^{-185}. With structures of size $2^{92.5}$, we are left with 2^{115} candidates after the 70-bit filter. Using the table-based variant of the extended filtering of Section 2.4.6, we can filter the 2^{115} pairs down to 2^{61} with a complexity of 2^{115}, but since this attack only covers 2^{116} keys with a given guess of K, it is not faster than exhaustive search.

More efficient filtering. In order to reduce the time complexity of the attack, we use a time-data trade-off, increasing the data complexity in order to have a stronger filtering. More precisely, we start with two structures of size 2^{96} (the maximum size possible within this framework) and we only keep the 2^{88} values with $\widehat{C}_1(u)[12] = 0$ and $\widehat{C}'_1(v)[12] = 0$, respectively. As explained in Section 2.4.5, we keep 2^{82} values after the 6-bit pre-filter of conditions (4) and (5). We store the first structure in a hash table indexed by

$$i = \widehat{C}_1[0,1,2,3] \| \widehat{\widehat{C}}_1[0,1,2,3], \qquad j = \widehat{C}_1[13] \| \widehat{\widehat{C}}_1[12,13],$$

and the second structure in a hash table indexed by

$$i' = \widehat{C}'_1[0,1,2,3] \| \widehat{\overline{C}}'_1[0,1,2,3] \oplus [\lambda,0,0,0,\lambda,0,0,0], \qquad j' = \widehat{C}'_1[13] \| \widehat{\overline{C}}'_1[12,13].$$

Because of the 6-bit pre-filter, there are only 2^{58} values of i and i' with non-empty buckets, and each bucket with a fixed (i,j) (respectively (i',j')) contains on average 1 element.

We can now generate efficiently the pairs that pass both the 70-bit filter, and the first 2-bit filter used in the extended filtering of Section 2.4.6. We first iterate over all 2^{58} choices of $i = i'$ with non-empty buckets, corresponding to the 70-bit filter. For each $i = i'$, we iterate over the 2^{46} choices of j and j' such that the 2-bit filter is satisfied, and generate the corresponding pairs. Since we have $\widehat{C}_1[12] = \widehat{C}'_1[12] = 0$, the 2-bit filter only depends on j and j', and we can pre-compute the 2^{46} values that satisfy it. This generates 2^{104} pairs for a cost of $2^{58} \cdot 2^{46} = 2^{104}$.

Then we apply the rest of the extended filter to reduce to 2^{52} pairs, and exhaustively test the suggested keys (for a cost of $2^{52+2+24}$). This attack covers 2^{124} keys and has complexity $(D,T,M) = (2^{106}, 2^{112}, 2^{83})$, after iterating over K. However, since the structures are smaller than required, the success probability is only $1 - e^{-1/512}$ (we consider 2^{176} quadruples, but the probability of following the characteristic is 2^{-185}).

Expected complexity. For a random key, the previous attack succeeds with probability $2^{-4} \cdot (1 - e^{-1/512}) \approx 1/8200$. Therefore, the full attack succeeds after 8200 repetitions on average,[4] with a total complexity of $(D,T,M) = (2^{119}, 2^{125}, 2^{83})$.

We can also slightly reduce the data complexity (at the expense of memory complexity) by reusing the structures of 2^{96} queries with different constraints on $\widehat{C}_1(u)[12]$ and $\widehat{C}'_1(v)[12]$.

3 Cryptanalysis of ForkSkinny

ForkSkinny [ALP+19a, ALP+19b] is a forked variant of Skinny [BJK+16]. It is the primitive used in the ForkAE [ALP+19a] submission to the NIST lightweight standardization effort, which has been selected in the second round.

3.1 Description of ForkSkinny

Skinny. Skinny is an SPN cipher inspired by the AES, but the operations are optimized to reduce the hardware requirement. In particular, the S-Box does not have optimal properties, the round keys are added to only half of the sate (without whitening keys in the first and last rounds), and the MixColumns operation does not use an MDS matrix (it only has branch number 2).

The are several variants of Skinny, with a state size of $n = 64$ or $n = 128$ bits, and a tweakey size of n bits, $2n$ bits, or $3n$ bits. The state is considered as a 4×4 matrix of cells (bytes or nibbles depending on the state size), and the round function (shown in Figure 4) follows roughly the AES structure:

- SubCells applies an S-Box on each cell of the state;
- AddConstants adds round constants to the state;
- AddRoundTweakey adds tweakey material to the first two rows of the state;
- ShiftRows shifts the second row of the state by 1 cell, the third row by 2 cells, and the last row by 3 cells;
- MixColumns multiplies each column of the state by an invertible matrix

[4] Note that we can use 492 different characteristics (with 123 choices of θ and using the four rotations), so that the probability of success of each attempt is mostly independent.

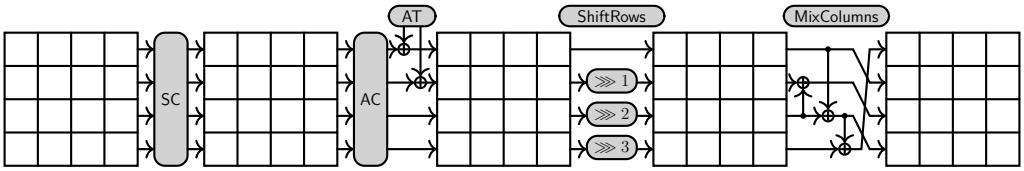


Figure 4: Skinny round function.

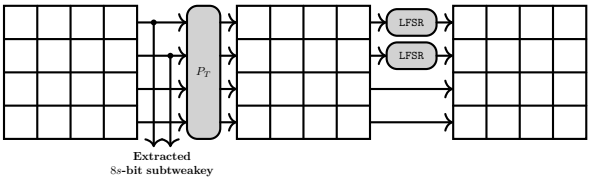


Figure 5: Skinny tweakey schedule.

Tweakey schedule. The tweakey schedule of Skinny uses the superposition construction of [JNP14]. The tweakey input (concatenation of the key and tweak) is divided into tweakey words of n bits, each word follows an independent schedule, and the subkeys are created by xoring elements from each word. For instance, with a tweakey size of $2n$ bits, the tweakey state has two words TK1 and TK2, and subkeys are constructed from the values $\mathrm{TK1} \oplus \alpha^i(\mathrm{TK2})$, where α is a linear transformation implemented with an LFSR.[5] This limits the number of steps where tweakey differences can cancel out if the LFSR has a large order.

Since Skinny uses only $n/2$ bits of key material in each round, each value $\mathrm{TK1} \oplus \alpha^i(\mathrm{TK2})$ is used for the subkeys of rounds $2i$ and $2i+1$. Step $2i$ uses the first two rows of $\mathrm{TK1} \oplus \mathrm{LFSR}^i(\mathrm{TK2})$ and permutes the cells, while step $2i+1$ uses the last two rows and permutes the cells.

Formally, the tweakey schedule for each chunk is defined as shown is Figure 5, where the permutation P_T swaps first and last rows, and permutes the cells of the first rows:

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

The key schedule uses different LFSRs for each word (with the identity for TK1), but the same permutation P_T.

ForkSkinny. ForkSkinny is a forked variant of Skinny, following the framework described earlier. Encryption starts with r_i round of Skinny, then the state is copied and processed in two branches, with r_0 and r_1 rounds, but a branch constant is added to the state before starting the second branch. Since there are no whitening keys, ForkSkinny uses $r_i + r_0 + r_1$ round keys, derived with the Skinny key schedule.

ForkSkinny was designed after the first attacks were published on ForkAES [BBJ+19], and uses $r_0 = r_1 > r_{\mathrm{tot}}/2$ to limit the impact of the reduced diffusion in reconstruction queries. For instance Skinny-128-256 has 48 rounds, but reconstruction queries against ForkSkinny-128-256 have to traverse a total of 54 rounds: 27 decryption rounds and 27 encryption rounds. The parameters suggested by the designers are given in Table 3.

As far as we know there is no previous cryptanalysis result on ForkSkinny, but previous analysis of Skinny can directly be applied to ForkSkinny.

[5] With a tweakey size of $2n$ bits, subkeys are constructed from the values $\mathrm{TK1} \oplus \alpha^i(\mathrm{TK2}) \oplus \beta^i(\mathrm{TK3})$.

Table 3: ForkSkinny parameters.

Primitive	block	tweak	tweakey	r_i	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

3.2 Notation

We denote the state at the input of round i as x_i, the state after SubCells as y_i, after AddRoundTweakey as z_i and after ShiftRows as w_i. The output of MixColumns is x_{i+1}. In our description of the attack we ignore the AddConstants operation and the branch constant for simplicity, since they do not impact differential attacks. The round subkey of round i (generated from the tweakey by the tweakey schedule) is denoted as tk_i, while the input tweakey words are denoted as $TK1$ and $TK2$.

For a 128-bit state s, we denote by $s[j]$ the j-th byte of the state, with the following byte-order (following the Skinny specification):

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

Equivalent first and last round. Since there are no whitening keys in Skinny, we can ignore the SubCells operation of the first round, and the ShiftRows and MixColumns operations in the last round, because they can be evaluated without knowing the key.

We also define an equivalent first round key that is applied after ShiftRows and MixColumns, so that we can also ignore those operations:

$$\widehat{tk}_1 = \mathsf{MC}(\mathsf{SR}(tk_1)).$$

3.3 Related-tweakey Attacks on Skinny

The most efficient attacks against Skinny are based on related-tweakey differential trails with some inactive rounds in the middle. Starting from the middle, differences are introduced by the tweakey, and propagated outwards. Therefore it is important to understand how many consecutive round keys can be inactive. In particular, impossible differential attacks use two independent trails, and each trail can take advantage of inactive rounds.

TK1 trails with two inactive rounds. When there is a single tweakey word with a difference, the analysis is rather simple. If there is an active byte in the master tweakey, it will alternatively move to the top and bottom half of the tweakey state, and every second round key has a non-zero difference. Therefore, we can have two consecutive inactive rounds. This is illustrated by Figure 6.

TK2 trails with four inactive rounds. When there are two tweakey words with a difference (*e.g.* a related-key attack on Skinny-128-256), the analysis is more complex. The attacker can choose differences in each tweakey word that will cancel at some intermediate round. This gives three consecutive inactive round keys, and four consecutive inactive rounds. Since all tweakey words have the same cell permutation in the tweakey schedule, the difference stays in a single cell of the round keys. This is illustrated by Figure 7.

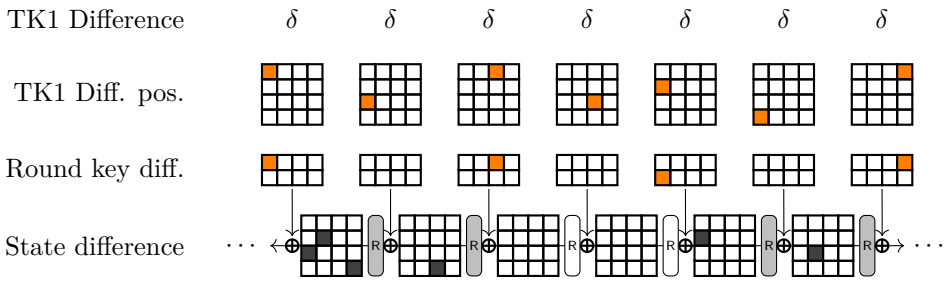


Figure 6: Skinny path construction with a single active tweakey word.

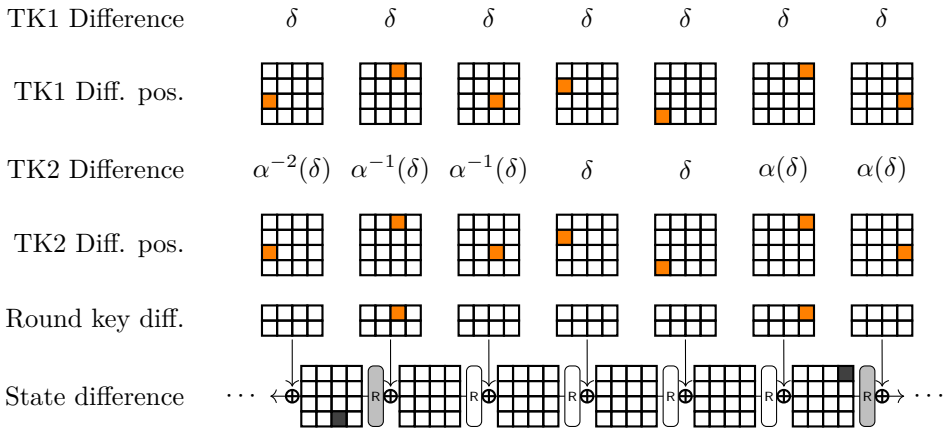


Figure 7: Skinny path construction with two active tweakey words.

Alternatively we can use the following table to represent the tweakey differences, where values in square brackets indicate that the difference is in the lower part of the state:

Round	2	3	4	5	6
TK1	δ	$[\delta]$	δ	$[\delta]$	δ
TK2	$\alpha^{-1}(\delta)$	$[\alpha^{-1}(\delta)]$	δ	$[\delta]$	$\alpha(\delta)$
Subkey	$\alpha^{-1}(\delta) \oplus \delta$	$\mathbf{0}$	$\mathbf{0}$	$\mathbf{0}$	$\alpha(\delta) \oplus \delta$

Thanks to the LFSR construction used in the tweakey schedule, it is possible to prove that such a cancellation can only happen every 30 rounds. More precisely, the smallest i such that there exist values $\delta \neq 0$ with $\alpha^i(\delta) = \delta$ is $i = 15$.

3.4 Related-tweakey Attacks on ForkSkinny

We now move from Skinny to ForkSkinny, and we focus on encryption queries because reconstruction queries have extra rounds to make them less useful to an attacker.

Encryption queries from P to C_0 are exactly equivalent to Skinny encryption, therefore attacks exploiting only those queries are not easier on ForkSkinny. However, encryption queries from P to C_1 use a slightly different tweakey schedule: the subkeys used are taken from indices $0, 1, \ldots, r_i - 1, r_i + r_0, r_i + r_0 + 1, \ldots r_i + r_0 + r_1 - 1$. To put it another way, there are r_0 "blank" rounds of key schedule at the forking point. This leads to a new class of weaknesses if the value of r_0 is poorly chosen, because cancellation in the round keys can occur more frequently than in the normal Skinny key schedule.

3.4.1 Extending Attacks by One Round when r_0 is Odd

When r_0 is odd, the round keys before and after the forking point are taken from the same half of the master tweakey. In particular, if there is no difference in this half, we have two consecutive inactive round keys. This allows to extend most differential trails (and therefore differential attacks) by one round.

For instance, attacks in the TK2 setting can use the following differences, with 4 consecutive inactive round keys rather than 3 (with $r_0 = 2t + 1$):

Round	1	2	3	4	$(2t+1)+5$	$(2t+1)+6$
TK1	δ	$[\delta]$	δ	$[\delta]$	$[\delta]$	δ
TK2	$\alpha^{-1}(\delta)$	$[\alpha^{-1}(\delta)]$	δ	$[\delta]$	$[\alpha^{t+1}(\delta)]$	$\alpha^{t+2}(\delta)$
Subkey	$\alpha^{-1}(\delta) \oplus \delta$	**0**	**0**	**0**	**0**	$\alpha^{t+2}(\delta) \oplus \delta$

In particular, all parameters proposed for ForkSkinny use odd values of r_0 which make this type of attack possible. The designers explained to us that this was a deliberate choice, because an even value of r_0 would give a similar property for reconstruction queries.

In general, taking advantage of this property requires to design a dedicated attack on the primitive because the blank rounds move the position of the active tweakey. However, the use of $r_1 = 31$ in ForkSkinny-128-288 makes it easier to reuse existing attacks because the cell permutation P_T has a period of 16. More precisely, the first active step after the blank rounds is step $2t + 7 = 37$ in the previous figure. Therefore the active tweakey is in the same position as at round 5, which would be the first active step in an attack on Skinny. This allows to reuse the same trails when extending an existing attack by one round (the same type of reuse will be shown in detail in Section 3.5).

3.4.2 Extending Attacks by Three Rounds when $r_0 = 27$

For some specific values of r_0, we can have even more consecutive inactive rounds. In particular, it is easy to see that the previous pattern leads to 6 inactive round keys if $\alpha^{t+2}(\delta) = \delta$. For the particular α corresponding to the LFSR used in Skinny, there are some choices of δ such that $\alpha^{15}(\delta) = \delta$. We denote by $\mathcal{S}$ the set of those values[6] (15 non-zero values, and the zero value). When $r_0 = 27$, and using such a δ, we have differential characteristics with 6 inactive round keys rather than 3 in the TK2 setting:

Round	1	2	3	4	27+5	27+6	27+7	27+8
TK1	δ	$[\delta]$	δ	$[\delta]$	$[\delta]$	δ	$[\delta]$	δ
TK2	$\alpha^{-1}(\delta)$	$[\alpha^{-1}(\delta)]$	δ	$[\delta]$	$[\alpha^{14}(\delta)]$	$\alpha^{15}(\delta)$	$[\alpha^{15}(\delta)]$	$\alpha^{16}(\delta)$
Subkey	$\alpha^{-1}(\delta) \oplus \delta$	**0**	**0**	**0**	**0**	**0**	**0**	$\alpha^{16}(\delta) \oplus \delta$

As it turns out, ForkSkinny-128-256 uses precisely $r_0 = 27$, therefore we expect that attacks against this variant can be extended by three rounds. However, this generally requires to adapt the attacks and to repeat the analysis, because the difference at round $27 + 8$ is now at a different position than it would have been at round 5 in the original Skinny attack.

We now present two explicit related-key attacks taking advantage of this property: a 24-round attack against ForkSkinny-128-256 when used with a 128-bit key and a 128-bit tweak, and a 26-round attack against ForkSkinny-128-256 when used with a 256-bit key. Both attacks reach about 3 more rounds than the best attack against Skinny with the same parameters.

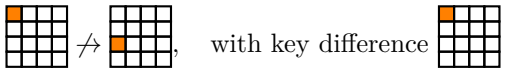
[6] This corresponds to all values for ForkSkinny variants with 4-bit cells.

3.5 A 24-round Attack on ForkSkinny-128-256 with 128-bit key

We first focus on ForkSkinny-128-256 when used with a 128-bit key and a 128-bit tweak, which corresponds to the usage in the ForkAE NIST submission.

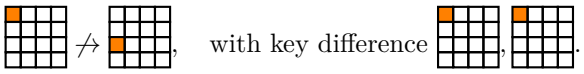
In this setting, we have a very interesting result: we can reproduce exactly the attack against Skinny-128-128 presented in [SMB18, Appendix A] to attack ForkSkinny-128-256, gaining 5 rounds. The initial attack is a related-key attack on ForkSkinny-128-128 based on trails with two inactive rounds (following the general idea of Figure 6). We turn it into a related-key related-tweak attack on ForkSkinny-128-256 with a 128-bit key, based on trails with seven inactive rounds as given in Section 3.4.2.

More precisely, the existing attack on Skinny-128-128 uses a 13-round impossible differential distinguisher build with the miss-in-the-middle technique, where the upper characteristic starts with 2 inactive rounds [SMB18, Figure 3]:



There are 255 specific values of the differences for which the impossible differential holds (see [SMB18] for details). The 13-round distinguisher is used to attack 19 rounds, by adding three rounds at the top, and three at the bottom.

In our attack we reuse this distinguisher, but we have 7 inactive rounds at the beginning, using two cancellations in the tweakey schedule (taking advantage of $r_0 = 27$). The new distinguisher has 5 extra rounds, plus 27 blank rounds in the key schedule. Since we add 32 key schedule rounds compared to the initial attack, the position of the active tweakey differences after the cancellations are the same as in the original attack, because the order of P_T is 16. Therefore, we have the following 18-round impossible differential distinguisher for ForkSkinny-128-256 versions with $r_0 = 27$, with the forking point after 4 rounds:



Again, the active bytes must satisfy some relations, and the key differences must be chosen in $\mathcal{S}$. More precisely, we have 15 impossible differentials, when the difference in TK1 is $\delta \neq 0 \in \mathcal{S}$, the difference in TK2 is $\alpha^{-1}(\delta)$, the difference at the input is $\alpha^{-1}(\delta) \oplus \delta$, and the difference at the output is $\alpha^6(\delta) \oplus \delta$. The full impossible differential is shown in Figure 8.

Since the impossible differential has the same shape as in the original Skinny attack, we can also reuse the key recovery part of the attack, adding 3 rounds in the backward and forward directions. Therefore, we attack a 24-round reduced version of ForkSkinny-128-256 with 128-bit key where $r_i = 7$, $r_0 = 27$ and $r_1 = 17$, with the same complexity as the attack of [SMB18, Appendix A]:

$$(D, T, M) = (2^{122.5}, 2^{125.5}, 2^{97.5}).$$

Unfortunately, we have not found an attack against Skinny-128-256 with 128-bit key in the literature to directly compare the security of Skinny and ForkSkinny, but it seems reasonable to expect that this type of attack would reach 21 rounds.[7]

More generally, we can reuse most of the differential attacks on Skinny-128-128 based on trails with two inactive rounds, and extend them by 5 rounds when attacking ForkSkinny-128-256 with a 128-bit key and $r_0 = 27$.

[7] Using a cancellation in the key schedule allows trails with four inactive rounds, so this type of impossible differential attack should reach 21 rounds. However, building the concrete attack requires some tedious work to verify the impossible differential and the key-recovery.

3.6 A 26-round Attack on ForkSkinny-128-256 with 256-bit key

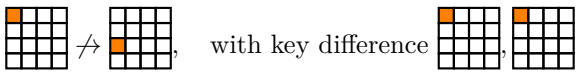
In order to show an attack against the largest number of rounds possible, we now assume the use of a 256-bit key (without tweak), and we allow the attacker to use related keys. This setting has been studied by several previous works in the case of Skinny-128-256 [SMB18, LGS17], and the best known attack reaches 23 rounds. Starting from one of those 23-round attacks, we build a 26-round attack against ForkSkinny-128-256, using the tweakey cancellation property with $r_0 = 27$. We follow the trail suggested in Section 3.4.2, with three inactive round keys before the forking point, and three more after the forking point. Our attack is an impossible differential attack following the blueprint of [SMB18, Section 4.3]. We briefly recall the basics of impossible differential attacks, but we point the reader to the paper of Boura, Naya and Suder [BNS14] for detailed explanations.

We first build an impossible differential trail with the miss-in-the-middle technique. The forward and the backward trails start with a fixed difference that cancels with the first round key difference, leading to some inactive rounds. In the forward trail, we use tweakey cancellation to have seven inactive rounds, while we have two inactive rounds in the backward trail, using an inactive round key when the tweakey difference is in the lower half of the state. Since all the active rounds are located after the forking point, we can actually use the same characteristic as used in [SMB18], changing only the inactive rounds at the beginning. The impossible differential is shown in Figure 8, it spans 18 round, but the first round does not include the SubCells operation (the S-Box layer). As expected this distinguisher gains 3 rounds compared to the attack of [SMB18].

Next, we use the distinguisher in a key-recovery attack. Following [SMB18], we have 3 rounds before the distinguisher, and 5 rounds after. The key-recovery part of our attack is very similar to the attack of [SMB18], but it has to be modified because the input difference is now in cell 0 instead of cell 1. The new key-recovery is illustrated by Figure 9 and is detailed in the next subsections.

3.6.1 Description of the Attack

An impossible differential attack starts from a distinguisher. In our case we have 15 related-key impossible differences for 18 rounds, of the form:



when the difference in TK1 is $\delta \neq 0 \in \mathcal{S}$, the difference in TK2 is $\alpha^{-1}(\delta)$, the difference at the input is $\alpha^{-1}(\delta) \oplus \delta$, and the difference at the output is $\alpha^6(\delta) \oplus \delta$.

The distinguisher is placed at rounds 4 to 21, and we collect a number of plaintext/ciphertext pairs under related keys with the required differences. Then we process each pair to find whether some keys would lead to the input and output differences of the impossible differential characteristic at rounds 4 and 21; if we locate such keys we know that they cannot be the actual encryption key. After processing enough data, we expect that only a small fraction of the keyspace remains. More precisely, the attack will process several structures of plaintexts, such that each structure will rule out a number of key candidates.

A structure is generated by fixing all bytes of the plaintext except bytes 1,4,11,14 to a random value, and taking the 2^{32} possible values of bytes 1,4,11, and 14. Since the attack uses related keys, we have to encrypt each structure under several keys. More precisely, we use 16 different keys, and we encrypt each plaintext under key $k_1^* \oplus [0, \ldots, 0, \delta], k_2^* \oplus [0, \ldots, 0, \alpha^{-3}(\delta)]$ (where $k_1^* \| k_2^*$ is the secret key), for all $\delta \in \mathcal{S}$, because byte 15 of the master key moves to byte 0 at round 4 for the impossible differential distinguisher.

By linearity of the LFSR α, any pair of tweakey in this set satisfies the conditions for the impossible differential distinguisher, with a difference δ in both words of the tweakey state tk_6 (the TK1 word is updated three times through the LFSR).

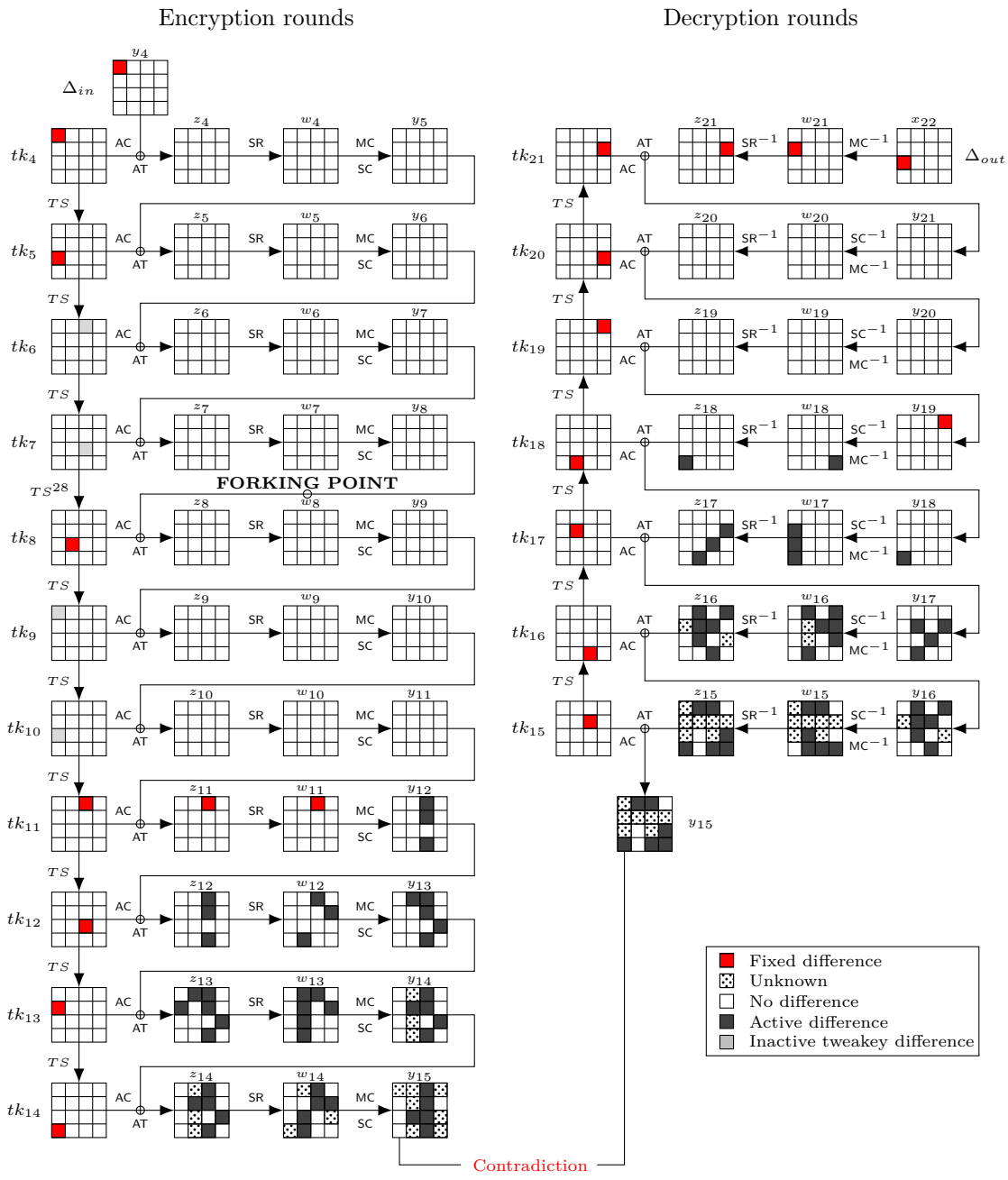


Figure 8: Impossible differential characteristic for 18-round ForkSkinny-128-256, using the second branch (4 rounds before forking point, 14 rounds after forking point).

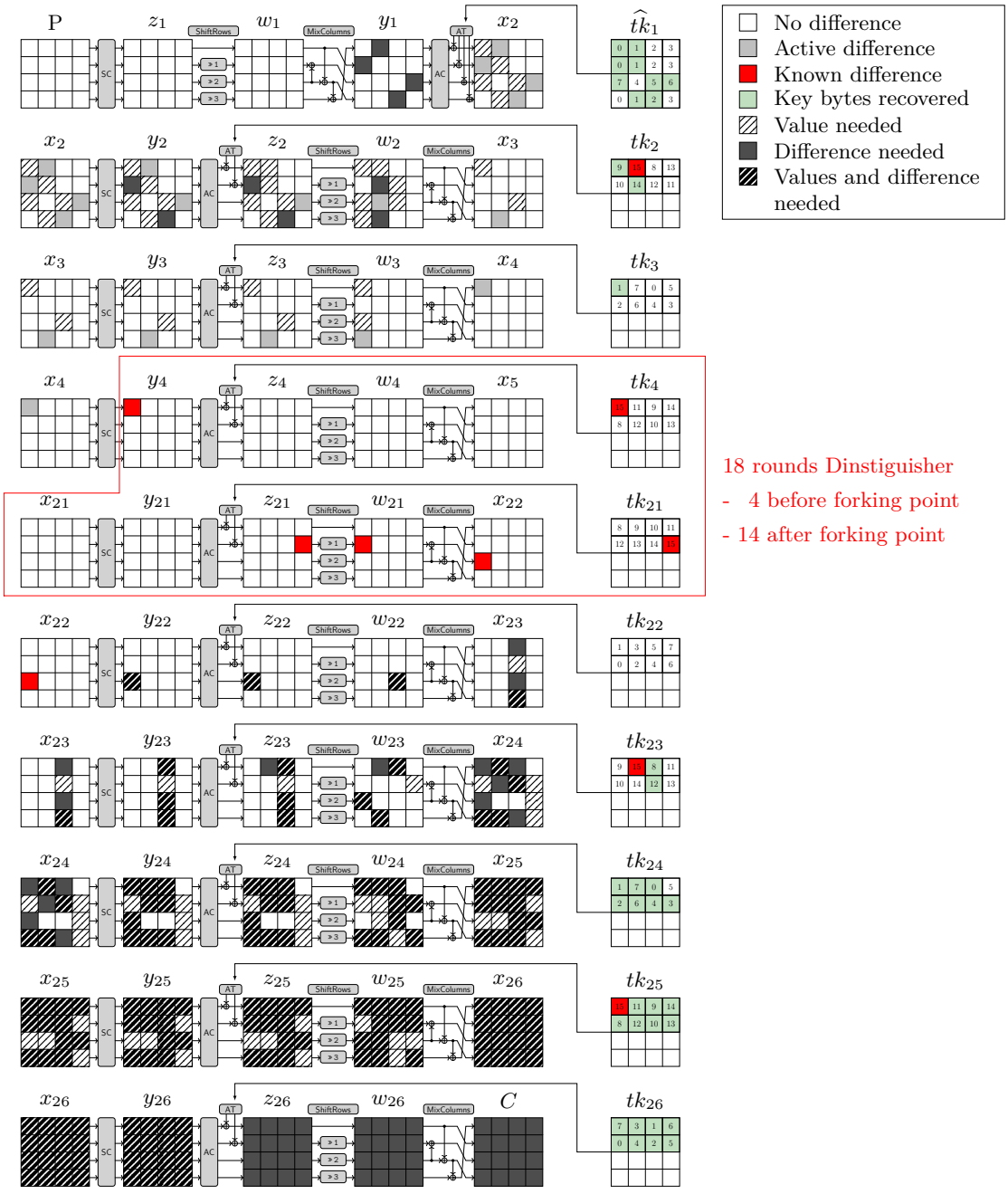


Figure 9: Tweakey recovery of the impossible differential attack of ForkSkinny-128-256 on 26 rounds.

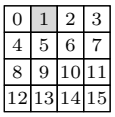
Each structure contains 2^{36} plaintext and values, and there are roughly $2^{36+35} = 2^{71}$ useful pairs in each set. In total, we use N pairs, *i.e.* $N/2^{71}$ different structures.

3.6.2 Processing the Pairs

We now explain how to process each pair to identify the keys that lead to the impossible differential. The following key recovery procedure is inspired by [SMB18]. We attach partial key information to each of the N pairs collected, initially empty, and we will incrementally fill up the key information.

1. **Round 1**. From the fixed difference $\Delta y_2[1] = \Delta tk_2[1]$ and the difference $\Delta x_2[1]$ derived from the plaintext, use Lemma 1 to deduce $\widehat{tk}_1[1]$ ($TK[1]$).

   We can represent the knowledge about the key graphically:

   Cells colored in light gray have been recovered once by the attacker (he does not know the values of both tweakey states $TK1$ and $TK2$). Cells colored in dark gray have been recovered twice by the attacker, so he can freely recover this tweakey byte at any round (he knows the corresponding cells in $TK1$ and $TK2$ by linearity). We will use this representation after each step of the process.

2. **Round 26**. Because $\Delta w_{25}[4, 10, 11, 14] = 0$, the MixColumns operation gives us four equations on Δx_{26}:

   (i) $$\Delta w_{25}[4] = \Delta x_{26}[4] \oplus \Delta x_{26}[12] \oplus \Delta x_{26}[8] = 0$$

   From the knowledge of the ciphertext, we can compute $\Delta x_{26}[8, 12]$ since no key material is added on bytes 8 and 12. Then, we compute the quantity $\Delta x_{26}[4]$. Therefore we know both $\Delta x_{26}[4]$ and $\Delta y_{26}[4]$. Using Lemma 1, we can determine $tk_{26}[4]$ ($TK[0]$).

   (ii) $$\Delta w_{25}[14] = \Delta x_{26}[2] \oplus \Delta x_{26}[14] = 0$$

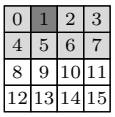
   (iii) $$\Delta w_{25}[10] = \Delta x_{26}[6] \oplus \Delta x_{26}[14] = 0$$

   $\Delta x_{26}[14]$ can be computed from the ciphertext to derive $\Delta x_{26}[2]$ and $\Delta x_{26}[6]$. Then we can apply Lemma 1 and determine $tk_{26}[2, 6]$ ($TK[1, 2]$).

   (iv) $$\Delta w_{25}[11] = \Delta x_{26}[7] \oplus \Delta x_{26}[15] = 0$$

   $\Delta x_{26}[15]$ can be computed from the ciphertext to derive $\Delta x_{26}[7]$. We use Lemma 1 to determine $tk_{26}[7]$ ($TK[5]$).

   We then guess $tk_{26}[0, 1, 3, 5]$ ($TK[3, 4, 6, 7]$) to compute the full state x_{26}. This step has a complexity of $N \times 2^{4\times 8}$ and we are left with $N \times 2^{4\times 8}$ candidates.

3. **Round 25**. We have $\Delta w_{24}[8, 9, 14] = 0$, the following equations can be derived:

(i) $$\Delta w_{24}[14] = \Delta x_{25}[2] \oplus \Delta x_{25}[14] = 0$$
(ii) $$\Delta w_{24}[8] = \Delta x_{25}[4] \oplus \Delta x_{25}[12] = 0$$
(iii) $$\Delta w_{24}[9] = \Delta x_{25}[5] \oplus \Delta x_{25}[13] = 0$$

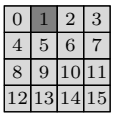
Since we can compute $\Delta x_{25}[12, 13, 14]$ from x_{26}, we can deduce $\Delta x_{25}[2, 4, 5]$ and apply Lemma 1 to recover $tk_{25}[2, 4, 5]$ ($TK[8, 9, 12]$).

We then guess $tk_{25}[3, 6, 7]$ ($TK[10, 13, 14]$) and compute the rightmost two columns of w_{24}. We can then compute $\Delta x_{24}[8, 12]$ from the values and differences of $w_{24}[10, 15]$. Since $w_{23}[0, 4] = 0$, we have an 8-bit filter:

$$\Delta w_{23}[0] \oplus \Delta w_{23}[4] = \Delta x_{24}[8] \oplus \Delta x_{24}[12] = 0.$$

Next, we guess $tk_{25}[0, 1]$ ($TK[11, 15]$), which allows to compute the full state x_{25}.

The complexity of this step is $2^{4\times 8}$ per candidate left after the previous step, therefore $N \times 2^{8\times 8}$, and there are $N \times 2^{8\times 8}$ remaining candidates.



4. **Round 24**. Having $\Delta w_{23}[9, 10, 12, 14] = 0$, the following equations can be derived:

(i) $$\Delta w_{23}[14] = \Delta x_{24}[14] \oplus \Delta x_{24}[2] = 0$$
(ii) $$\Delta w_{23}[10] = \Delta x_{24}[14] \oplus \Delta x_{24}[6] = 0$$

We can compute $\Delta x_{24}[14]$ from x_{25}, deduce $\Delta x_{24}[2, 6]$ and recover $tk_{24}[2, 6]$ ($TK[0, 4]$) using Lemma 1.

(iii) $$\Delta w_{23}[9] = \Delta x_{24}[5] \oplus \Delta x_{24}[13] = 0$$

We can compute $\Delta x_{24}[13]$ from x_{25}, deduce $\Delta x_{24}[5]$ and recover $tk_{24}[5]$ ($TK[6]$) using Lemma 1. Since the difference $\Delta w_{23}[1]$ cancels out with the tweakey, we have an 8-bit filter:

$$\Delta tk_{23}[1] = \Delta w_{23}[1] = \Delta x_{24}[5].$$

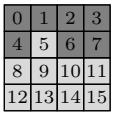
Since $tk_{24}[0]$ ($TK[1]$) has already been recovered twice, the attacker can compute $\Delta x_{24}[0]$ and the last equation becomes an 8-bit condition:

(iv) $$\Delta w_{23}[12] = \Delta x_{24}[0] \oplus \Delta x_{24}[12] = 0.$$

We then guess $tk_{24}[1, 4, 7]$ ($TK[2, 3, 7]$), and compute the values and difference of $x_{23}[10, 14]$ from $w_{24}[8, 13]$. Because $\Delta w_{22}[2, 6] = 0$, we have an 8-bit filter:

$$\Delta w_{22}[2] \oplus \Delta w_{22}[6] = \Delta x_{23}[10] \oplus \Delta x_{23}[14] = 0.$$

This step has a complexity of $2^{5\times 8}$ per candidate from the previous step, therefore a total complexity of $N \times 2^{9\times 8}$. At the end of this step, there are $N \times 2^{8\times 8}$ candidates.



5. **Round 1**. Since we know $\widehat{tk}_1[4, 11, 14]$ ($TK[0, 2, 6]$) from the previous steps, we can compute $\Delta w_2[5, 9, 13]$. Because $\Delta x_3[1, 5, 9] = 0$, we have a 16-bit filter:

   (i) $$\Delta x_3[9] = \Delta w_2[5] \oplus \Delta w_2[9] = 0$$

   (ii) $$\Delta x_3[1] \oplus \Delta x_3[5] = \Delta w_2[9] \oplus \Delta w_2[13] = 0.$$

   This step has a complexity of 1 per candidate, therefore a total complexity of $N \times 2^{8\times 8}$. We are left with $N \times 2^{6\times 8}$ candidates.

6. **Round 23**.Because $\Delta w_{22}[14] = 0$, the following equation can be derived:

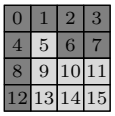
   (i) $$\Delta w_{22}[14] = \Delta x_{23}[2] \oplus \Delta x_{23}[14] = 0.$$

   We can compute $\Delta x_{23}[14]$ from x_{24} and deduce $\Delta x_{23}[2]$. Using Lemma 1, we recover $tk_{23}[2]$ ($TK[8]$).

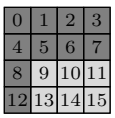
   Next, we guess $tk_{23}[6]$ ($TK[12]$). $\Delta z_{21}[7]$ cancels out with the tweakey, therefore we have an 8-bit filter:

   $$\Delta tk_{21}[7] = \Delta z_{21}[7] = \Delta x_{22}[8].$$

   This step has a complexity of 2^8 per candidate, therefore a total complexity of $N \times 2^{7\times 8}$, and we are left with $N \times 2^{6\times 8}$ candidates.



7. **Round 1**. Guess $\widehat{tk}_1[10]$ ($TK[5]$). Since $\widehat{tk}_1$ is fully known, we can fully compute y_2. This step has a complexity of 2^8 per candidate, therefore a total complexity of $N \times 2^{7\times 8}$, and we are left with $N \times 2^{7\times 8}$ candidates.



8. **Round 2 and 3**. Guess $tk_2[1, 5]$ ($TK[14, 15]$). We know $w_3[8, 12]$ and $\Delta w_3[12]$ from $x_3[8, 13]$ and $\Delta x_3[13]$. From $\Delta w_3[12]$ we can compute $\Delta x_4[0]$. Additionally, we know $\Delta y_4[0] = \Delta tk_4[0]$, so we can apply Lemma 1, giving us one possible value in average for $x_4[0]$. We derive the following equation from the round 3 MixColumns operation:

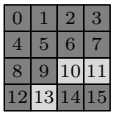
   $$x_4[0] = w_3[0] \oplus w_3[8] \oplus w_3[12].$$

   Knowing $w_3[8, 12]$ and $x_4[0]$, we can deduce $w_3[0]$. We also know $tk_3[0]$ ($TK[1]$) from the previous steps, so we can compute $y_3[0]$ then $x_3[0]$. We derive the following equation from the round 2 MixColumns operation:

   $$x_3[0] = w_2[0] \oplus w_2[8] \oplus w_2[12].$$

   Knowing $w_2[8, 12]$ we can recover $w_2[0] = z_2[0]$. We get $tk_2[0]$ ($TK[9]$).

   This step has a complexity of $2^{2\times 8}$ per candidate, therefore a total complexity of $N \times 2^{9\times 8}$, and we are left with $N \times 2^{9\times 8}$ candidates.



3.6.3 Complexity Analysis

The total number of key bytes recovered is 29, and we deduce $2^{9\times 8}$ impossible keys from each pair. In order to reduce the key space by at least a factor $1/e$, the number of pairs N should be such that:

$$N \times 2^{9\times 8} \geq 2^{29\times 8}.$$

This implies $N \geq 2^{160}$, therefore we need to construct at least $2^{160-71} = 2^{89}$ sets, using $D = 2^{89+36} = 2^{125}$ plaintexts. The analysis phase has a complexity of $N \times 2^{9\times 8} = 2^{232}$ so the time complexity will be dominated by the exhaustive search over the remaining key space, with $T = 2^{256}/e \approx 2^{254.6}$. A naive implementation would require a memory of $2^{29\times 8}$ to store all the potential keys and remove impossible ones. However an implementation using the early abort technique of [LKKD08] only requires to store the plaintext/ciphertext pairs. Indeed, Algorithm 1 gives an implementation of the attack were we store the pairs and iterate over the possible key bytes values to perform the key recovery.

Therefore, we end up with a complexity of

$$(D, T, M) = (2^{160}, 2^{254.6}, 2^{160}).$$

Complexity parameters from [BNS14]. We can verify our complexity analysis using the generic formula of [BNS14]. The parameters corresponding to our attack are:

$$|\Delta_{\text{in}}| = 4.5 \times 8 \qquad |\Delta_{\text{out}}| = 16 \times 8$$
$$c_{\text{in}} = 4 \times 8 \qquad c_{\text{out}} = 16 \times 8$$
$$|k_{\text{in}} \cup k_{\text{out}}| = 29 \times 8$$

The formula for the minimum data complexity $D_{\min}$ given in [BNS14] confirms our analysis:

$$D_{\min} = N_{\min} \times 2^{n+1-|\Delta_{\text{in}}|-|\Delta_{\text{out}}|} = 2^{c_{\text{in}}+c_{\text{out}}} \times 2^{n+1-|\Delta_{\text{in}}|-|\Delta_{\text{out}}|} = 2^{125}.$$

We can also reduce the time complexity, at the expanse of an increase in the data complexity. For instance with a data complexity of $N = 2^{127}$, we reduce the fraction of remaining keys to $P \approx e^{-4} \approx 2^{-5.7}$, so that the time complexity is reduced to $2^{250.3}$.

Conclusion

This work shows that the security of forkciphers must be carefully analyzed. While basic security arguments indicate that forking a (tweakable) block cipher should not reduce its security, we have studied the two concrete instances proposed, and found a significant security drop. Even though forkciphers reuse the round function of a block cipher, and keep the same number of rounds, the forkcipher model gives extra freedom to the attacker.

In the case of ForkAES, the security loss comes from the reduced diffusion when alternating decryption rounds and encryption rounds in reconstruction queries, as identified by previous works [BBJ+19]. Our improved analysis can attack the full ForkAES using reconstruction queries with 5 decryption rounds followed by 5 encryption rounds, even though the best attacks on AES only reach 7 rounds.

In the case of ForkSkinny, the security loss comes from the modified tweakey schedule in encryption queries going to the second branch. In particular the parameters chosen by the designers interact badly with properties of the Skinny tweakey schedule, allowing some attacks to be extended. More precisely, when r_0 is an odd value, two consecutive rounds use the same half key, and when r_0 is 27 we can have six consecutive rounds without tweakey difference in the TK2 model, rather than three. Since one of the ForkSkinny parameters uses $r_0 = 27$, we can extend the best attacks to three more rounds. The best attack on Skinny-128-256 reaches 23 rounds, but we can break ForkSkinny-128-256-7-27-19,

a reduced version of the recommended ForkSkinny-128-256-21-27-27 with 26 rounds. Other values of r_0 also lead to more cancellations than in the original Skinny, but exploiting them requires to build dedicated attacks and we leave this to future work. In particular, it would be interesting to study attacks against reduced version with $r_0 = r_1$ which is more natural than the reduced versions studied in this paper.

Acknowledgments

This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement no. 714294 - acronym QUASYModo).

Algorithm 1 Implementation of this attack with early abort and low memory.
tk represents the accumulated knowledge about the key.

Input: L_0 with $|L_0| = N$
  for all $tk_{26}[0,1,3,5], tk_{25}[3,6,7]$ **do**
    $L_1 \leftarrow \{\}$
    for all $(p,c) \in L_0$ **do** ▷ $2^{7s} \times N$ iterations
      Deduce $\widehat{tk}_1[1], tk_{26}[2,4,6,7], tk_{25}[2,4,5]$
      if $\Delta x_{24}[8] = \Delta x_{24}[12]$ **then**
        Append$(L_1, (p,c,tk))$
        ▷ $|L_1| = 2^{-s} \times N$
    for all $tk_{25}[0,1]$ **do**
      $L_2 \leftarrow \{\}$
      for all $(p,c,tk) \in L_1$ **do** ▷ $2^{8s} \times N$ iterations
        Deduce $tk_{24}[2,5,6]$
        if $\Delta x_{24}[13] = \Delta tk_{23}[1]$ **and** $\Delta x_{24}[0] = \Delta x_{24}[8]$ **then**
          Append$(L_2, (p,c,tk))$
          ▷ $|L_2| = 2^{-3s} \times N$
      for all $tk_{24}[1,4,7]$ **do**
        $L_3 \leftarrow \{\}$
        for all $(p,c,tk) \in L_2$ **do** ▷ $2^{9s} \times N$ iterations
          Deduce $tk_{23}[2]$
          if $\Delta x_{23}[10] = \Delta x_{23}[14]$ **and** $\Delta w_2[5] = \Delta w_2[9] = \Delta w_2[13]$ **then**
            Append$(L_3, (p,c,tk))$
            ▷ $|L_3| = 2^{-6s} \times N$
        for all $tk_{23}[6]$ **do**
          $L_4 \leftarrow \{\}$
          for all $(p,c,tk) \in L_3$ **do** ▷ $2^{7s} \times N$ iterations
            if $\Delta x_{22}[8] = \Delta tk_{21}[7]$ **then**
              Append$(L_4, (p,c,tk))$
              ▷ $|L_4| = 2^{-7s} \times N$
          for all $\widehat{tk}_1[10], tk_2[1,5]$ **do**
            Create hash table H indexed by 13 deduced keys words
            for all $(p,c,tk) \in L_4$ **do** ▷ $2^{9s} \times N$ iterations
              Deduce $tk_2[0]$
              $H[tk] \leftarrow 1$
            for all tk **do** ▷ 2^{29s} iterations
              if $H[tk] = 0$ **then**
                Run exhaustive search over 2^{3s} keys

References

[ALP+19a] Elena Andreeva, Virginie Lallemand, Antoon Purnal, Reza Reyhanitabar, Arnab Roy, and Damian Vizár. ForkAE v.1. Submission to the NIST Lightweight Cryptography standardization process, 2019.

[ALP+19b] Elena Andreeva, Virginie Lallemand, Antoon Purnal, Reza Reyhanitabar, Arnab Roy, and Damian Vizár. Forkcipher: A new primitive for authenticated encryption of very short messages. In Steven D. Galbraith and Shiho Moriai, editors, *ASIACRYPT 2019, Part II*, volume 11922 of *LNCS*, pages 153–182. Springer, Heidelberg, December 2019.

[ARVV18] 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.

[BBJ+19] Subhadeep Banik, Jannis Bossert, Amit Jana, Eik List, Stefan Lucks, Willi Meier, Mostafizar Rahman, Dhiman Saha, and Yu Sasaki. Cryptanalysis of ForkAES. In Robert H. Deng, Valérie Gauthier-Umaña, Martín Ochoa, and Moti Yung, editors, *ACNS 19*, volume 11464 of *LNCS*, pages 43–63. Springer, Heidelberg, June 2019.

[BC18] Christina Boura and Anne Canteaut. On the boomerang uniformity of cryptographic sboxes. *IACR Trans. Symm. Cryptol.*, 2018(3):290–310, 2018.

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

[BNS14] Christina Boura, María Naya-Plasencia, and Valentin Suder. Scrutinizing and improving impossible differential attacks: Applications to CLEFIA, Camellia, LBlock and Simon. In Palash Sarkar and Tetsu Iwata, editors, *ASIACRYPT 2014, Part I*, volume 8873 of *LNCS*, pages 179–199. Springer, Heidelberg, December 2014.

[CHP+18] Carlos Cid, Tao Huang, Thomas Peyrin, Yu Sasaki, and Ling Song. Boomerang connectivity table: A new cryptanalysis tool. In Jesper Buus Nielsen and Vincent Rijmen, editors, *EUROCRYPT 2018, Part II*, volume 10821 of *LNCS*, pages 683–714. Springer, Heidelberg, April / May 2018.

[DEM16] Christoph Dobraunig, Maria Eichlseder, and Florian Mendel. Square attack on 7-round kiasu-BC. In Mark Manulis, Ahmad-Reza Sadeghi, and Steve Schneider, editors, *ACNS 16*, volume 9696 of *LNCS*, pages 500–517. Springer, Heidelberg, June 2016.

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

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

[DL17] Christoph Dobraunig and Eik List. Impossible-differential and boomerang cryptanalysis of round-reduced kiasu-BC. In Helena Handschuh, editor, *CT-RSA 2017*, volume 10159 of *LNCS*, pages 207–222. Springer, Heidelberg, February 2017.

[DR13] Joan Daemen and Vincent Rijmen. *The design of Rijndael: AES – The Advanced Encryption Standard*. Springer Science & Business Media, 2013.

[JNP14] 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.

[LGS17] 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.

[LKKD08] Jiqiang Lu, Jongsung Kim, Nathan Keller, and Orr Dunkelman. Improving the efficiency of impossible differential cryptanalysis of reduced Camellia and MISTY1. In Tal Malkin, editor, *CT-RSA 2008*, volume 4964 of *LNCS*, pages 370–386. Springer, Heidelberg, April 2008.

[MAY16] Tolba Mohamed, Abdelkhalek Ahmed, and Amr Youssef. A meet in the middle attack on reduced round Kiasu-BC, 2016.

[MDRMH10] Hamid Mala, Mohammad Dakhilalian, Vincent Rijmen, and Mahmoud Modarres-Hashemi. Improved impossible differential cryptanalysis of 7-round AES-128. In Guang Gong and Kishan Chand Gupta, editors, *INDOCRYPT 2010*, volume 6498 of *LNCS*, pages 282–291. Springer, Heidelberg, December 2010.

[SMB18] Sadegh Sadeghi, Tahereh Mohammadi, and Nasour Bagheri. Cryptanalysis of reduced round SKINNY block cipher. *IACR Trans. Symm. Cryptol.*, 2018(3):124–162, 2018.

A Truncated Characteristics for Larger Classes of Weak Keys

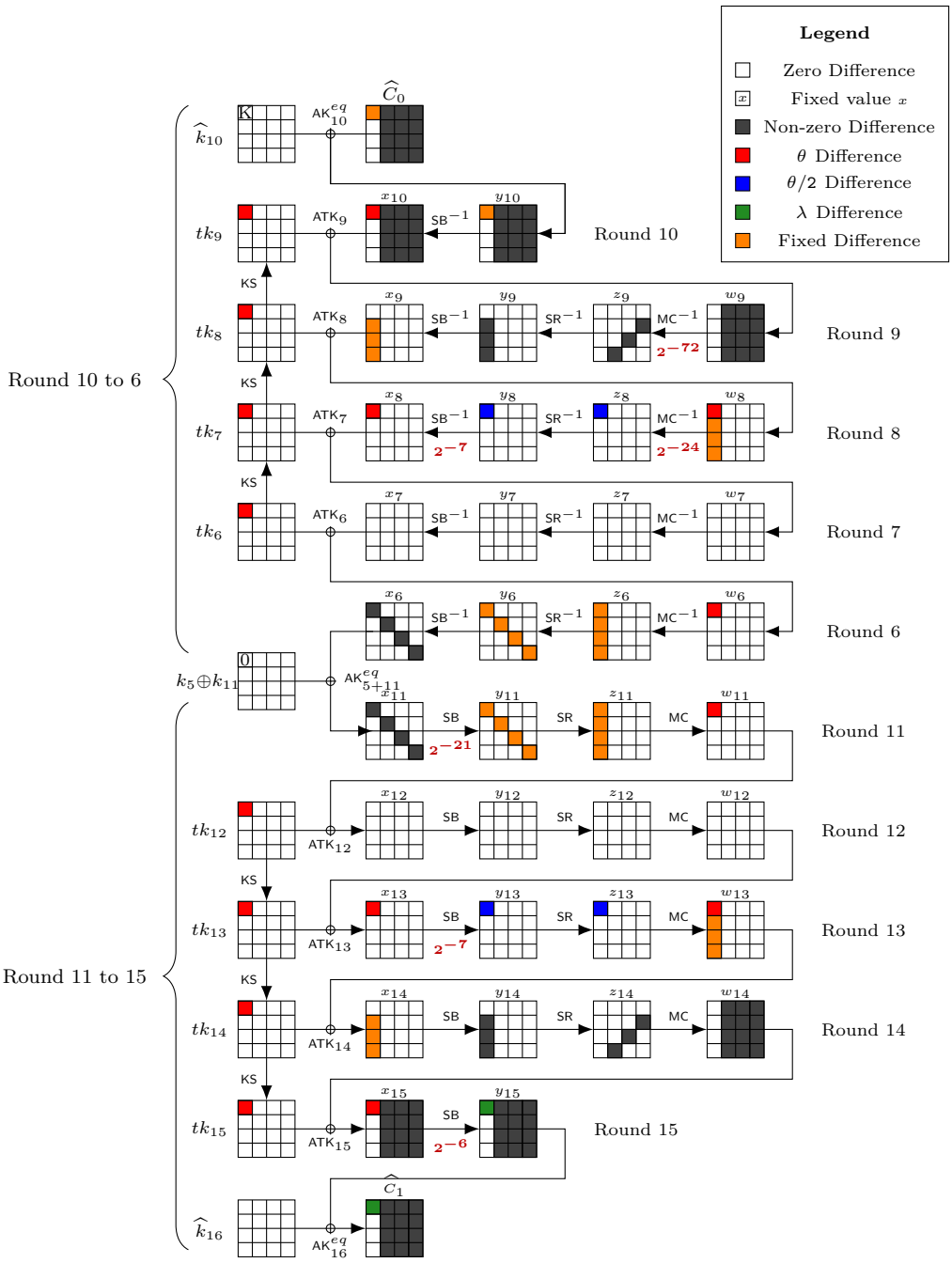


Figure 10: Differential characteristic for 2^{119} weak keys.

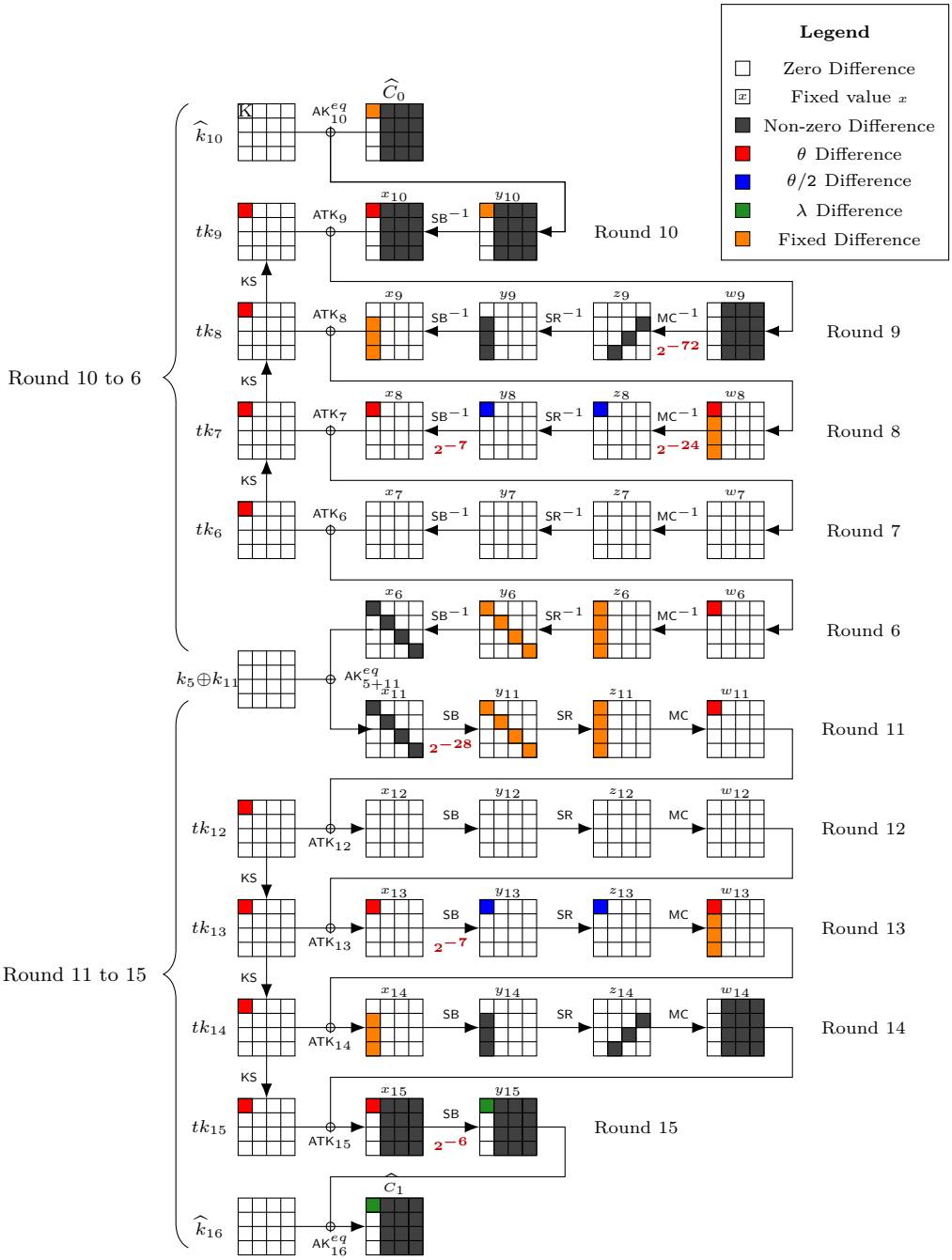


Figure 11: Differential characteristic for 2^{124} weak keys.