Practical Key Recovery Attacks on FlexAEAD

Orr Dunkelman, Maria Eichlseder, Daniel Kales, Nathan Keller, Gaëtan Leurent, Markus Schofnegger

Practical Key Recovery Attacks on FlexAEAD *

Orr Dunkelman[1], Maria Eichlseder[2], Daniel Kales[2],
Nathan Keller[3], Gaëtan Leurent[4], and Markus Schofnegger[2]

[1] Computer Science Dept., University of Haifa, Israel
orrd@cs.haifa.ac.il
[2] Graz University of Technology, Austria
{firstname.lastname}@iaik.tugraz.at
[3] Department of Mathematics, Bar Ilan University, Ramat Gan, Israel
nkeller@math.biu.ac.il
[4] Inria, Paris, France
gaetan.leurent@inria.fr

Abstract. FlexAEAD is a block cipher candidate submitted to the NIST Lightweight Cryptography standardization project, based on repeated application of an Even-Mansour construction. In order to optimize performance, the designers chose a relatively small number of rounds, using properties of the mode and bounds on differential and linear characteristics to substantiate their security claims. Due to a forgery attack with complexity of 2^{46}, FlexAEAD was not selected to the second round of evaluation in the NIST project.
In this paper we present a *practical key recovery* attack on FlexAEAD, using clusters of differentials for the internal permutation and the interplay between different parts of the mode. Our attack, that was fully verified in practice, allows recovering the secret subkeys of FlexAEAD-64 with time complexity of less than 2^{31} encryptions (with experimental success rate of 75%). This is the first practical key recovery attack on a candidate of the NIST standartization project.

Keywords: Authenticated encryption · NIST LWC · practical key recovery · truncated differential

1 Introduction

Background. FlexAEAD [12] is a candidate algorithm in the ongoing NIST Lightweight Cryptography (LWC) standardization project [14]. The FlexAEAD family of authenticated encryption algorithms is based on the previously published authenticated encryption design FlexAE [9,10,11]. Compared to FlexAE, FlexAEAD was modified to also handle associated data blocks, and the generation of the ciphertext blocks was amended by an additional call to the internal keyed permutation to better resist reordering attacks.

* This paper is partially based on [3], presented at the IMACC 2019 workshop. The main results of the paper, presented in Section 4, are new.

One noteworthy property of the FlexAEAD design is its use of a primitive with certain non-ideal properties, in particular the possibility to find differential distinguishers. This primitive, the Even-Mansour keyed permutation PF_K, is essentially used in a triple-encryption construction with different keys K_0, K_1, K_2 to encrypt plaintext blocks into ciphertext blocks, while the intermediate encryption results are accumulated to later derive the tag. For this reason, the designers argue in [12, Sec. 3] that it is sufficient to ensure that no differential or linear distinguishers can be found for the combined number of rounds. They derive corresponding bounds on the maximum probability of differential and linear characteristics, based on the excellent properties of the underlying AES S-box and the (weaker) diffusion properties of the (more lightweight) linear layer.

This approach deviates from the classical separation of mode and primitive, where the security proof of the first relies on well-defined security assumptions regarding the latter. However, related ideas have enjoyed considerable popularity in the CAESAR competition for authenticated encryption schemes, since they allow secure designs with a very lightweight footprint (e.g., [1]) or very high performance (e.g., [17]). Moreover, such ideas can also be found in several other NIST LWC submissions. In these examples, unlike FlexAEAD, the reduced primitives typically serve as state update functions where the state size is considerably larger than the absorbed data block size.

Previous results. Several previous works observed weaknesses of FlexAEAD and used them to mount forgery attacks. In a note posted on NIST's LWC mailing list, Mège [7] presented a trivial padding domain separation attack for associated data. In [16], Rahman et al. presented truncated differential and yoyo-game attacks on the keyed internal permutation of FlexAEAD. Furthermore, they showed that these attacks can be leveraged into a forgery attack on the entire encryption scheme, with complexity of $2^{50}, 2^{60}$, and 2^{80} for FlexAEAD-64, FlexAEAD-128, and FlexAEAD-256 (respectively). In [3] (on which this paper is partially based), Eichlseder et al. presented improved forgery attacks on the cryptosystem, with complexities of $2^{46}, 2^{54}$, and 2^{70} for FlexAEAD-64, FlexAEAD-128, and FlexAEAD-256 (respectively). Due to these forgery attacks, FlexAEAD was not selected for the second round of evaluation in the NIST LWC standartization project [15].

The designers of FlexAEAD have responded to the attacks with a series of suggested tweaks intended to mitigate them. We briefly discuss these tweaks and their effect on our results in Section 6.2.

Our contributions. Key recovery attacks on AEAD algorithms are very rare. Usually, even in lightweight algorithms, the key generation part is strong, and even when weaknesses are found, they lead to forgery attacks and not to key recovery. FlexAEAD seems to be no exception in this regard. The secret subkeys K_0, K_1, K_2, K_3 are generated from the master key via triple-encryption using the internal keyed permutation PF_K, and this part of the algorithm seems very hard to penetrate. Indeed, all previous attacks on FlexAEAD are forgery attacks.

Table 1: Comparison of our attacks on FlexAEAD with previously known attacks

Variant	Attack type	Complexity	Source
FlexAEAD-64	Forgery	2^{50}	[16]
FlexAEAD-64	Forgery	2^{46}	[3]
FlexAEAD-64	Key Recovery	2^{31}	Sec. 4
FlexAEAD-128	Forgery	2^{60}	[16]
FlexAEAD-128	Forgery	2^{54}	[3], Sec. 5
FlexAEAD-128	Key Recovery	2^{59}	Sec. 4
FlexAEAD-256	Forgery	2^{80}	[16]
FlexAEAD-256	Forgery	2^{70}	[3], Sec. 5
FlexAEAD-256	Key Recovery	2^{140}	Sec. 4

In this paper we present a practical key recovery attack on FlexAEAD-64. Like in [3,16], the starting point of our attack is a truncated differential of PF_K. However, the application of PF_K in the mode which we target differs from the application targeted in [3,16]. On the one hand, this makes the construction of the differential more complex, and its probability lower. On the other hand, this allows us not only obtaining a forgery attack, but also recovering part of the secret key. Then, we combine truncated differential attacks on other applications of PF_K with exploitation of weaknesses of the mode to recover the full secret subkey K_0, K_1, K_2, K_3 with overall complexity of less than 2^{31}. We have fully verified the result experimentally, and the attack recovers the full subkey in less than 3 minutes on a 24-core machine, with a success rate of 75%.

Interestingly, the attack does not apply to FlexAE— the predecessor of FlexAEAD, and the best attack we could find on FlexAE-64 has complexity of about 2^{41}. Thus, it appears that the insertion of an additional encryption layer in the transition from FlexAE to FlexAEAD, which was intended to increase the security level, actually *made the cipher more vulnerable*.

Our key recovery attack applies also to FlexAEAD-128 and FlexAEAD-256, but with significantly higher complexities of about 2^{59} and 2^{140}, respectively. For these variants, the forgery attacks initially presented in [3] have significantly lower complexities. A comparison of our attacks with previously known attacks is presented in Table 1.

Outline. We recall the FlexAEAD design in Section 2. The basic truncated differential of PF_K used in our key recovery attack is presented in Section 3. Our key recovery attack on FlexAEAD-64 is presented in Section 4. For the sake of completeness, we describe the forgery attacks on FlexAEAD-128 and FlexAEAD-256 (initially presented in [3]), in Section 5. In Section 6, we discuss the application of our attacks to FlexAE, and the effect of tweaks proposed by the designers of FlexAEAD on our attacks. Finally, we conclude in Section 7.

2 Description of FlexAEAD

In this section, we summarize the construction of FlexAEAD [12].

The main building block of FlexAEAD is a keyed permutation PF_K. This permutation is an Even-Mansour construction with whitening keys K_A, K_B, where the master key is $K = K_A \,||\, K_B$. In FlexAEAD-n, the Even-Mansour construction operates on n-bit blocks, divided into nibbles $\{b_i\}_{i=0,1,\ldots,n/4-1}$. The inner permutation consists of $r = \log_2(n/2)$ rounds, and we denote it as PF. Each round consists of two operations: First, a Block Shuffle layer is applied, in which the nibbles are rearranged in the order $b_0, b_{n/8}, b_1, b_{n/8+1}, \ldots, b_{n/8-1}, b_{n/4-1}$. (This layer is reminiscent of Tree-Structured SPNs [6].) Then, a 3-round Feistel construction whose round function consists of parallel application of $n/16$ copies of the 8-bit AES S-Box [2] is applied. The full construction of PF_K is given in Figure 1. For a more detailed description of the building blocks, we refer to the NIST submission document [12].

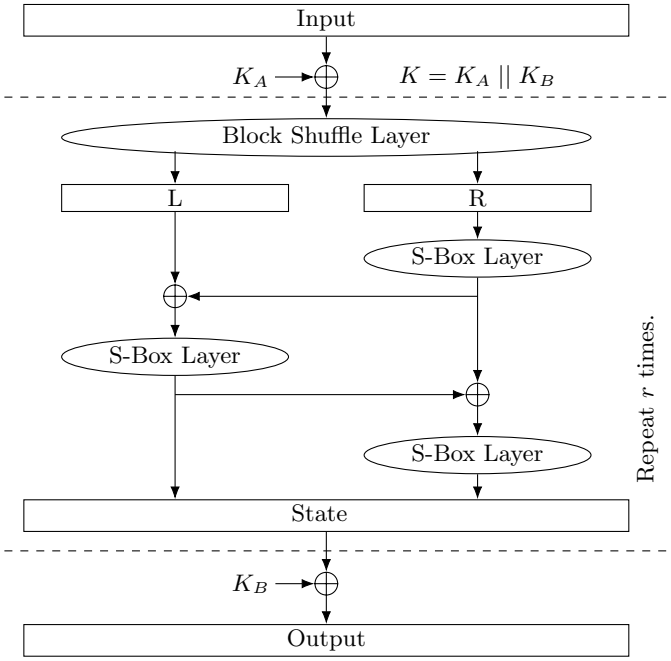
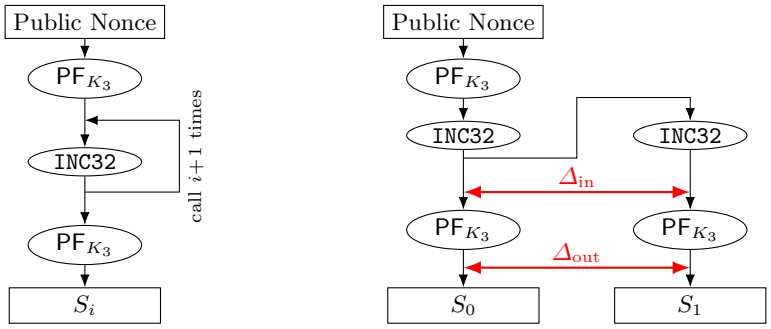


Fig. 1: Keyed permutation PF_K [12] with $r \in \{5, 6, 7\}$ for FlexAEAD-$\{64, 128, 256\}$.

PF_K is used with four different keys in the FlexAEAD construction. These four keys are derived from a master key K^* by applying PF_{K^*} three times to an initial state of 0^n, iterating this process to generate enough bits for the four subkeys K_0, K_1, K_2, K_3. A base counter is generated by applying PF_{K_3} to the nonce. This base counter is then used to generate the sequence $(S_0, \ldots, S_{n+m-1})$

in the following way. First, it is incremented by the step INC32, which treats each 32-bit block as an 32-bit little-endian integer and increases it by one modulo 2^{32}. PF_{K_3} is then again applied to the result of INC32, yielding the block S_0. Further blocks S_i are generated by calling INC32 on the base counter $i+1$ times before finally applying PF_{K_3}, as shown in Figure 2a.



(a) Generation of the sequence S. (b) Differential $\Delta_{\text{in}} \to \Delta_{\text{out}}$.

Fig. 2: The generation process of the sequence S and counter-based differentials in it.

The sequence $S_0, \ldots, S_{n-1}, S_n, \ldots, S_{n+m-1}$ is then used to mask the associated data blocks $A_0, \ldots, A_{n-1}$ and plaintext blocks $P_0, \ldots, P_{m-1}$, as well as intermediate results of the ciphertext generation process. This construction is inspired by the Integrity Aware Parallelizable Mode (IAPM) [4,5]. To compute the tag T, PF_{K_0} is applied to the XOR of the intermediate results after application of PF_{K_2} to each masked block, plus a constant indicating whether the last plaintext block was a full or a partial block. The full construction is illustrated in Figure 3. Note that while the theoretical construction of FlexAEAD allows truncation of the tag (denoted in the figure by 'MSB'), in the NIST submission the tag is not truncated in any of the variants of FlexAEAD.

3 The Truncated Differential of PF_K used in Our Attack

In our attack, we target the first part of the generation of the sequence S, namely, the application of PF_{K_3} to generate the base counter from the publicly known nonce.

The basic observation. We observe that if for two nonces N, N', we have

$$\texttt{INC32}(\mathsf{PF}_{K_3}(N')) = \mathsf{PF}_{K_3}(N), \tag{1}$$

then for any $i \geq 0$, we have $S_{i+1}(N') = S_i(N)$. This can be easily detected by an attacker, by asking for the encryption of the messages (P_0, P_1) and (P'_0, P'_1) under

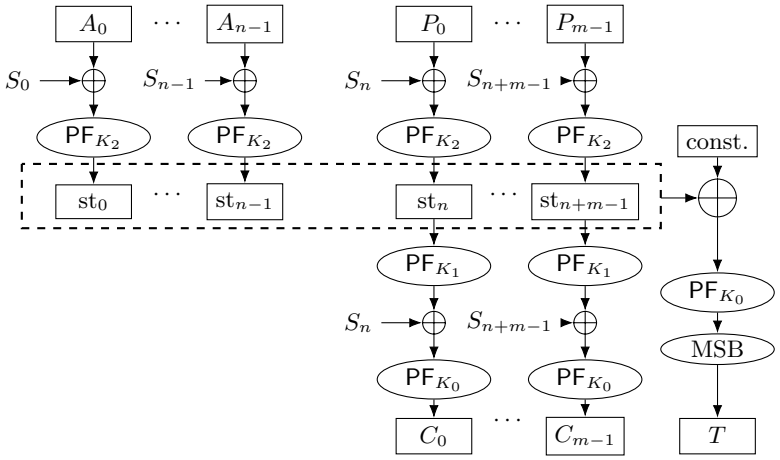


Fig. 3: The FlexAEAD mode for authenticated encryption (simplified, from [12]). K_0, K_1, K_2 are derived from the master key, and the S_i's are derived from the master key and nonce.

the nonces N, N', respectively, where $P'_1 = P_0$. By the structure of FlexAEAD-n, if (1) holds, then the corresponding ciphertexts (C_0, C_1) and (C'_0, C'_1) must satisfy $C'_1 = C_0$. Otherwise, the probability that this equality is satisfied is 2^{-n}.

Note that the relation (1) is actually a convenient output difference of PF_{K_3} (specifically, it consists of an *additive* difference of 1 in each 32-bit word of the output). As the corresponding inputs to PF_{K_3} – namely, the nonces N, N' – can be chosen by the attacker, it follows that any differential of PF_{K_3} whose output difference corresponds to (1) can be used to distinguish FlexAEAD from a random permutation. As we shall see in Section 4, such differentials can be used to mount key recovery attacks on the cryptosystem as well.

Comparison with previous works. We note that the previous works [3,16] target the second part of the generation of the sequence S, namely, the application of PF_{K_3} to generate the sequence $\{S_i\}$ from the base counter. These works are based on the observation that two consecutive values of the counter yield a convenient *input difference* Δ_{in} to PF_{K_3}, while the corresponding output difference is $\Delta_{\text{out}} = S_i \oplus S_{i+1}$, that can be checked by the attacker by observing its influence on the tag (see Figure 2b). As a result, the previous works can use any differential of PF_{K_3} with the convenient input difference Δ_{in}. We have a free choice of the input difference, but are confined to a somewhat less convenient output difference. However, the ability of checking the differential directly via the ciphertext, and not only indirectly via the influence on the tag, gives us extra power that allows mounting not only forgeries but also key recovery attacks.

The truncated differential. As was observed in [3,16], the Feistel part of the round function of PF_K in FlexAEAD-n can be viewed as $n/16$ Super S-boxes of 16-to-16

bits, applied in parallel. The first Super S-box involves nibbles $b_0, b_1, b_{n/8}, b_{n/8+1}$, the second involves nibbles $b_2, b_3, b_{n/8+2}, b_{n/8+3}$, etc.

The Block Shuffle Layer transforms nibbles $b_0, b_{n/8}$ to nibbles b_0, b_1. This gives rise to an iterative truncated differential of PF_K with probability of about 2^{-8} per round: Start with a possibly non-zero difference only in nibbles $b_0, b_{n/8}$. After the Block Shuffle Layer, the difference is possibly non-zero only in nibbles b_0, b_1, and hence, only the first Super S-box is active (i.e., has a non-zero input difference). With probability of about 2^{-8}, the output difference of the Feistel layer is non-zero only in nibbles $b_0, b_{n/8}$, and the differential can be continued iteratively. This iterative differential belongs to the family of differentials presented in [3,16]. Alternatively the output difference of the Feistel layer can be non-zero only in nibbles $b_1, b_{n/8+1}$ with a probability of about 2^{-8}. In this case the next round is active only in the second Super S-Box; at each round there are two possibilities to keep a single Super S-Box active.

Our truncated differentials for FlexAEAD-64 are presented in Figure 4. In particular, Rounds 2,3 coincide with the iterative truncated differential we just described, with the first truncated differential being only active in the first Super S-Box, and the three other differential characteristics moving to different Super S-Boxes while keeping the same input and output differences.

In Round 4, the only active Super S-box is the first one (like above), but we ask for a specific output difference, that corresponds to Equation (1). Specifically, we ask that after the XOR with the second whitening subkey K_B, the *additive* difference will be 01_x in nibbles b_1, b_9 and zero in all other nibbles. This holds with probability of about 2^{-16}, and almost ensures that Equation (1) holds. (The only exceptional cases are where the values in one of the nibbles b_1, b_9 are $00_x, FF_x$; their total probability is about $1/128$, and so we can neglect them.)

In Rounds 0,1 we could use the truncated differential described above, but we prefer to activate more Super S-boxes, in order to reduce the data complexity by using *structures of nonces*. Specifically, we start with an input difference that is possibly non-zero in all even nibbles. After the first Block Shuffle Layer, the active nibbles are $b_0, b_1, b_4, b_5, b_8, b_9, b_{12}, b_{13}$, and so, only two Super S-boxes are active. With probability of about 2^{-16}, the output difference is (possibly) non-zero only in nibbles b_0, b_4, b_8, b_{12}. After the Block Shuffle Layer of Round 1, the active nibbles are only b_0, b_1, b_8, b_9, and so, in Round 1 only the first Super S-box is active. With probability of about 2^{-8}, the output difference is (possibly) non-zero only in nibbles b_0, b_8, which is the input difference of the iterative truncated differential we use in Rounds 2,3.

As seen in Figure 4, there are four similar truncated differentials with probability 2^{-56}, with the same input and output differences, activating different Super S-Boxes at each round. There are also two alternative truncated characteristics with the same input/output differences and probability 2^{-56} given in Figure 5. These additional characteristics have two active Super S-Boxes rather than one in Round 1, and we do not use them in our key-recovery attack because this makes recovering the key harder.

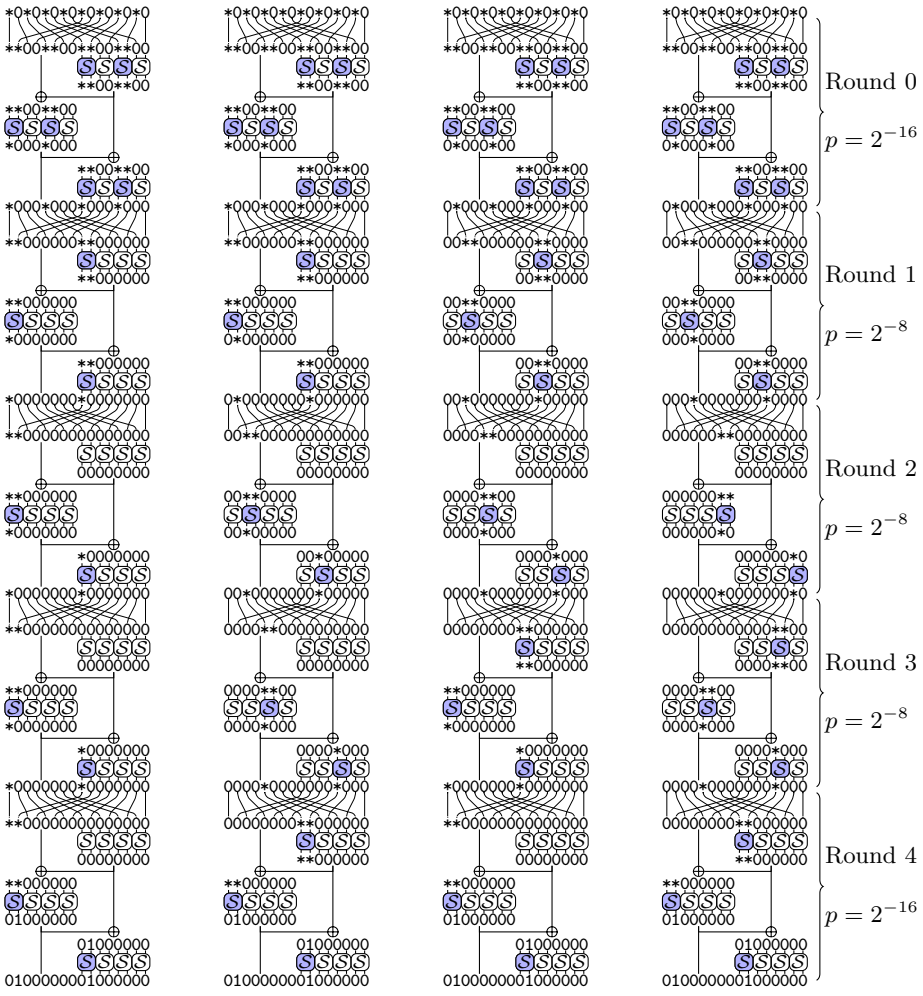


Fig. 4: The truncated differentials of PF_K used in our key recovery attack. An asterisk denotes an unknown value.

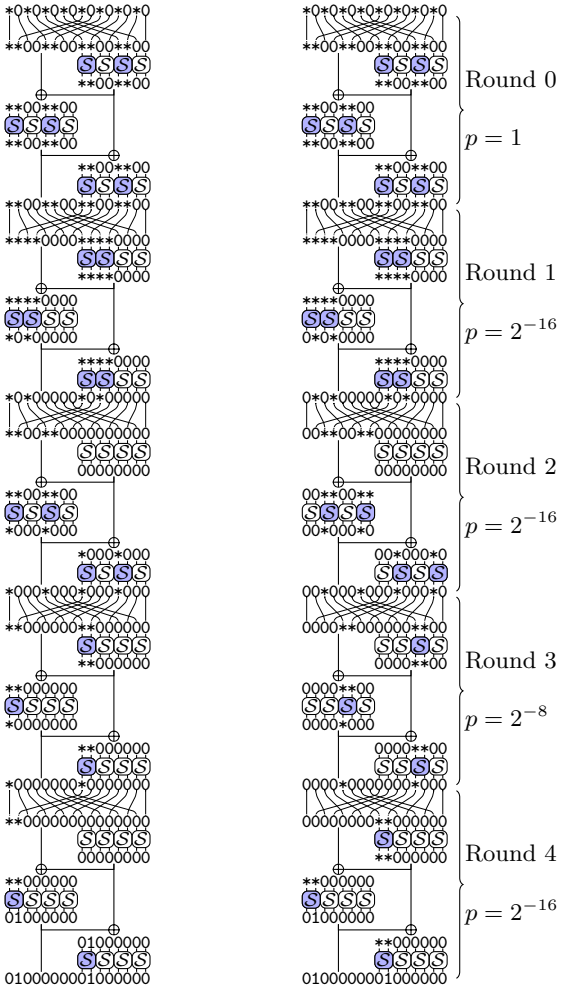


Fig. 5: Alternative truncated differential of PF_K, not used in our key recovery attack. Pairs following those differentials are considered 'wrong pairs'.

The probability of the differential. The total probability that a pair follows one the four differentials we use is $4 \cdot 2^{-16} \cdot (2^{-8})^3 \cdot 2^{-16} = 2^{-54}$, which is significantly lower than the probability of the differentials presented in [3,16]. However, as we shall see below, due to the usage of structures of nonces, the complexity of an attack based on this differential is much lower than the complexities of the attacks described in [3,16].

4 Key Recovery Attacks on FlexAEAD

In this section we present key recovery attacks on FlexAEAD. We describe the practical attack on FlexAEAD-64 and analyze its complexity in Section 4.1. In Section 4.2 we describe the practical verification of the attack, and in Section 4.3 we briefly describe key-recovery attacks on FlexAEAD-128 and FlexAEAD-256.

4.1 The key recovery attack on FlexAEAD-64

Our key recovery attack consists of six phases:

1. Recovering the first half of K_3, using a truncated differential attack on the generation of the basic counter.
2. Recovering the second half of K_3, using a truncated differential attack on the generation of the sequence $\{S_i\}$ from the basic counter.
3. Recovering the second half of K_0, using a truncated differential attack on the last application of PF_K in the ciphertext generation.
4. Recovering the first half of K_0, using a property of the mode.
5. Recovering K_2, using the previously obtained subkeys and a truncated differential attack on PF_{K_2}.
6. Recovering K_1, using the previously obtained subkeys and a truncated differential attack on PF_{K_1}.

The six phases are presented below, along with their complexity analysis.

Phase 1: Recovering K_A^3, the first half of K_3. This phase is divided into several steps. We describe them in detail, as most of the following phases use similar steps.

Step 1: Finding several pairs of nonces that satisfy the truncated differential. We consider a structure of 2^{28} nonces, all whose odd-numbered nibbles (i.e., nibbles $b_1, b_3, \ldots, b_{15}$) are equal to zero. We ask for the encryption of the two-block message $(P_0, P_1) = (0, 0)$ under each of the nonces. We insert the corresponding ciphertexts (C_0, C_1) into a hash table, and find in the table all pairs $(C_0, C_1), (C_0', C_1')$, such that $C_0 = C_1'$.

We claim that about 6 such pairs are expected, and about 4 of them (called 'right pairs') satisfy the used truncated differentials (2 satisfy the alternative truncated differentials). Indeed, note that any two of the nonces in the structure satisfy the input difference of our truncated differential. Each truncated

differential is satisfied with probability of 2^{-56}, and hence, one pair that satisfy each differential are expected. (Note that the total number of pairs is close to $(2^{28})^2 = 2^{56}$, since we consider an additive output difference after PF_{K_3} and a match $C_0 = C'_1$, and so, the pairs we look at are ordered.) As was shown in Section 3, for each such pair (N, N'), if the plaintexts encrypted under these nonces satisfy $P_0 = P'_1$, then the corresponding ciphertexts must satisfy $C_0 = C'_1$. In our case, all pairs of plaintexts satisfy this relation, and thus, the pair will be detected. On the other hand, the probability that a 'random' pair satisfies $C_0 = C'_1$ is 2^{-64}, and so, with probability of $1 - 2^{56}/2^{64} \approx 1$, no 'random' pairs satisfy our condition.

Analysis. The data complexity of this step is 2^{28} two-block messages, or 2^{29} blocks in total. The memory complexity is 2^{29}, and the time complexity is dominated by encryption of the plaintexts (due to the usage of a hash table). The expected number of right pairs is 4, and hence, by a standard Poisson approximation, the probability that at least one pair remains is 98%.

Step 2: Recovering K_A^3 using a right pair. Consider a pair of nonces (N, N') that satisfies the truncated differential, and examine the differential transition in Round 0. In this round, two Super S-boxes are active, and in each of them, we have an 8-bit condition on the output difference. As each of these Super S-boxes depends on different bits of K_A^3 (i.e., the first half of K_3), we consider them separately.

The first Super S-box depends on nibbles b_0, b_4, b_8, b_{12} of K_A^3. We can guess the values of these nibbles, and check whether the output difference of Round 0 is zero either in nibbles b_1, b_9 or in nibbles b_0, b_8 (depending on the truncated differential followed). As this is an 7-bit condition, the expected number of key guesses that pass this filtering is $2^{16} \cdot 2^{-7} = 2^9$.

The other active Super S-box depends on nibbles b_2, b_6, b_{10}, b_{14} of K_A^3. Similarly to the first Super S-box, we can guess these nibbles and check whether the output difference of Round 0 is zero either in nibbles b_5, b_{13} or in nibbles b_4, b_{12}. Again, we expect 2^9 keys to pass this filtering. However, the choice of the active nibbles must match with the choice made in the first Super S-box: we have either b_1, b_5, b_9, b_{13}, or b_0, b_4, b_8, b_{12}. Therefore, we expect 2^{17} candidates for the even nibbles of K_A^3.

For each of those candidates, we can compute the active Super S-box of Round 1. If nibbles b_0, b_4, b_8, b_{12} (respectively b_1, b_5, b_9, b_{13}) are active at the output of Round 0, the Block Shuffle Layer moves them to nibbles b_0, b_1, b_8, b_9 (respectively b_2, b_3, b_{10}, b_{11}) corresponding to the first (respectively second) Super S-Box of Round 1. Following the truncated differentials of Figure 4, we expect that only two nibbles are active after the Super S-Box: either b_0, b_8 or b_1, b_9 (respectively either b_2, b_{10} or b_3, b_{11}). As this is a 7-bit condition, we expect $2^{17} \cdot 2^{-7} = 2^{10}$ candidates remaining for the the even nibbles of K_A^3.

Then, we consider Round 2. Given the even nibbles of K_A^3 we can compute $b_0, b_1, b_2, b_3, b_8, b_9, b_{10}, b_{11}$ at the beginning of Round 2, and we know that only two nibbles are active: either (b_0, b_8), (b_1, b_9), (b_2, b_{10}), or (b_3, b_{11}) (corresponding to the four truncated differentials). Without loss of generality, we assume that

b_0, b_8 is active, corresponding to the first truncated differential (the other cases are similar).

We now guess the values of nibbles b_4, b_6, b_{12}, b_{14}. Combined with known nibbles b_0, b_2, b_8, b_{10}, we have all the even nibbles at the beginning of Round 2, and we can verify that the truncated differential is followed at Round 2 and 3, leading to a 16-bit filter. Therefore we expect 2^{10} candidates remaining.

Finally, we guess the values of nibbles b_5, b_7, b_{13}, b_{15} at the beginning of Round 2. This provides the full internal state: we check the last round difference to obtain a 16-filter, and we compute backwards to get 2^{10} candidates for K_A^3.

Analysis. The last guess requires $2 \cdot 2^{10} \cdot 2^{16}$ 3-round encryptions (repeated for 1.5 candidate pairs on average), which is negligible compared to Step 1. With high probability, the correct value of K_A^3 is among the 2^{10} candidates, and we can filter the right value using a few additional queries. Indeed, when K_A^3 is known we can easily generate pairs of nonces leading to $C_0 = C'_1$.

Phase 2: Recovering K_B^3, the second half of K_3. In this phase we target the second half of the generation of the sequence $\{S_i\}$, and more specifically, the generation of S_0 from the basic counter.

Recall that $S_0 = \mathsf{PF}_{K_3}(\texttt{INC32}(\mathsf{PF}_{K_3}(N))$, where N is the public nonce. Consider encryption processes under the same secret key and two nonces N, N'. Observe that since the first half of K_3 (denoted by K_A^3) is known, the difference $\mathsf{PF}_{K_3}(N) \oplus \mathsf{PF}_{K_3}(N')$ is known as well. The operation INC32 affects this difference in a way we cannot fully predict, but as we shall see below, its effect can be neglected in our case. Hence, we essentially know (and even can 'almost' choose) the input difference to the second function PF_{K_3}. On the other hand, we can indirectly check its output difference. Indeed, this output difference is equal to $S_0 \oplus S'_0$ (where S'_0 is the value that corresponds to S_0 in the encryption with N'). For each Δ, we can check whether $S_0 \oplus S'_0 = \Delta$ by encrypting single-block messages P_0, P'_0 such that $P_0 \oplus P'_0 = \Delta$ under N, N' (respectively), and checking whether the tags collide.

This allows us attacking the second application of PF_{K_3}, using essentially the same basic truncated differential described above. (Note that here we have more freedom, as we can choose any output difference we wish. However, we did not find a way to use this to significantly reduce the complexity of this phase, which is anyway lower than the complexity of Phase 1.) Our attack goes as follows:

1. Take $2^{28.5}$ distinct values X_i that are equal to zero in all odd nibbles.
2. Find the $2^{28.5}$ nonces N_i such that $X_i = \mathsf{PF}_{K_3}(N_i) \oplus K_B^3$. (Note that this can indeed be done, as the right hand side depends only on K_A^3 which we already recovered; the XOR with K_B^3 undoes the final key whitening of PF_{K_3}.)
3. For each nonce N_i, ask for the encryption of a single-block message P_0^i under N_i, where all P_0^i's are equal to zero in all bytes except for bytes 0 and 4, and are distributed evenly in bytes 0 and 4. Insert all resulting tags into a hash table and find all pairs (N_i, N_j) whose corresponding tags collide.

4. For each of the colliding pairs, assume that the corresponding pair (X_i, X_j) is a right pair with respect to the truncated differential described above and use it to recover the internal state of PF_K in the way described in Step 2 of Phase 1.
5. Each value of the internal state suggests a candidate for K_B^3, filter the roughly 2^{10} candidates using a few extra queries.

Why does the algorithm work? For each i, j, the difference $X_i \oplus X_j$ is zero in all odd nibbles. The operation `INC32` can affect this property only if the least significant byte in one of the 32-bit words in either X_i or X_j is FF_x. This happens with probability of at most 4/256, and thus, can be neglected (with a small effect on the overall success probability). Thus, we may assume that each pair (X_i, X_j) satisfies the input difference of the differential described above.

As was shown above, with probability 6×2^{-40}, the corresponding difference after Round 3 is non-zero only in two nibbles (either $(0, 8)$ or $(4, 12)$), and thus, only the first Super S-box is active in Round 4. For each non-zero difference α, β in bytes 0 and 8, the probability that the output difference of Round 4 (in bytes) is $(\alpha, 0, 0, 0, \beta, 0, 0, 0)$, is about 2^{-16}. Hence, for each pair (i, j) such that $P_0^i \neq P_0^j$ (and almost all the pairs satisfy this condition by our choice of the P_0^i's), we have

$$\Pr[S_0^i \oplus S_i^j = P_0^i \oplus P_j^i] \approx 6 \times 2^{-56}.$$

By the structure of FlexAEAD-64, for each such pair (i, j), we obtain a collision in the tag. As the structure contains approximately $(2^{28.5})^2/2 = 2^{56}$ unordered pairs (P_i, P_j), about 6 collisions that originate from the differentials are expected: 4 'right pairs' corresponding to one of the differentials of Figure 4, and 2 'wrong pairs' corresponding to one of the differentials of Figure 5. On the other hand, the probability of a random collision in the tag is 2^{-64}, and so no random collisions are expected.

Given a right pair, K_B^3 can be found exactly in the same way as in Step 2 of Phase 1 (that is, by examining the first round of the differential and using the fact that K_A^3 is known). The 2^{10} candidates can be filtered as follows: choose two arbitrary nonces N and N', compute the corresponding S_0 and S_0' using $K_3 = K_A^3 \,||\, K_B^3$, and encrypt the message S_0 under nonce N and S_0' under nonce N'. If the K_3 candidate is correct, the two tags collide.

Analysis. The data complexity of this phase is $2^{28.5}$ single-block messages, the memory complexity is $2^{28.5}$, and the time complexity is dominated by encrypting the plaintexts. At the end of this phase, the entire subkey K_3 is known with a very high probability.

Phase 3: Recovering K_B^0, the second half of K_0. At this phase, we target the last application of PF_{K_0} in the ciphertext generation. Note that as we already know K_3, we can compute the sequence $\{S_i\}$. In the encryption process of the message block P_i, the value S_{n+i} is used twice: first it is XORed to P_i as an initial whitening, and then it is XORed again to the intermediate value before the last application of PF_{K_0}.

This allows us to mount a differential attack on PF_{K_0}. First, we choose a convenient input difference Δ_{in}. Then, we choose two values S_0, S'_0 such that $S_0 \oplus S'_0 = \Delta_{in}$. We then take two nonces N, N' that lead to S_0, S'_0, and encrypt under them single-block plaintexts P_0, P'_0 with difference Δ_{in}, so that the initial XOR with S_0, S'_0 cancels the plaintext difference. The equality between intermediate states is preserved until the second XOR with S_0, S'_0, which generates input difference of Δ_{in} to PF_{K_0} (see Figure 8). As the corresponding outputs of PF_{K_0} are the ciphertexts C_0, C'_0, this allows mounting a truncated differential attack on PF_{K_0}, as claimed above.

In the attack, we use variants of the basic truncated differential used in Phases 1,2. Note that our situation here differs from the situation in the previous phases in two aspects. On the one hand, we can observe the ciphertext difference directly, while in Phase 2 we could only view the effect of the output difference on the tag and in Phase 1 we had to hit a specific output difference. This allows us to easily detect right pairs, and also reduces the complexity of this step. On the other hand, while in Phases 1,2 we know the actual input values to PF_K, here we know the output values (which are the ciphertexts) and the input difference.

To find the right pairs efficiently, we take the same input difference like in Phases 1,2, and observe that we can leave out the last round of the differential (which affects only the output difference in nibbles b_0, b_1, b_8, b_9 of the ciphertext) and check that the ciphertext difference is zero in the remaining 12 nibbles. This increases the probability of the differential to 2^{-40}, and thus, reduces the data complexity significantly.

In the subkey recovery phase, we use the same strategy as previously. Step 2 of Phase 1 recovers candidates for the internal state of PF_K using only the known input difference and output difference, assuming that the first rounds follow one of the truncated differentials of Figure 4. After recovering the internal state of PF_K, we will deduce candidates for the *second* half of K_0 (namely, K_B^0) using the known ciphertext, while the previous phases generated candidates for the first half of the key using the known plaintext.

The attack algorithm is the following:

1. Take $2^{20.5}$ distinct values $\{S_0^i\}_i$ that are equal to zero in all odd nibbles.
2. Find the $2^{20.5}$ nonces N_i such that $S_0^i = \mathsf{PF}_{K_3}(\texttt{INC32}(\mathsf{PF}_{K_3}(N_i)))$. (Note that this can indeed be done, as K_3 is known.)
3. For each nonce N_i, ask for the encryption of a single-block message $P_0^i = S_0^i$ under N_i. Insert all resulting ciphertexts C_0^i into a hash table and find all pairs (N_i, N_j) whose corresponding ciphertexts collide in all nibbles except for b_0, b_1, b_8, b_9.
4. For each of the colliding pairs, assume that the corresponding ciphertext pair (C_0^i, C_0^j) is a right pair with respect to the truncated differential described above and use it to recover the internal state of PF_K using the algorithm of Step 2 of Phase 1.
5. Each value of the internal state suggests a candidate for K_B^0; filter the 2^{10} candidates using non-colliding pairs, by partially decrypting though PF_{K_0} and checking that the input difference matches the difference between the S_0 values.

Analysis. By the choice of the plaintexts, for each i, j, the difference $S_0^i \oplus S_0^j$ cancels the plaintext difference, and hence, the input difference to PF_{K_0} is $S_0^i \oplus S_0^j$, which satisfies the input difference of the basic truncated differential. Hence, with probability 6×2^{-40}, the ciphertext difference $C_0^i \oplus C_0^j$ is zero in all nibbles except for b_0, b_1, b_8, b_9. Since the data set contains $(2^{20.5})^2/2 = 2^{40}$ unordered pairs, 6 pairs are expected: 4 'right pairs' following the differentials of Figure 4 and 2 'wrong pairs' following the differentials of Figure 4. The data complexity of this part is $2^{20.5}$ single-block messages, the memory complexity is $2^{20.5}$, but the time complexity is about 2^{28} 3-round encryptions following the previous analysis. With very high probability, we have at least one 'right pair' and K_B^0 is recovered successfully.

Phase 4: Recovering K_A^0, the first half of K_0. Consider the encryption of an *empty message* by FlexAEAD-64. By the structure of the scheme, the tag that corresponds to an empty message is $T = \mathsf{PF}_{K_0}(1010 \dots 10_x)$ (note that there is no truncation in FlexAEAD-64). Since we already know K_B^0, we can ask for the tag T of an empty message, and partially decrypt it through $\mathsf{PF}_{K_0}^{-1}$, until the initial key whitening that uses K_A^0. This gives us the value $K_A^0 \oplus 1010 \dots 10_x$, and thus, the value K_A^0 as well.

Analysis. This step requires the encryption of a single message, and its complexity is negligible.

Phase 5: Recovering K_2. Observe that in the encryption of a single-block message P_0, the tag is

$$T = \mathsf{PF}_{K_0}(1010 \dots 10_x \oplus \mathsf{PF}_{K_2}(P_0 \oplus S_0)).$$

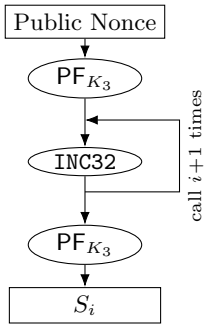
As we already know K_0, we can apply $\mathsf{PF}_{K_0}^{-1}$ to both sides of the equation and obtain the value $\mathsf{PF}_{K_2}(P_0 \oplus S_0)$, which is the output of PF_{K_2}. Regarding the corresponding input, $P_0 \oplus S_0$, we not only know it, but can even choose it, by choosing appropriate values of P_0 and the nonce N. This allows us to mount a truncated differential attack on PF_{K_2}, in which we know the actual input values, can choose the input difference, and can observe the output values directly.

In particular, we can use the same attack applied in Phase 3, to recover K_B^2 with data, memory, and time complexity of $2^{20.5}$. Once K_B^2 is recovered, we can partially decrypt the value $\mathsf{PF}_{K_2}(P_0 \oplus S_0)$ through $\mathsf{PF}_{K_2}^{-1}$ until the initial key whitening that uses K_A^2, and to retrieve K_A^2, as $P_0 \oplus S_0$ is known.

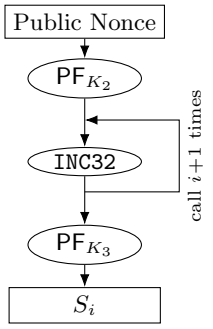
Analysis. This step has data, memory, and time complexity of $2^{20.5}$, and recovers K_2 with an overwhelming probability.

Phase 6: Recovering K_1. In the encryption process of a single-block message P_0, we have

$$\mathsf{PF}_{K_1}(\mathsf{PF}_{K_2}(P_0 \oplus S_0)) = \mathsf{PF}_{K_0}^{-1}(C_0) \oplus S_0,$$



(a) FlexAEAD specification.



(b) FlexAEAD reference implementation.

Fig. 6: The generation process of the sequence S.

where C_0 is the corresponding ciphertext. Since we already know K_0 and K_3, we can compute the right hand side, which is the output of PF_{K_1}. On the other hand, as we know K_2 and K_3, we can choose the input to PF_{K_1}, by an appropriate choice of P_0 and the nonce. This allows us mounting a truncated differential attack on PF_{K_1}, just like in Phase 5.

Analysis. As this phase is essentially identical to Phase 5, it has data, memory, and time complexity of $2^{20.5}$, and recovers K_1 with an overwhelming probability.

Summary of the attack. The overall complexity of the attack is dominated by Phases 1,2 whose complexities are 2^{29} and $2^{28.5}$, respectively. Hence, the attack recovers the full subkeys K_0, K_1, K_2, K_3 with complexity of less than 2^{30}.

4.2 Practical verification

We have verified the attack with the reference implementation of FlexAEAD [12]. However, we discovered that this implementation does not match the specification of FlexAEAD, and differs in the generation of the base counter. As shown in Figure 6 it is computed by applying PF_{K_2} to the nonce, rather than PF_{K_3}. In particular, the first element in the S sequence is computed as $S_0 = \mathsf{PF}_{K_3}(\texttt{INC32}(\mathsf{PF}_{K_2}(N))$ instead of $S_0 = \mathsf{PF}_{K_3}(\texttt{INC32}(\mathsf{PF}_{K_3}(N))$.

We adapt our attack to this variant of FlexAEAD as follows:

1. Phase 1 is unchanged but recovers K_A^2 rather than K_A^3.
2. Phase 2 uses the known value of K_A^2, and recovers the internal state of PF_{K_3} in $\mathsf{PF}_{K_3}(\texttt{INC32}(\mathsf{PF}_{K_2}(N))$. We can deduce the value of $K_B^2 \oplus K_A^3$, using a few extra queries to compensate the impact of the modular addition in INC32.
3. We modify Phase 3 to start from a set of values X_i equal to zero in all odd nibbles, and we compute the nonces as

$$N_i = \mathsf{PF}^{-1}\left(\mathsf{PF}^{-1}(X_i) \oplus K_B^2 \oplus \texttt{0x0100000001000000} \oplus K_A^3\right) \oplus K_A^2.$$

With probability 1/4, the INC32 function behaves like an XOR, and we have $S_0^i = \mathsf{PF}_{K_3}(\texttt{INC32}(\mathsf{PF}_{K_2}(N_i)) = X_i \oplus K_B^3$. We encrypt single block messages $P_0^i = N_0^i$ with nonces N_0^i and we obtain the properties required for Phase 3. Even though we do not know the actual values of S_0^i, they are constant in all odd nibbles when the INC32 function behaves linearly.
We have to increase the amount of data by a factor 4 to compensate the probability 1/4, and we recover K_B^0 with data complexity $2^{22.5}$.

4. Phase 4 is unchanged and recovers K_A^0.
5. Phase 5 has to be adjusted in the same way as Phase 3, because we do not control the value of S_0, but we control differences in S_0, assuming that there is no carry in INC32.
We recover the internal state of PF_{K_2} with data complexity $2^{22.5}$, and we deduce K_B^2. Since we already know $K_B^2 \oplus K_A^3$ from Phase 2, we recover K_A^3, and we can now compute exactly the value $S_0 \oplus K_B^3$ using K_A^2, K_B^2, and K_A^3. We can also decrypt the value $\mathsf{PF}_{K_2}(P_0 \oplus S_0)$ suing K_2, and recover K_B^3.

We have implemented the attack as described above, with all the steps of the attack to recover the full key K_0, K_1, K_2, K_3. We performed 100 simulations of the attack, with a new random key for each experiment. We found that the strategy is successful, but the success rate is somewhat lower than expected. In particular, the Poisson approximation for the number of right pairs in Phase 1 does not hold, probably because the pairs are not independent. Using the amount of data given above (2^{29} for Phase 1, $2^{28.5}$ for Phase 2, and $2^{22.5}$ for Phases 3 and 5), Phase 1 has on average 4.1 collisions with a standard deviation of 3.6. According to the analysis above, we should detect on average 6 pairs (4 right pairs and 2 wrong pairs), with a standard deviation of $\sqrt{6} \approx 2.5$. Our experiments show fewer pairs than expected, and more variability, resulting in a smaller success rate. In particular, we had 17 experiments with 0 collisions, rather than 0.2% expected with a Poisson distribution with $\lambda = 6$, and 32 failures at Phase 1, rather than 2% expected with a Poisson distribution with $\lambda = 4$. The correct key was recovered 56 times, and 44 experiments had insufficient data to recover the key (with most failures at Phase 1). The attack usually runs in less than 1.5 minutes on a 24-core (48-thread) machine (with an Intel Xeon Gold 5118 CPU), but there are a few instances when some pairs take longer to examine and the attack takes up to 3 minutes. We note that about half of the time is spent encrypting the queries (inside the reference code of FlexAEAD).

When doubling the amount of data, the attack takes 3 minutes and has a success rate of roughly 75%; Phase 1 has on average 13.8 collisions with a standard deviation of 9.5 (out of 100 trials). The correct key was recovered 75 times out of 100, and 25 experiments had insufficient data to recover the key (in 22 cases the recovery failed at Phase 1).

4.3 Key recovery attacks on FlexAEAD-128 and FlexAEAD-256

The attack on FlexAEAD-64 described above can be applied also to FlexAEAD-128 and FlexAEAD-256, at the expense of a significant increase in the complexity.

Specifically, it is easy to see that all phases of the attack except Phase 1 apply essentially without change to FlexAEAD-128 and FlexAEAD-256. The complexity increase comes from Phase 1, which dominates the complexity of other phases.

Recall that in Phase 1 of the attack, we target the application of PF_{K_3} that generates the base counter from the nonce. We consider truncated differentials of this function, whose output difference is cancelled by the $INC32$ operation, and thus, is non-zero only in nibbles b_1, b_9.

In the case of FlexAEAD-128, an output difference of PF_{K_3} can be cancelled by the $INC32$ operation only if it non-zero in nibbles b_1, b_9, b_{17}, b_{25}, and thus, we must choose a truncated differential that activates two Super S-boxes in the last round (instead of a single Super S-box in the case of FlexAEAD-64). Moreover, the probability of the last round is reduced to 2^{-32}, since we ask for a specific output difference in two Super S-boxes. The best differential we could find under these restrictions has overall complexity of 2^{-88} (composed of $2^{-16}, 2^{-8}, 2^0, 2^{-16}, 2^{-16}, 2^{-32}$ in Rounds 0,1,2,3,4,5, respectively). The differential shares Rounds 0,1,2 with the basic differential described above, and activates bytes $(0, 1)$, $(0, 2)$, and $(0, 4)$ in Rounds 3,4,5, respectively. As in the FlexAEAD-64 attack, there are four possible truncated differentials with probability 2^{-88}, but we do not expect any wrong pair from truncated differentials similar to Figure 5. Therefore, using 2^{24} structures of 2^{32} nonces we expect to have at least one pair following the differential with complexity 2^{57}. As explained below, we have to repeat this step 4 times, so that the total complexity of the attack is 2^{59}.

In the case of FlexAEAD-256, an output difference of PF_{K_3} can be cancelled by the $INC32$ operation only if it non-zero in nibbles $b_1, b_9, b_{17}, b_{25}, b_{33}, b_{41}, b_{49}, b_{57}$, and thus, we must choose a truncated differential that activates four Super S-boxes in the last round (instead of a single Super S-box in the case of FlexAEAD-64). Moreover, the probability of the last round is reduced to 2^{-64}, since we ask for a specific output difference in four Super S-boxes. The best differential we could find under these restrictions has overall complexity of 2^{-168} (composed of $2^{-16}, 2^{-8}, 2^0, 2^{-16}, 2^{-32}, 2^{-32}, 2^{-64}$ in Rounds 0,1,2,3,4,5,6, respectively). The differential shares Rounds 0,1,2 with the basic differential described above, and activates bytes $(0, 1)$, $(0, 1, 2, 3)$, $(0, 2, 4, 6)$, and $(0, 4, 8, 12)$ in Rounds 3,4,5,6, respectively. By using 2^{104} structures of 2^{32} nonces we obtain an overall complexity of $8 \times 2^{137} = 2^{140}$ for Phase 1, and consequently, for the entire attack.

Finally, we need to tweak Step 2 of Phase 1. This procedure recovers candidates for the internal state of PF (or equivalently, the key) using the known input and output differences, with two parts. First it recovers 2^{10} candidates (out of 2^{32}) for the even nibbles of the input state using conditions in the first two rounds. Then it guesses values to build a full internal state and verifies the remaining rounds to filter the guesses. In FlexAEAD-128 and FlexAEAD-256, guessing a full state is too expensive, therefore we only use the first part, recovering 2^{10} candidates out of 2^{32} for a partial key ($b_0, b_4, b_8, b_{12}, b_{16}, b_{20}, b_{24}, b_{28}$ in FlexAEAD-128, $b_0, b_8, b_{16}, b_{24}, b_{32}, b_{40}, b_{48}, b_{56}$ in FlexAEAD-256). Then we collect another data set using an alternative truncated differential whose first round affects other nibbles of the key, and recover 2^{10} candidates for the corresponding partial key.

In FlexAEAD-128, we require 4 repetitions and obtain 2^{40} candidates for the full key. In FlexAEAD-256, we require 8 repetitions and obtain 2^{80} candidates for the full key. We can then filter those candidates using additional queries as in the FlexAEAD-64 attack. The complexity is dominated by the filtering of key candidates, with complexity 2^{40} for FlexAEAD-128 and 2^{80} for FlexAEAD-256. The complexity of this procedure can be reduced using several right pairs to extract more information of each partial key, but since it is not the bottleneck of our attack we omit this improvement.

In the following section we present forgery attacks with a much lower complexity on FlexAEAD-128 and FlexAEAD-256.

5 Forgery Attacks on FlexAEAD

In this section, we first propose several additional truncated differential characteristics for PF_{K_3}. Then, we show how to apply the resulting counter differential to obtain several forgery attacks on FlexAEAD-64, FlexAEAD-128, and FlexAEAD-256.

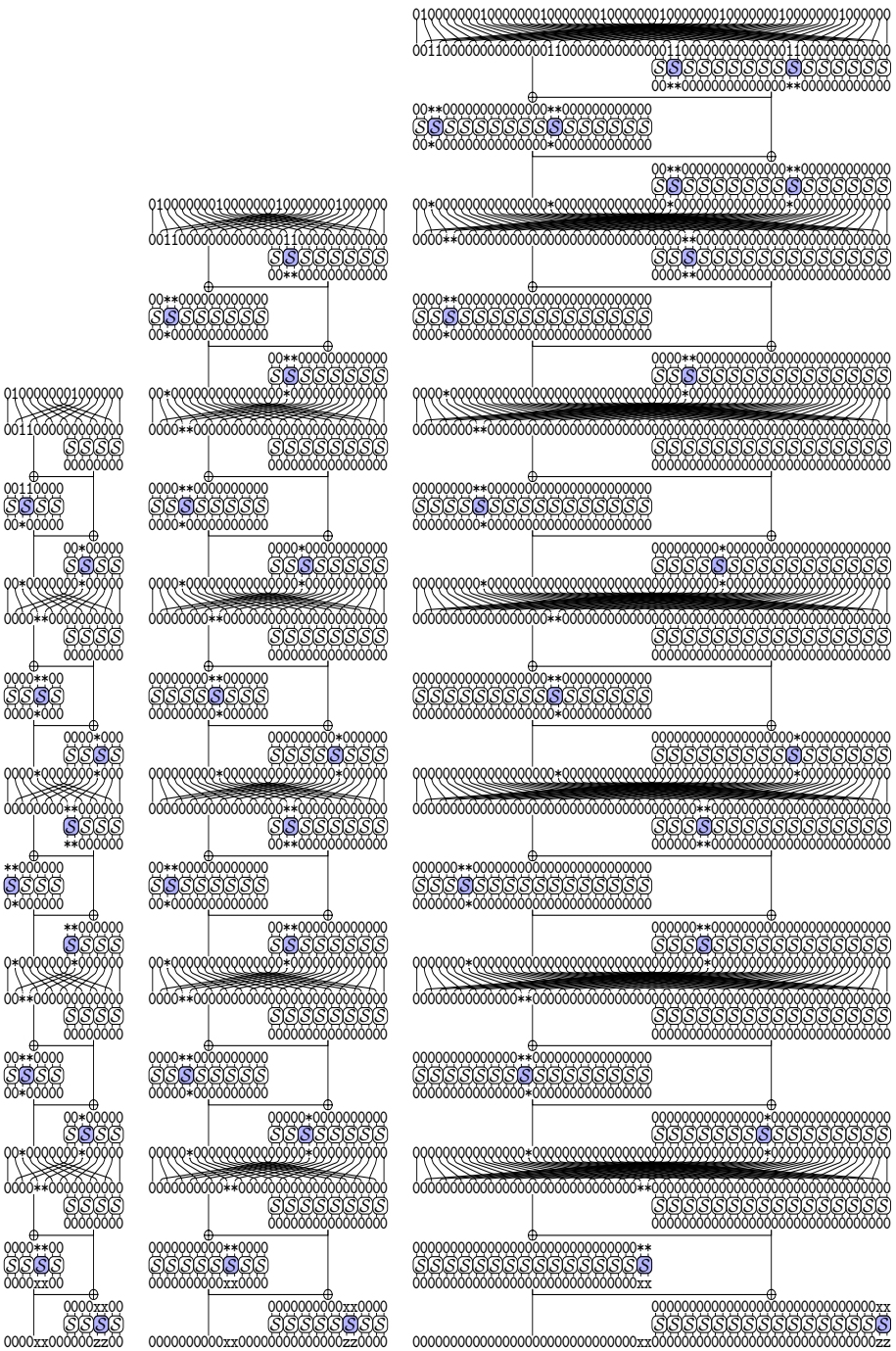
5.1 Differential Characteristics for the Counter Sequence

Recall the generation of the sequence S, as shown in Figure 2a. The intermediate state is updated by calling INC32, incrementing each 32-bit block of the state. Consider the difference between two states S_i and S_{i+1}: The only difference between the inputs to the final call to PF_{K_3} is one additional call to INC32. A little-endian addition by 1 behaves like an XOR operation with probability $\frac{1}{2}$ (exactly when the least significant bit of the state is zero). Therefore, the call INC32 behaves like an XOR with a probability of 2^{-2} (2^{-4}, 2^{-8}) for FlexAEAD-64 (FlexAEAD-128, FlexAEAD-256). This process is shown in Figure 2b.

Consider the input difference $\Delta_{\text{in}} =$ 01000000 01000000 for FlexAEAD-64 (repeated twice for FlexAEAD-128 and four times for FlexAEAD-256). The best truncated differential characteristics for PF starting with this input difference we were able to find are depicted in Figure 7. An explanation on the tool we used to find the differentials – a combined Mixed-Integer Linear Programming (MILP) [8,18] and Constraint Programming (CP) / Satisfiability (SAT) model, as well as the best exact differential characteristics we were able to find, can be found in [3].

5.2 Forgery Attacks for FlexAEAD Using the Counter Difference

We can now use these differentials $\Delta_{\text{in}} \to \Delta_{\text{out}}$ in the counter sequence to mount forgery attacks on the full FlexAEAD-64, FlexAEAD-128, and FlexAEAD-256 schemes. In the following, we describe several different approaches.



(a) F-64: 2^{-46} (b) FlexAEAD-128: 2^{-54} (c) FlexAEAD-256: 2^{-70}

Fig. 7: Truncated differential characteristics for full-round PF_K in FlexAEAD variants.

Changing Associated Data. We query a tag for some plaintext P with associated data $A = A_0 \,||\, A_1$, where $A_0 \oplus A_1 = \Delta_{\text{out}}$. With a probability of about 2^{-46} (FlexAEAD-64), 2^{-54} (FlexAEAD-128), or 2^{-70} (FlexAEAD-256), the sequence blocks S_0 and S_1 follow the cluster of differential characteristics, and therefore also fulfill $S_0 \oplus S_1 = \Delta_{\text{out}}$. Then, $A_0 \oplus A_1 \oplus S_0 \oplus S_1 = 0$, so $S_0 \oplus A_0 = S_1 \oplus A_1$, resulting in a contribution of the two associated data blocks to the checksum of $\mathsf{PF}_{K_2}(S_0 \oplus A_0) \oplus \mathsf{PF}_{K_2}(S_1 \oplus A_1) = 0$.

Now, if we swap A_0 and A_1, with the same reasoning, the contribution to the checksum will again be 0, so the original tag is valid for the modified associated data with swapped blocks.

Although the example above assumes a distance of 1 between associated data blocks, we can generalize this property and also find similar differential characteristics for higher distances j. Distances with lower hamming weight and with several suitable Xor differences following the same truncated difference generally result in a better probability. In practical experiments on round-reduced FlexAEAD, we observed an even higher success probability than expected when swapping associated data blocks, such as examples with a non-zero, but constant contribution to the checksum.

Truncating Ciphertext. In a similar fashion to the previous attack, we can also use this strategy to create a forgery targeting the plaintext.

Again, consider the generation of the sequence S, using the same strategy and differential characteristics as in Section 5.1. Now query a tag with a plaintext $P = P_0 \,||\, \cdots \,||\, P_{m-2} \,||\, P_{m-1}$, where $P_{m-2} \oplus P_{m-1} = \Delta_{\text{out}}$. With the same reasoning and success probability as before, the combined contribution to the checksum of P_{m-2} and P_{m-1} is 0, since, like in the previous attack, $P_{m-2} \oplus S_{n+m-2} = P_{m-1} \oplus S_{n+m-1}$, and therefore $\mathsf{PF}_{K_2}(S_{n+m-2} \oplus P_{m-2}) \oplus \mathsf{PF}_{K_2}(S_{n+m-1} \oplus P_{m-1}) = 0$.

We can now produce a forgery by truncating the last two ciphertext blocks, since the contribution of the corresponding plaintext blocks to the checksum and therefore the tag is 0, and the number of blocks does not influence the tag.

Reordering Ciphertext. For their submission to the NIST Lightweight Cryptography standardization project, the designers of FlexAEAD updated their design from the previous version FlexAE in order to include associated data and prevent trivial reordering attacks. We show a forgery based on reordering ciphertexts of a chosen-plaintext query. Again, this attack is based on the same property of the sequence S as the two previous attacks.

Consider a chosen plaintext $P = P_0 \,||\, P_1$, where $P_0 \oplus P_1 = \Delta_{\text{out}}$, and the corresponding ciphertext $C = C_0 \,||\, C_1$ and tag T. As before, this difference of 1 in the block index results in the differential characteristics depicted in Figure 7. In FlexAEAD, the sequence values S_0 and S_1 are added at two points during the encryption process, so that the internal difference Δ_{out} propagates as shown in Figure 8. By now swapping the ciphertext blocks C_0 and C_1, we have a valid forgery using the original tag T. If the sequence generation followed the chosen characteristic, the two swapped ciphertext blocks will again have a checksum

contribution of 0 during the decryption process. However, the resulting plaintext blocks are unpredictable.

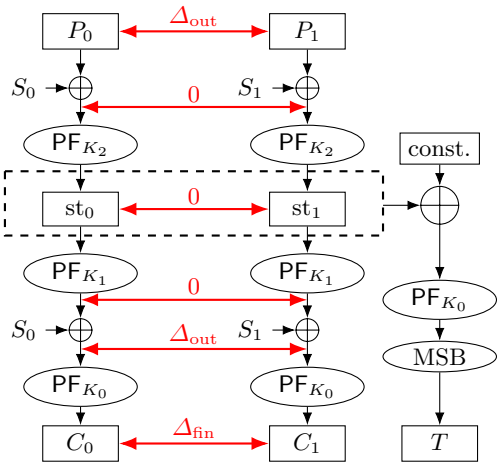


Fig. 8: Propagation of differences in the FlexAEAD encryption function, assuming $S_0 \oplus S_1 = \Delta_{\text{out}}$.

6 Applicability to FlexAE and Variants of FlexAEAD

In this section we discuss the applicability of our attacks to FlexAE (the predecessor of FlexAEAD), and the effect of the tweaks proposed by the designers of FlexAEAD on our results.

6.1 Applicability to FlexAE

FlexAE, published at IEEE ICC 2017 [10], is the predecessor design of FlexAEAD. It features a slightly simpler mode that omits the step $\mathsf{PF}_{K_1}(\cdot) \oplus S_j$ in the computation of ciphertext block C_j and does not support associated data A (see Figure 9). The primitive also shows minor differences, such as 3 slightly different S-boxes, which have no significant impact on the security analysis. In addition, the ordering of the subkeys used in FlexAEAD and FlexAE are different. Since this does not impact the security, we use notations that are similar to FlexAEAD in our analysis. The additional steps in FlexAEAD were added by the designers to fix problems in FlexAE; in particular, the ciphertext reordering attack of Section 5.2 works with probability 1 for FlexAE.

Forgery attacks. All forgery attacks on FlexAEAD described in Section 5 apply without change to FlexAE, since the part of the algorithm they target remains

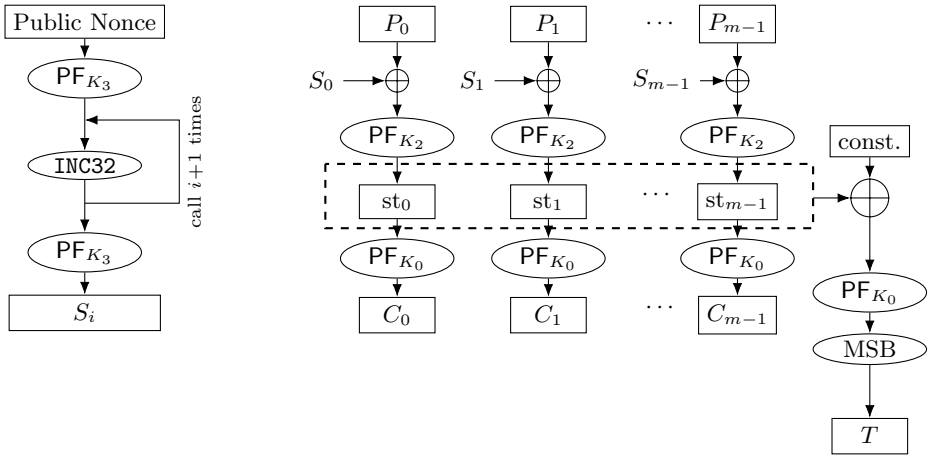


Fig. 9: The FlexAE mode for authenticated encryption (simplified, from [10]). For consistency with FlexAEAD, we denote the subkeys as K_0, K_2 and K_3 rather than K_0, K_1 and K_2.

unchanged. Furthermore, FlexAE permits forgeries with zero encryption queries, as illustrated in Figure 10. The following is a forgery with probability about 2^{-54} for FlexAE-64-128 (or, with similar characteristics, 2^{-86} for FlexAE-128-256, 2^{-150} for FlexAE-256-512, or 2^{-278} for FlexAE-512-1024): Take an arbitrary nonce N and a single-block ciphertext C, and select $T = C \oplus \Delta_{\text{out}}$ with $\Delta_{\text{out}} =$ xx000000 zz000000, where xx is an arbitrary nonzero difference and zz is the high-probability S-box output difference with $\mathbb{P}[\texttt{xx} \to \texttt{zz}] = 2^{-6}$. This works (i.e., the pair (C, T) is valid) based on the differential with input difference $\Delta_{\text{in}} =$ 10101010 10101010 for PF_{K_0} due to the constant addition between the computation of a tag T and a single-block ciphertext C. The actual success probability is likely higher; there are several similar clusters contributing to the same differential (see Figure 10), and the alternative padding constant $(\texttt{01})^*$ instead of $(\texttt{10})^*$ gives an alternative compatible Δ_{in} to double the probability.

In this aspect, the transition from FlexAE to FlexAEAD indeed strengthened the design.

Key-recovery attacks. Interestingly, it appears that the key recovery attacks described in Section 4 do not directly apply to FlexAE. Specifically, the first two phases of the attacks work without change and allow recovering the full subkey K_3 and the sequence $\{S_i\}$. However, the third phase of the attack (which is intended to recover the second half of K_0) exploits the fact that in FlexAEAD, the value S_i is interleaved with the data again before the last application of PF_{K_0}. This allows an adversary to induce difference into the input of PF_{K_0} (as is shown in Figure 8) and observe the corresponding output difference in the ciphertext. In FlexAE, the value S_i is interleaved with the data only once, and there is no direct way to observe both the input and the output of a single PF_K function.

However, we found an alternative attack, with a higher complexity, but still practical. The idea is to consider two-block messages, and to study the differential propagation in PF_{K_2} between the two blocks of a given message, instead of studying pairs of messages. We start with a large set of messages, for which the corresponding inputs of PF_{K_2} constitute a pair with a convenient input difference $P_0 \oplus S_1 \oplus P_1 \oplus S_2$. We observe that the tag value is computed directly from the output difference of PF_{K_2} (that is to say, $\mathsf{PF}_{K_2}(P_0 \oplus S_1) \oplus \mathsf{PF}_{K_2}(P_1 \oplus S_2)$). In particular, if there exist high probability differential characteristics, we can detect when two different messages follow the same trail, because the corresponding tags collide.

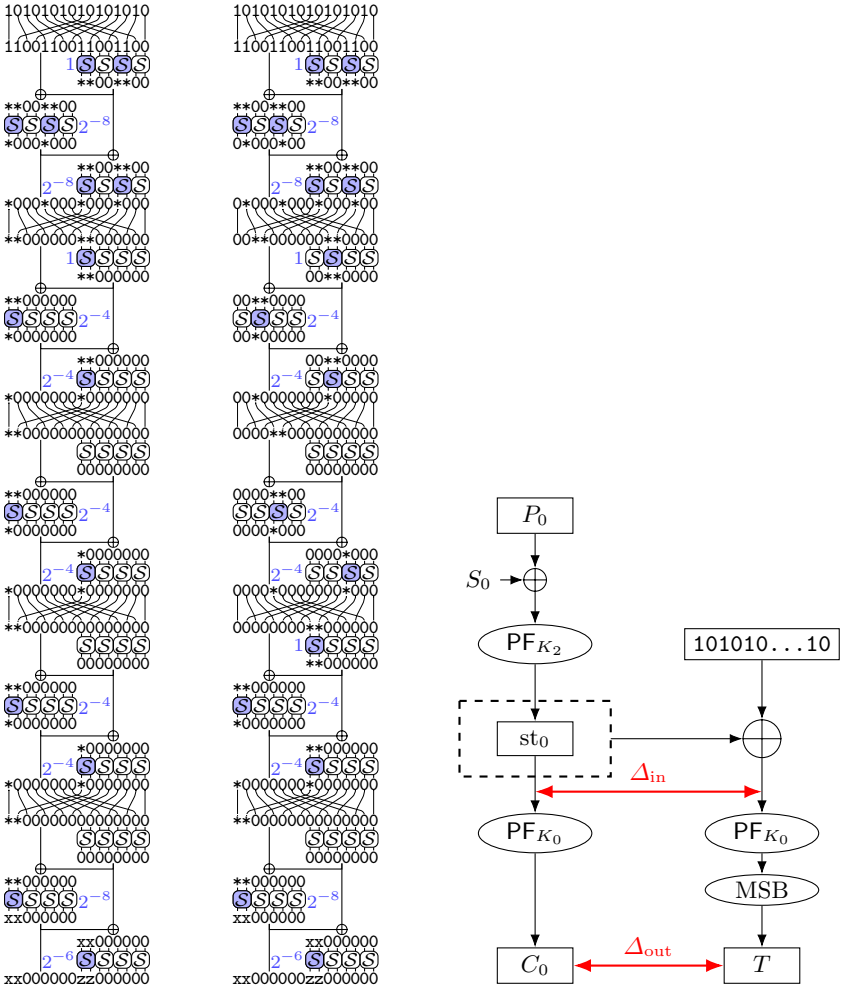


Fig. 10: Zero-query forgery for full FlexAE-64-128 with clusters of probability 2^{-54}.

Starting from a difference in a single Super S-box, we know that at each round there is a probability of 2^{-7} to stay with a single active Super S-box (either the same one or a different one). If we do not constrain the last round, we obtain one of 2^{18} output differences with four active nibbles (in two different Super S-boxes) with probability 2^{-28}.

Using a set of 2^{38} messages, we expect that 2^{10} of them follow one of the differentials, and we expect on average two collisions between the tags due to the differentials (there are $\binom{2^{10}}{2} \approx 2^{19}$ pairs of right messages, and the two messages follow the same differential with probability 2^{-18}). In addition, there should be about $\binom{2^{38}}{2} \cdot 2^{-64} \approx 2^{11}$ random collisions (wrong pairs).

In order to recover the key, we use the same procedure as in Step 2 of Phase 1. This procedure recovers candidates for the internal state of PF using only the known input difference and output difference. Here, we do not know exactly the output difference, but we have a list of 2^{18} candidates. Since we have a single active Super S-box in the first round, we also have a smaller filter than in the previous attacks. We expect the procedure of Step 2 of Phase 1 to return a list of 2^{36} state candidates on average, with a time complexity of 2^{36} partial encryptions.

In order to reduce the complexity, we fix the input difference to be the same for all messages, so that we only run the procedure once, and we deduce a list of 2^{47} candidates for the key, from the list of 2^{36} internal state candidates and the list of 2^{11} collisions detected. Wrong key candidates can be filtered by checking whether the other message in the collision also follows the differential (requiring two evaluations of PF_{K_2}).

Finally, this algorithm recovers the key K_A^2 with high probability, with a data complexity of 2^{38} two-block messages, and a time complexity dominated by 2^{48} evaluations of PF_{K_2}.

After K_A^2 is recovered, we can control the input differences to PF_{K_0} and use the same strategy as in Phase 3 of the attack of Section 4 to recover K_B^0. Then, K_A^0 can be found instantly using the tag generation of an empty message (like in Phase 4 of the attack in Section 4), and the second half of K_2 can be found instantly, as the value before its insertion is known from the plaintext side and the value after its insertion is known from the ciphertext side.

Practical verification. We have implemented Phases 1 and 2, and the variant of Phase 3 of the FlexAE-64 attack. We used the reference code of FlexAEAD-64, because there is no reference code available for FlexAE, but these phases use parts of the design that are identical in FlexAEAD and FlexAE. However, since the reference implementation of FlexAEAD does not match the specification (as explained in Section 4.2), Phase 1 recovers K_A^2 rather than K_A^3, Phase 2 recovers $K_B^2 \oplus K_A^3$ rather than K_B^3, and the new Phase 3 recovers $K_A^2 \oplus K_B^3$ rather than K_A^2. We also have to deal with carries in the `INC32` operation in Phase 3, but this can be done without increasing the complexity (when attacking the variant corresponding to the reference code, the keys recovered at that point are not sufficient to compute exactly the S_i's).

The attack succeeds as expected, with less than three days of computation on a 24-core (48-thread) machine. Due to the long running time, we only ran the attack once, but this is sufficient to verify that the strategy is successful. Due to memory limitations, the new Phase 3 was run with a set of $2^{37.9}$ messages, and 1793 collisions were detected, matching the analysis. We note that we find around $2^{33.5}$ state candidates rather than 2^{36}, so that the total complexity is actually around $2^{45.5}$ rather than 2^{48}.

Time-data tradeoff. It is also possible to reduce the time complexity at the cost of an increased data complexity using three-way collisions. Starting with a set of 2^{41} messages, we expect to have 2 three-way collisions in the tags due to messages following the differential, while three-way collisions should not happen by chance with those parameters. This results in an attack with a data complexity of 2^{41} two-block messages and time complexity dominated by the encryption of the messages (the key-recovery part requires only 2^{37} partial encryptions).

This complexity is higher than the corresponding complexity for FlexAEAD-64. Hence, in this aspect, the transition from FlexAE to FlexAEAD *weakened the cipher*, making it more vulnerable to a practical attack recovering the full subkey.

Forgery attack against FlexAE-128 and FlexAE-256. The forgery attack can also be adapted to larger variants of FlexAE. Since Phase 1 has a significantly higher complexity for larger variants, it dominates the modified Phase 3, and the complexity of the attack against FlexAE and FlexAEAD is the same.

More precisely, we use the time-data trade-off as above in order to avoid dealing with wrong pairs. For FlexAE-128, we obtain one of 2^{18} output differences with four active nibbles (in two different Super S-boxes) with probability 2^{-35}. Starting from a set of 2^{48} messages, we have 2^{13} messages reaching one of the high probability output differences, and we expect a three-way collision. Starting from the collision, we extract 2^{10} candidates for a 32-bit partial key. After repeating the procedure 4 times, we obtain 2^{40} candidates for K_A^2 that we filter with some additional queries. For FlexAE-256 the procedure is similar. The overall complexity of this attack is 2^{59} for FlexAE-128 and 2^{140} for FlexAE-256.

6.2 Tweaks Suggested by FlexAEAD's Designers

In response to observations by several authors (see Section 1), the designers of FlexAEAD have informally proposed a number of tweaks to mitigate the issues [13]. Proposed tweaks include, in chronological order:

1. Changing the increment in `INC32` from `0x00000001` to `0x11111111` in order to preclude low-weight Xor-distances between close blocks;
2. Reducing data limits to at most 2^{32} blocks per encryption (message plus associated data) with additional, stricter, limits on associated data length (2^{28} blocks for FlexAEAD-128, 2^{23} blocks for FlexAEAD-256);
3. Modifying the associated data padding to $10 \dots 0$ and executing PF_{K_2} twice for the last block unless the original length was exactly a multiple of the block length, in which case the padding is omitted;

4. Significantly strengthening the linear layer with an additional diffusion step;
5. Including a function of the final counter after associated data processing, $\mathsf{PF}_{K_2}(\mathsf{PF}_{K_2}(S_n))$, in the checksum computation.

Changes (2), (3) and (5) appear to mitigate the issues related to the mode raised in [3,7], but do not affect the attacks presented in this paper (since we never use long messages in our attacks). Change (1) increases the complexity of our attacks, due to the need to wait for many blocks until the counter increment leads to a convenient difference. However, it does not prevent the attacks completely, even together with Change (2). For example, consider associated data blocks A_0 and A_j with $j = 2^{28} = \texttt{0x10000000}$ for FlexAEAD-64. The corresponding counter increment between these blocks is $j \cdot \texttt{0x11111111} \equiv j \pmod{2^{32}}$, which corresponds to an input XOR difference of $\Delta_{\text{in}} = \texttt{000000*0 000000*0}$ with very high probability. This leads to an attack with the same success probability as the one in Section 5 using a truncated characteristic closely related to the one in Figure 7a (with mirrored truncated difference patterns at the input to each round). The key recovery attack described in Section 4 works with similar changes.

Unless more significant changes such as (4) are considered, it is necessary to apply smaller data limits (a) for all variants and (b) for both associated data and plaintext. In addition, a more detailed analysis of potential tradeoffs with a slightly higher number of active nibbles in Δ_{in} with a lower success probability, but also a sufficiently low distance between the swapped data blocks, is recommended.

We have not analyzed the impact of the most significant change (4) in detail, but it appears to significantly improve resistance against differential cryptanalysis and thus against all attacks presented in this paper.

7 Conclusion

In this paper we presented a fully verified practical key recovery attack on FlexAEAD-64, as well as forgery and key recovery attacks on FlexAEAD-128, FlexAEAD-256, and FlexAE. Our attacks are based on a strong clustering effect which leads to a powerful truncated differential attack on the internal keyed permutation PF_K, together with interplay between various parts of the mode. The result – practical key recovery – is rather rare in modern designs, and demonstrates the fragility of authenticated encryption schemes whose internal permutation has non-optimal properties. Interestingly, our most powerful key-recovery attack does not apply to FlexAE– the predecessor of FlexAEAD. It appears that the insertion of an additional encryption layer in the transition to FlexAEAD, which was intended to increase the security level, actually made the cipher more vulnerable.

Acknowledgements. We thank the designers of FlexAE and FlexAEAD for their comments on a preliminary version of this analysis on the NIST LWC mailing list. Some of the results presented in this paper were obtained during a

workshop dedicated to cryptanalysis of the NIST lightweight candidates, held in the framework of the European Research Council project 'LightCrypt' (ERC StG no. 757731). We thank the participants of the workshop who contributed to the discussion on FlexAEAD, and in particular Tomer Ashur, Roberto Avanzi, Anne Canteaut, Itai Dinur, Eran Lambooij, Eyal Ronen, and Yu Sasaki, for their valuable suggestions.

References

1. Bertoni, G., Daemen, J., Peeters, M., Van Assche, G., Van Keer, R.: Ketje v2. Submission to CAESAR: Competition for Authenticated Encryption. Security, Applicability, and Robustness (Round 3) (2014), http://competitions.cr.yp.to/round3/ketjev2.pdf
2. Daemen, J., Rijmen, V.: The Design of Rijndael: AES – The Advanced Encryption Standard. Information Security and Cryptography, Springer (2002), https://doi.org/10.1007/978-3-662-04722-4
3. Eichlseder, M., Kales, D., Schofnegger, M.: Forgery attacks on flexae and flex-aead. In: Albrecht, M. (ed.) Cryptography and Coding - 17th IMA International Conference, IMACC 2019, Oxford, UK, December 16-18, 2019, Proceedings. Lecture Notes in Computer Science, vol. 11929, pp. 200–214. Springer (2019). https://doi.org/10.1007/978-3-030-35199-1_10, https://doi.org/10.1007/978-3-030-35199-1_10
4. Jutla, C.S.: Encryption modes with almost free message integrity. In: Pfitzmann, B. (ed.) Advances in Cryptology – EUROCRYPT 2001. LNCS, vol. 2045, pp. 529–544. Springer (2001), https://doi.org/10.1007/3-540-44987-6_32
5. Jutla, C.S.: Encryption modes with almost free message integrity. Journal of Cryptology **21**(4), 547–578 (2008), https://doi.org/10.1007/s00145-008-9024-z
6. Kam, J.B., Davida, G.I.: Structured design of substitution-permutation encryption networks. IEEE Transactions on Computers **28**(10), 747–753 (1979), https://doi.org/10.1109/TC.1979.1675242
7. Mège, A.: Official comment: FlexAEAD. Posting on the NIST LWC mailing list, https://groups.google.com/a/list.nist.gov/d/msg/lwc-forum/DPQVEJ5oBeU/YXWOQjfjBQAJ
8. Mouha, N., Wang, Q., Gu, D., Preneel, B.: Differential and linear cryptanalysis using mixed-integer linear programming. In: Wu, C., Yung, M., Lin, D. (eds.) Information Security and Cryptology – Inscrypt 2011. LNCS, vol. 7537, pp. 57–76. Springer (2011), https://doi.org/10.1007/978-3-642-34704-7_5
9. do Nascimento, E.M.: Algoritmo de Criptografia Leve com Utilização de Autenticação. Ph.D. thesis, Instituto Militar de Engenharia, Rio de Janeiro (2017), http://www.comp.ime.eb.br/pos/arquivos/publicacoes/dissertacoes/2017/2017-Eduardo.pdf
10. do Nascimento, E.M., Xexéo, J.A.M.: A flexible authenticated lightweight cipher using Even-Mansour construction. In: IEEE International Conference on Communications – ICC 2017. pp. 1–6. IEEE (2017), https://doi.org/10.1109/ICC.2017.7996734
11. do Nascimento, E.M., Xexéo, J.A.M.: A lightweight cipher with integrated authentication. In: Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais – SBSEG. pp. 25–32. Sociedade Brasileira de

Computação (2018), https://portaldeconteudo.sbc.org.br/index.php/sbseg_estendido/article/view/4138
12. do Nascimento, E.M., Xexéo, J.A.M.: FlexAEAD. Submission to Round 1 of the NIST Lightweight Cryptography Standardization process (2019), https://csrc.nist.gov/CSRC/media/Projects/Lightweight-Cryptography/documents/round-1/spec-doc/FlexAEAD-spec.pdf
13. do Nascimento, E.M., Xexéo, J.A.M.: Official comment: FlexAEAD. Posting on the NIST LWC mailing list (2019), https://csrc.nist.gov/CSRC/media/Projects/Lightweight-Cryptography/documents/round-1/official-comments/FlexAEAD-official-comment.pdf
14. National Institute of Standards and Technology (NIST): Lightweight cryptography standardization process (2019), https://csrc.nist.gov/projects/lightweight-cryptography
15. National Institute of Standards and Technology (NIST): Status report on the first round of the nist lightweight cryptography standardization process (2019), https://nvlpubs.nist.gov/nistpubs/ir/2019/NIST.IR.8268.pdf
16. Rahman, M., Saha, D., Paul, G.: Cryptanalysis of FlexAEAD. In: Nitaj, A., Youssef, A.M. (eds.) Progress in Cryptology - AFRICACRYPT 2020 - 12th International Conference on Cryptology in Africa, Cairo, Egypt, July 20-22, 2020, Proceedings. Lecture Notes in Computer Science, vol. 12174, pp. 152–171. Springer (2020). https://doi.org/10.1007/978-3-030-51938-4_8, https://doi.org/10.1007/978-3-030-51938-4_8
17. Wu, H., Preneel, B.: AEGIS v1.1. Submission to CAESAR: Competition for Authenticated Encryption. Security, Applicability, and Robustness (Round 3 and Final Portfolio) (2014), http://competitions.cr.yp.to/round3/aegisv11.pdf
18. Wu, S., Wang, M.: Security evaluation against differential cryptanalysis for block cipher structures. IACR Cryptology ePrint Archive, Report 2011/551 (2011)