IACR Communications in Cryptology
ISSN 3006-5496, Vol. 1, No. 4, 20 pages.
https://doi.org/10.62056/ayfhp2fgx

On the Key-Commitment Properties of Forkcipher-based AEADs

Mostafizar Rahman[1], Samir Kundu[2] and Takanori Isobe[1]

[1] University of Hyogo, Kobe, Japan
[2] Siksha 'O' Anusandhan (Deemed to be) University, Bhubaneswar, India

Abstract. Forkcipher-based AEADs have emerged as lightweight and efficient cryptographic modes, making them suitable for resource-constrained environments such as IoT devices and distributed decryption through MPC. These schemes, including prominent examples like Eevee (Jolteon, Espeon, and Umbreon), PAEF, RPAEF, and SAEF, leverage the properties of forkciphers to achieve enhanced performance. However, their security in terms of key commitment, a critical property for certain applications such as secure cloud services, as highlighted by Albertini et al. (USENIX 2022), has not been comprehensively analyzed until now.
In this work, we analyze the key-commitment properties of forkcipher-based AEADs. We found that the majority of forkcipher-based AEAD schemes lack key-commitment properties, primarily due to the distinctive manner in which they process associated data and plaintext. For two different keys and the same nonce, an adversary can identify associated data and plaintext blocks that produce identical ciphertext-tags with a complexity of $O(1)$. Our findings apply to various forkcipher-based AEADs, including Eevee, PAEF, and SAEF, and naturally extend to less strict frameworks, such as CMT-1 and CMT-4.
These findings highlight a significant limitation in the robustness of forkcipher-based AEADs. While these modes are attractive for their lightweight design and efficiency, their deployment should be restricted in scenarios where explicit robustness or key-commitment security is required.

Keywords: Key committing · Eevee · Forkcipher · Forkskinny

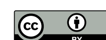
1 Introduction

Key commitment ensures that a ciphertext C can only be successfully decrypted using the exact key that was originally used to encrypt the corresponding plaintext. In cryptographic systems, if it were possible to find a ciphertext that could be decrypted to valid plaintexts under two different keys, it would violate the key commitment principle. In traditional terms, it is generally expected that an Authenticated Encryption with Associated Data (AEAD) scheme should provide confidentiality and integrity for the data. It guarantees that data remains confidential and unchanged during transmission, ensuring both privacy and integrity. This is achieved by combining encryption and message authentication codes (MACs) to protect the data from unauthorized access and tampering.

The necessity of an AEAD scheme being a key-committing one in addition to providing confidentiality and integrity is studied in [GLR17, DGRW18] in the context of Facebook message franking [Fac16, Mil17]. In Facebook's *end-to-end* encrypted messaging, reporting abusive messages requires a balance between user privacy and the need to verify reported content. To address this, Facebook introduced "message franking," a method to

E-mail: mrahman454@gmail.com (Mostafizar Rahman), samirkundu3@gmail.com (Samir Kundu), takanori.isobe@ai.u-hyogo.ac.jp (Takanori Isobe)

This work is licensed under a "CC BY 4.0" license.



Received: 2024-10-09 Accepted: 2024-12-03

include cryptographic proof in abuse reports to verify the reported message's authenticity. In [DGRW18], it is demonstrated to exploit Facebook's message franking scheme, where a malicious user can send an inappropriate image to a recipient without the recipient being able to report it as abuse. This issue arises from the use of a fast but non-committing authenticated encryption (AE) scheme. In [LGR21], it is shown that due to the use of non-committing AEADs, attackers can recover user's password from the Shadowsocks proxy servers. Attacks are also shown on the password-authenticated key exchange protocol OPAQUE [JKX18] when it is implemented using non-committing AEADs [LGR21]. Subsequently, vulnerabilities due to the use of non-committing AEADs also appeared in different contexts, such as key rotation schemes and envelope encryption, as discussed in a recent study [ADG+22].

In recent developments, new definitions have emerged that emphasize committing not just to the key but also to the associated data and nonce [CR22, BH22]. While new schemes have been proposed [CR22, ADG+22] to adhere to these updated definitions, there remains uncertainty regarding the implementation of commitment in existing AEAD schemes. The specific mechanisms through which these existing schemes ensure commitment, particularly regarding the associated data and nonce, requires further investigations. Clarification on these aspects is essential for understanding the security guarantees provided by different AEAD schemes. Recently, committing security analysis is carried out on several existing AEAD schemes like CCM, GCM, OCB3 [MLGR23], Ascon [NSS23], AEZ [CFI+23], Aegis, Rocca-S [DFI+24].

In Asiacrypt 2019, the forkcipher primitives and their associated AEAD modes (PAEF, SAEF, and RPAEF) were introduced [ALP+19]. Following this, many forkcipher schemes were proposed [KLL20, ABPV21, AW23, DDLM24, Man24]. In ACM CCS 2023, forkcipher-based AEAD schemes Eevee [BPA+23] was proposed. These schemes are based on the ForkSkinny forkcipher [ALP+19], offering lightweight and efficient modes designed for IoT devices and enabling distributed decryption through MPC. These modes, including Umbreon, Jolteon, and Espeon, are fully parallelizable in decryption, making them highly efficient. Umbreon prioritizes security, providing full OAE security that degrades logarithmically with nonce-misuse. Jolteon focuses on performance, with smaller state requirements but lower security under nonce-misuse compared to Umbreon. Espeon offers an intermediate trade-off, providing similar performance to Jolteon but with tweak-size-dependent security under nonce-misuse. Unlike existing designs, Eevee modes are based on a single primitive, providing simplicity and improved security against relevant threats. It is important to emphasize that Eevee was initially proposed with a focus on secure computation in IoT-to-cloud scenarios. As demonstrated in [ADG+22], AEAD schemes employed in cloud services should provide key-commitment security. A failure to provide this security property could enable adversaries to exploit the structural properties of the scheme, leading to ciphertext and tag forgery attacks. This highlights the need for higher-level protocols employing AEADs to carefully consider the type of AEAD they use. If the security of a protocol depends on key-commitment and non-committing AEADs are employed, it could lead to significant vulnerabilities and potentially compromise the entire protocol. Our analysis focuses on evaluating the key-commitment properties of Eevee and identifying any associated risks, ensuring the scheme's suitability for its intended applications. Our attacks on the key-committing security of Eevee indicate that Eevee may not be suitable for general use-cases, although it remains appropriate for scenarios where commitment is not required. This aligns with the suggestion from [LGR21]- "*We suggest considering a shift towards committing AEAD being the default for general use and using non-committing AEAD only for applications shown to not require robustness.*"

Our Contribution. In Asiacrypt 2019, a new symmetric-key primitive called the *forkcipher* and its associated AEAD modes- PAEF, SAEF, and RPAEF were introduced

for lightweight applications [ALP+19]. Later, a forkcipher-based AEAD family named Eevee was proposed, focusing on cloud service applications. The key-commitment security of such cloud services, particularly when using AEAD schemes, is crucial, as highlighted in [ADG+22], which shows that AEAD schemes lacking key-commitment security can lead to serious vulnerabilities, as demonstrated through attacks on the envelope encryption scheme used by the AWS encryption SDK. This suggests that even well-designed AEAD schemes, when employed in cloud services, might be susceptible to attacks if they do not ensure key-commitment. Given that cloud-based secure computation is one of the primary application domains for Eevee, it becomes imperative to investigate its potential weaknesses against key-commitment attacks.

We assess here the key committing security of the AEAD modes in Eevee family. We demonstrate that certain fork-cipher based AEAD schemes do not exhibit key-committing properties within the rigorous FROB game framework [FOR17]. Our analysis strategy exploit the processing of the associated data and plaintext. We demonstrate that for two different keys and the same nonce, as adversary can always find a corresponding associated data block and plaintext blocks so that two equal ciphertext-tags are generated. Consider K^1, K^2 be two keys and N is a nonce. Consider (K^1, N, A^1, P^1) generates a ciphertext-tag pair (C,τ) where A^1, P^1 are the associated data and plaintext, respectively. Then using our attack, one can find a block associated data A^2 and a plaintext P^2 (where the size of P^2 and P^1 are equal) such that (K^2, N, A^2, P^2) generates the same (C,τ). This allows us to find tag collisions with a complexity of $O(1)$. Moreover, the absence of such committing properties in the FROB framework implies that these schemes lack committing properties in less strict frameworks such as CMT-1 and CMT-4. Further, we demonstrate the applicability of our strategy to the initial forkcipher-based AEAD modes, PAEF and SAEF, revealing their potential weaknesses against key-committing attacks. Since the attack on SAEF is similar to the attack on PAEF, only the attack on PAEF is implemented. Similarly, as the attacks on the three modes of Eevee are similar, only the attack on Umbreon is implemented (the attacks on SAEF, Jolteon and Espeon are not implemented due to their similarity with the other implemented attacks). We note that no claims regarding the key-committing properties of these ciphers have been made in [ALP+19, BPA+23]. Furthermore, our attacks merely demonstrate that most of these ciphers lack key-committing properties and do not compromise their claimed AEAD security. Finally, we discuss the challenges of devising dedicated countermeasures for these schemes without incurring significant overhead. As a result, we consider the generic countermeasures that have been proposed previously.

Structure of the paper. The rest of the paper is organized as follows. In Section 2, along with introducing some notations, discussions on committing authenticated encryption frameworks and the specification on Eevee and other forkcipher-based AEAD modes are provided. An overview of the attack strategy is described in Section 3.1. Dedicated attacks on Eevee and PAEF, SAEF are presented in Section 3.2 and Section 3.3, respectively. Section 4 discusses about the challenges of devising dedicated countermeasures for these schemes, considering small overheads. Finally, the concluding remarks are furnished in Section 5. The attack vectors of the implemented attacks are provided in Appendix A.

2 Preliminaries

First, we introduce some notations that are followed throughout the paper. Next, we delve into the concepts surrounding committing authenticated frameworks. Finally, we provide a brief overview of the Eevee family of AEAD modes.

2.1 Notations

$\langle i \rangle_d$: d-bit encoding of a number i

$F_{K,T,s}(M)$: Encryption of a message M using a forkcipher F with key K and tweak T. For $s = 0$, 1 or b, left, right or both ciphertext blocks are given as output.

$F^{-1}_{K,T}(C)$: Decryption of a ciphertext C using a forkcipher F with key K and tweak T. Only the encrypted message is given as output.

$F^{R}_{K,T}(C)$: Outputs the other ciphertext block, when queried using a ciphertext block C.

$|X|$: represents the size of vector X in number of bits

$X_0 \cdots X_m X_* \xleftarrow{n} X$: Divide a vector X into several vectors, where only X_* has a size less than n bits, while $X_0, \ldots, X_m$ have exactly n bits.

$\overline{X}$: Bit-wise complement of the vector X

$Tr^l(X)$: Trims the first l bits from the vector X

2.2 Committing Authenticated Encryption (AE) Frameworks

Consider a symmetric encryption scheme Σ with encryption and decryption algorithms denoted by Σ_{Enc} and Σ_{Dec}, respectively, defined as:

$$\Sigma_{Enc} : \mathcal{K} \times \mathcal{N} \times \mathcal{AD} \times \mathcal{P} \rightarrow \mathcal{C},$$

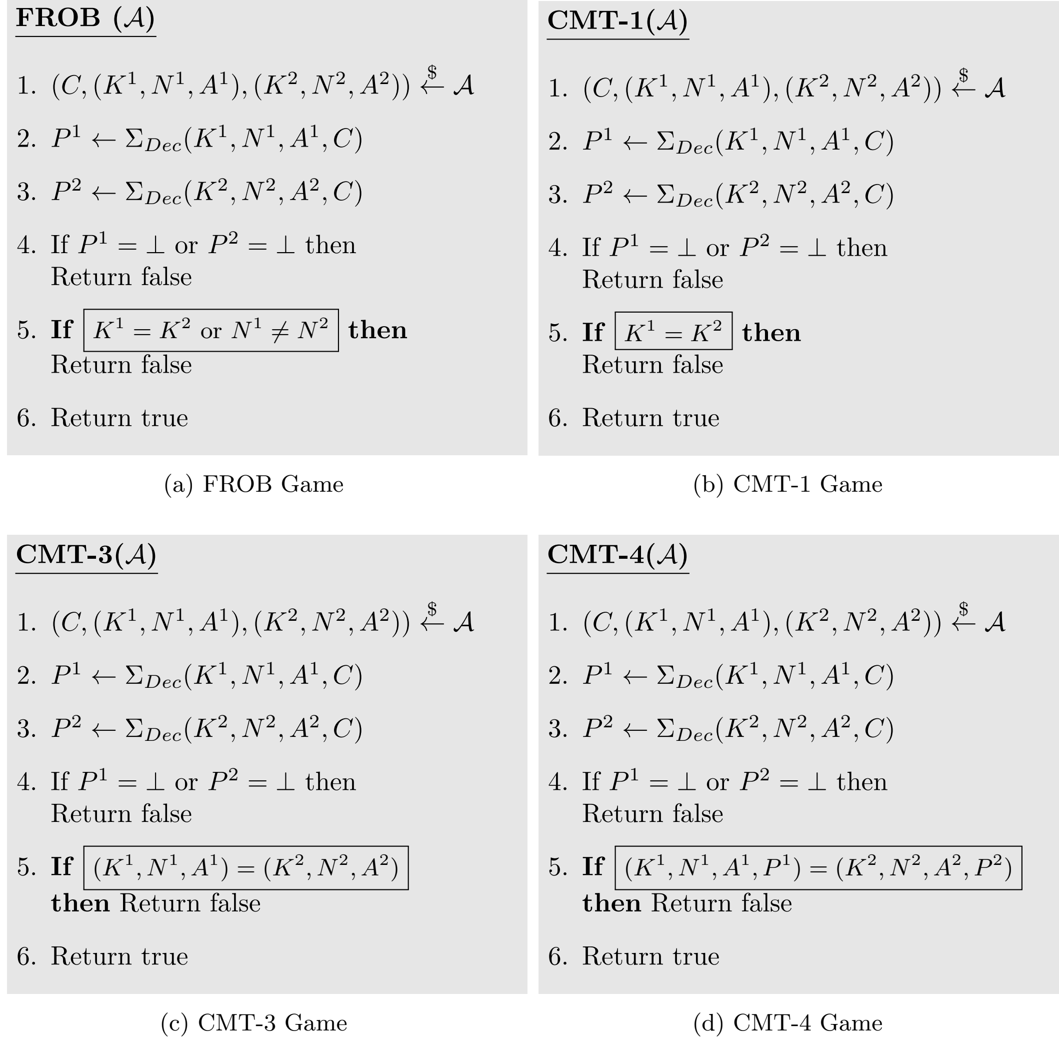
and

$$\Sigma_{Dec} : \mathcal{K} \times \mathcal{N} \times \mathcal{AD} \times \mathcal{C} \rightarrow \mathcal{P} \cup \{\perp\},$$

where $\mathcal{K}$, $\mathcal{N}$, $\mathcal{AD}$, $\mathcal{P}$, and $\mathcal{C}$ represent the key, nonce, associated data, plaintext/message, and ciphertext spaces, respectively. This scheme is formally referred to as a "nonce-based authenticated encryption scheme supporting associated data" or an nAE scheme.

A committing authenticated encryption (cAE) scheme ensures that the key, nonce, associated data, or message used to create a ciphertext is definitively determined. In this framework, the adversary's goal is to create a ciphertext that can be derived from two different sets of keys, nonces, associated data, and messages. Consider $C^i \leftarrow \Sigma_{Enc}(K^i, N^i, A^i, P^i)$ where $K^i \in \mathcal{K}$, $N^i \in \mathcal{N}$, $A^i \in \mathcal{AD}$, $P^i \in \mathcal{P}$, and $C^i \in \mathcal{C}$ for $i \in 1, 2$. The adversary seeks to find C^1 and C^2 such that $C^1 = C^2$ while $(K^1, N^1, A^1, P^1) \neq (K^2, N^2, A^2, P^2)$.

Various committing security frameworks have been proposed, such as CMT-1, where the ciphertext solely commits to the key. In this scenario, the adversary must produce $((K^1, N^1, A^1, P^1), (K^2, N^2, A^2, P^2))$ such that $K^1 \neq K^2$ and $\Sigma_{Enc}(K^1, N^1, A^1, P^1) = \Sigma_{Enc}(K^2, N^2, A^2, P^2)$. The CMT-4 framework relaxes these constraints, allowing the commitment to encompass any input of Σ_{Enc}, not just the key. Here, the adversary can breach CMT-4 security by constructing a set $((K^1, N^1, A^1, P^1), (K^2, N^2, A^2, P^2))$ where $(K^1, N^1, A^1, P^1) \neq (K^2, N^2, A^2, P^2)$ and $\Sigma_{Enc}(K^1, N^1, A^1, P^1) = \Sigma_{Enc}(K^2, N^2, A^2, P^2)$. Bellare and Hoang introduced CMT-3 [BH22], which is slightly more restrictive than CMT-4, replacing the constraint $(K^1, N^1, A^1, P^1) \neq (K^2, N^2, A^2, P^2)$ with $(K^1, N^1, A^1) \neq (K^2, N^2, A^2)$. The FROB game, originally proposed by Farshim, Orlandi, and Rosie and



(a) FROB Game (b) CMT-1 Game (c) CMT-3 Game (d) CMT-4 Game

Figure 1: Different Frameworks for Committing Security.

later adapted to the AEAD setting in [ADG+22], imposes an even stricter condition, requiring $N^1 = N^2$ in addition to $K^1 \neq K^2$. It has been shown that CMT-3 security implies CMT-1, which in turn implies the FROB game. This hierarchy of security notions demonstrates the increasing challenge for adversaries, with the FROB game presenting the most formidable obstacle. All the related games are outlined in Fig. 1.

2.3 Forkcipher and Associated AEAD Modes

Here, we discuss about the forkcipher-based AEAD modes. First of all, we provide a brief description of forkcipher primitive. Then, we briefly describe PAEF, SAEF, RPAEF and Eevee.

2.3.1 Forkcipher

The notion of forkcipher was introduced by Andreeva, Reyhanitabar, Varici and Vizár [ARVV18] which is tweakable symmetric-key primitive with a fixed input length and an expanding fixed output length. It takes as input an n-bit message M, a public tweak T, and a

secret key K, and produces two n-bit ciphertexts C_0 and C_1. The message M can be reconstructed from either C_0 or C_1. Moreover, one ciphertext block can be reconstructed from the other. The encryption algorithm of a forkcipher can be formally defined as $F : \{0,1\}^k \times \{0,1\}^t \times \{0,1\}^n \times \{0,1,b\} \rightarrow \{0,1\}^n \cup \{\{0,1\}^n \times \{0,1\}^n\}$, where k, n, and t represent the key size, block size, and tweak size of F, respectively. Additionally, there is a selector s that determines the type of output. If $s = 0$ or $s = 1$, the output is C_0 or C_1, respectively. If $s = b$, both C_0 and C_1 are output. Fig. 2 illustrates the encryption of a message using a forkcipher.

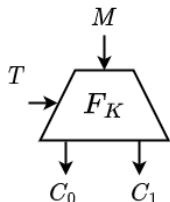


Figure 2: Encryption of a message M using a Forkcipher F with key K and tweak T.

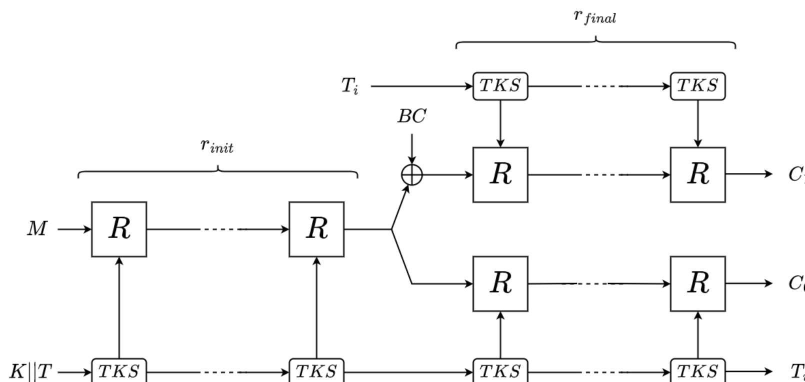


Figure 3: Encryption using ForkCipher. The ciphertext $C_0||C_1$ is computed using the plaintext M and key-tweak pair $K||T$. Here, R, TKS and BC denote the round function, tweak-key scheduling function and branch-constant, respectively. The total number of rounds before and after the forking is denoted by r_{init} and r_{final}, respectively.

An instantiation of the forkcipher is ForkSkinny [ALP+19] which is based on tweakable block cipher Skinny [BJK+16]. Fig. 3 illustrates the generic encryption process of a ForkSkinny. The round function R of ForkSkinny follows the design specification of the Skinny and is described as

$$R = Mixcolumn \circ Addconstant \circ Addroundtweakey \circ Shiftrow \circ Subcell$$

where each of these operations (apart from the *Addconstant*) along with the tweakey schedule (TKS) are identical to the ones defined for Skinny. The only difference is in the *Addconstant* operation where 7-bit constants are used (in Skinny, 6-bit constants are used). In the context of the current work, the details pertaining to the constants and TKS are omitted. Several variants of ForkSkinny are proposed which are listed in Table 1.

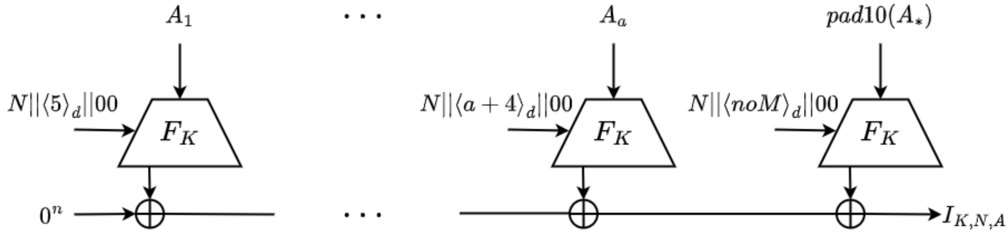


Figure 4: Associated Data Processing in Umbreon, Jolteon and Espeon. The function $pad10$ adds padding bits to the last incomplete block. Note that, $d = t - |N| - 2$ where t is the tweak size of the underlying forkcipher primitive.

Table 1: Variants of ForkSkinny

Primitive	Block size	Tweak size	Key Size	r_{init}	r_{final}
ForkSkinny-64-192	64	64	128	17	23
ForkSkinny-128-192	128	64	128	21	27
ForkSkinny-128-256	128	128	128	21	27
ForkSkinny-128-288	128	160	128	25	31
ForkSkinny-128-384	128	256	128	25	31

2.3.2 Description of PAEF, SAEF and RPAEF

PAEF (Parallel AEAD from a Forkcipher), SAEF (Sequential AEAD from a Forkcipher) and RPAEF (Reduced Parallel AEAD from a Forkcipher) were proposed in Asiacrypt 2019 [ALP+19]. PAEF provides optimal security in the nonce-respecting model and supports full parallelism, SAEF operates in a sequential manner, offering birthday-bound security and enabling low-overhead implementations, whereas RPAEF extends the PAEF by using both forkcipher output blocks only during the final call, enhancing efficiency for longer messages.

For PAEF, SAEF and RPAEF, associated data and message are partitioned into blocks of n bits where each block is processed with one call to the underlying forkcipher primitive F. In the case of PAEF, the tweak of F is composed of (i) ν bits of nonce where $0 < \nu \leq t - 4$, (ii) a three bit flag $f_0||f_1||f_2$ and (iii) $(t - \nu - 3)$ bit encoding of the block index (the encoding is maintained separately for associated data and message blocks). $f_0 = 1$ if a message block is being processed, $f_1 = 1$, if the last processed block (either message or associated data) is incomplete and $f_2 = 1$ for the last block of both message and associated data. RPAEF is a derivative of PAEF that mostly depends on the left output block of the underlying forkcipher. The tweaks for the underlying forkcipher calls are generated in the same way as PAEF, the only difference being the size of tweaks is increased by padding 0's. The processing of message and the generation of tags for SAEF, PAEF, RPAEF are outlined in Algorithm 1, Algorithm 2 and Algorithm 3, respectively.

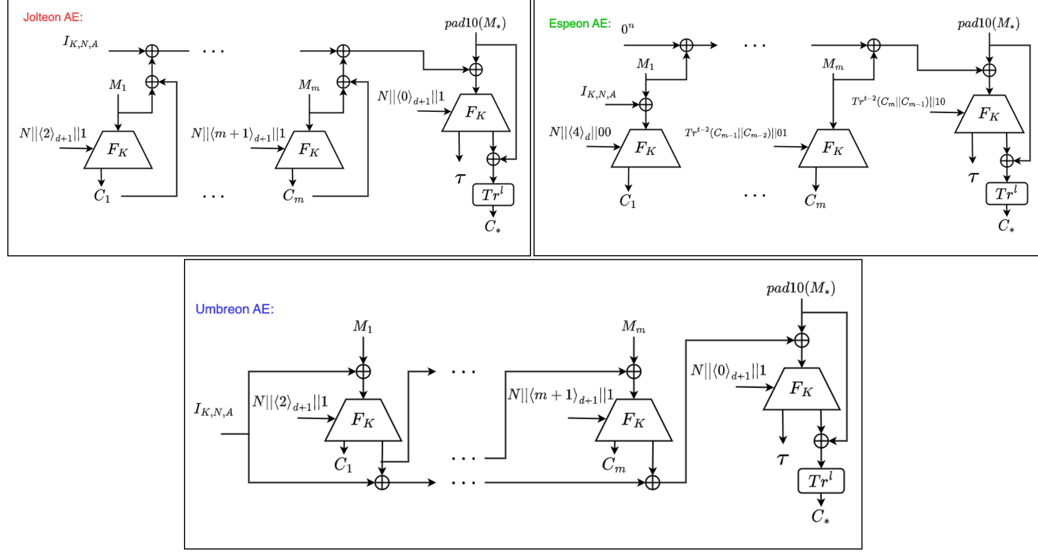


Figure 5: Processing of message and generation of tag in the Eevee family of AEAD modes. If the last message block is complete, then $N||\langle 1\rangle_{d+1}||1$, $Tr^{t-2}(C_m||C_{m-1})||11$ and $N||\langle 1\rangle_{d+1}||1$ is used as a tweak for Jolteon, Espeon and Umbreon, respectively, instead of the ones shown in the figure. Note that, $|M_*| = l$ and $d = t - |N| - 2$ where t is the tweak size of the underlying forkcipher primitive. In the case of Espeon, $C_0 = N||0^{n-|N|}$.

Algorithm 1: Encryption Algorithm of SAEF

Input: A key K, nonce N associated data A and message M
Output: A ciphertext C and a tag τ

1. $A_1 A_2 \cdots A_a A_* \xleftarrow{n} A$
2. $M_1 M_2 \cdots M_a M_* \xleftarrow{n} M$
3. $noM \leftarrow 0$
4. **if** $|M| = 0$ **then**
5. $\quad noM \leftarrow 1$
6. $\Delta \leftarrow 0^n$; $T \leftarrow N||0^{t-4-\nu}||1$
7. **for** $i = 1$ **to** a **do**
8. $\quad T \leftarrow T||000$
9. $\quad \Delta \leftarrow F_K^{T,0}(A_i \oplus \Delta)$
10. $\quad T \leftarrow 0^{t-3}$
11. **if** $|A_*| = n$ **then**
12. $\quad T \leftarrow T||noM||10$
13. $\quad \Delta \leftarrow F_K^{T,0}(A_* \oplus \Delta)$
14. $\quad T \leftarrow 0^{t-3}$
15. **else if** $|A_*| > 0$ *or* $|M| = 0$ **then**
16. $\quad T \leftarrow T||noM||11$
17. $\quad \Delta \leftarrow F_K^{T,0}((A_*||10^*) \oplus \Delta)$
18. $\quad T \leftarrow 0^{t-3}$
19. **for** $i = 1$ **to** m **do**
20. $\quad T \leftarrow T||001$
21. $\quad C_i, \Delta \leftarrow F_K^{T,b}(M_i \oplus \Delta) \oplus (\Delta, 0^n)$
22. $\quad T \leftarrow 0^{t-3}$
23. **if** $|M_*| = n$ **then**
24. $\quad T \leftarrow T||100$
25. **else if** $|M_*| > 0$ **then**
26. $\quad T \leftarrow T||101$
27. **else**
28. $\quad$ **return** Δ
29. $C_*, T \leftarrow F_K^{T,b}(pad10(M_*)) \oplus (\Delta, 0^n)$
30. $C = C_1|| \cdots ||C_m||C_*$
31. $\tau = Tr^{|M_*|}(T)$
32. **return** $C||\tau$;

2.3.3 Description of Eevee

Eevee is a provably secure family of lightweight authenticated encryption with associated data (AEAD) modes proposed for *IoT-to-cloud* secure computation [BPA+23]. Three forkcipher-based AEAD modes- Umbreon, Jolteon and Espeon constitute the Eevee family.

AEAD Modes Jolteon, Umbreon and Espeon. The three AEAD modes, first of all, processes the associated data (AD) A. A is sub-divided into blocks of n bits and

Algorithm 2: Encryption algorithm of PAEF

```
Input: Key K, nonce N, associated data A, message M
Output: Ciphertext C and tag τ
1  A_1 A_2 ··· A_a A_* <-n- A
2  M_1 M_2 ··· M_a M_* <-n- M
3  S ← 0^n; c ← (t − ν − 3)
4  for i = 1 to a do
5      T ← N||000||⟨i⟩_c
6      S ← S ⊕ F_K^{T,0}(A_i)
7  if |A_*| = n then
8      T ← N||001||⟨a + 1⟩_c
9      S ← S ⊕ F_K^{T,0}(A_*)
10 else if |A_*| > 0 or |M| = 0 then
11     T ← N||011||⟨a + 1⟩_c
12     S ← S ⊕ F_K^{T,0}(A_*||0)
13 for i = 1 to m do
14     N||000||⟨i⟩_c
15     C_i, S' ← F_K^{T,b}(M_i)
16     S ← S ⊕ S'
17 if |M_*| = n then
18     T ← N||101||⟨m + 1⟩_c
19 else if |M_*| > 0 then
20     T ← N||011||⟨a + 1⟩_c
21 else
22     return S
23 C_*, T ← F_K^{T,b}(pad10(M_*))
24 C_* ← C_* ⊕ S
25 C = C_1|| ··· ||C_m||C_*
26 τ = Tr^{|M_*|}(T)
27 return C||τ;
```

Algorithm 3: Encryption algorithm of RPAEF

```
Input: Key K, nonce N, associated data A, message M
Output: Ciphertext C and tag τ
1  A_1 A_2 ··· A_a A_* <-n- A
2  M_1 M_2 ··· M_a M_* <-n- M
3  S ← 0^n; c ← (t − ν − 3)
4  for i = 1 to a do
5      T ← N||000||⟨i⟩_c||0^n
6      S ← S ⊕ F_K^{T,0}(A_i)
7  if |A_*| = n then
8      T ← N||001||⟨a + 1⟩_c||0^n
9      S ← S ⊕ F_K^{T,0}(A_*)
10 else if |A_*| > 0 or |M| = 0 then
11     T ← N||011||⟨a + 1⟩_c||0^n
12     S ← S ⊕ F_K^{T,0}(A_*||0)
13 for i = 1 to m do
14     N||000||⟨i⟩_c||0^n
15     C_i ← F_K^{T,0}(M_i)
16     S ← S ⊕ M_i
17 if |M_*| = n then
18     T ← N||101||⟨m + 1⟩_c||S
19 else if |M_*| > 0 then
20     T ← N||011||⟨a + 1⟩_c||S
21 else
22     return S
23 C_*, T ← F_K^{T,b}(pad10(M_*))
24 C = C_1|| ··· ||C_m||C_*
25 τ = Tr^{|M_*|}(T)
26 return C||τ;
```

Figure 6: Encryption Algorithms for PAEF (left) and RPAEF (right)

the last incomplete block is padded with 10^*. The AD processing part of all the three modes are similar and is illustrated in Fig. 4. Note that, if no padding bits are required (i. e. the last block of A is complete), then in the processing of the last associated data block $N||\langle 2 + noM\rangle_d||00$ is used as a tweak instead of $N||\langle noM\rangle_d||00$. Subsequently, the processing of message and the generation of tags for the three AEAD modes are depicted in Fig. 5.

In the encryption process, both Jolteon and Espeon optimize performance by utilizing only one branch of the forkcipher up to the final processed message block. Jolteon further enhances performance by allowing parallelization of forkcipher evaluations during encryption, albeit with a trade-off of reduced security. Conversely, Umbreon and Espeon operate sequentially and utilize either both branches of the forkcipher throughout (Umbreon) or longer tweaks (Espeon), which provides additional security benefits. In the context of the current work, details pertaining to the decryption are omitted.

3 Analyzing Key-commitment Weaknesses

Now, we discuss about the specific weaknesses against key-commitment of some forkcipher-based AEAD modes. First, we give a brief overview of the strategy which is used to mount

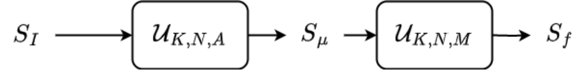


Figure 7: Generalized view corresponding to the processing of associated data and message

key-committing attacks. Then, the specific attacks on the three AEAD modes of Eevee are discussed. Finally, we also show the application of the devised strategy on PAEF and SAEF.

3.1 Overview of the Strategy

Here, first, we provide a generalized view of the processing of the nonce, associated data, and message along with the secret key. We refer the readers to Fig. 7 for this generalized view. Initially, consider an initial state S_I which is essentially the zero state. Now, due to the processing of associated data using the nonce and key, which can be represented as some transformation $\mathcal{U}_{K,N,A}$, the intermediate state S_μ is generated. Thereafter, processing the message using the nonce and the secret key, the final state S_f is reached, which is essentially the ciphertext-tag pair.

In this analysis, we focus on attacking the FROB security of AEADs in the Eevee family. Our goal is to produce the same ciphertext-tag pair using (K^1, N^1, A^1, M^1) and (K^2, N^2, A^2, M^2), where $K^1 \neq K^2$ and $N^1 = N^2$. Here, K^1 and K^2 are the keys, N^1 and N^2 are the nonces, A^1 and A^2 are associated data, and M^1 and M^2 are the messages. Given that (K^1, N^1, A^1, M^1) produces $C||\tau$, where C and τ are the ciphertext and tag, respectively, we aim to find a (K^2, A^2, M^2) such that (K^2, N^1, A^2, M^2) produces the same $C||\tau$.

To achieve this, we first choose a K^2 such that $K^1 \neq K^2$. Then, we fix the ciphertext as C and the tag as τ, and attempt to find a message M^2 using the key K^2 and the nonce N^1. However, due to the nature of the AEAD algorithm, the intermediate states S_{μ^1} and S_{μ^2} will not be equal. Here, S_{μ^1} and S_{μ^2} represent the intermediate states when (K^1, N^1, A^1, M^1) and (K^2, N^2, A^2, M^2) are used as inputs, respectively. To reach the initial state S_I, we must find a suitable A^2 that allows us to transition from μ^2 to S_I. In the following attacks, it is considered that the underlying forkcipher primitives encrypt a n-bit message and produce $2n$-bit ciphertext. Referring to Fig. 3, it is important to note that due to the forkcipher's reliance on an invertible round function R, knowledge of either M, C_l, or C_r, along with the key and tweak, is sufficient to determine the other two.

Comparison with generic attack. In the current context, an attack is considered valid if its complexity is lower than the generic attack complexity, which, for these schemes, depends solely on the tag length. Successfully forging a valid tag is enough to compromise the key-committing security, as it renders the detection of an incorrect key impossible. For an AEAD scheme with a t-bit tag, the data complexity of a generic attack is $2^{t/2}$. Therefore, any attack that recovers a valid (K^2, A^2, M^2) with a data complexity lower than $2^{t/2}$ can be considered valid.

3.2 Key-commiting Attacks on Jolteon, Espeon and Umbreon

Here, we give details of finding the same tag and ciphertext using two different sets of keys, nonces, associated data and plaintext. As stated earlier, consider that (K^1, N^1, A^1, M^1) generates $C||\tau$. Initially, assume that only one block of message is encrypted, i. e. $|M^1| = n$. Now, we will show the procedure to find a (K^2, A^2, M^2) such that (K^2, N^1, A^2, M^2) generates the same $C||\tau$.

3.2.1 Finding (K^2, A^2, M^2) for Jolteon.

We refer the readers to Fig. 8 for the attack on Jolteon. Fix the permutation for key K^2 and tweak $N||\langle 0\rangle_{d+1}||1$. Now, using the notion of reconstruction query and decryption (where τ is used as the query) $C^{'}$ and $M^{'}$ can be determined, respectively. Compute M^2 by taking XOR of $C^{'}$ and C i.e. $M^2 = C^{'} \oplus C$.

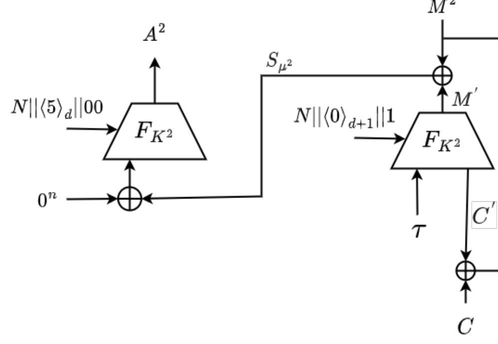


Figure 8: Key Committing Attack on Jolteon with 128-bit Message

To find A^2, we need to compute the intermediate state S_{μ^2} as $M^2 \oplus M^{'}$. Now, S_{μ^2} is decrypted using the permutation constructed from key K^2 and the tweak $N||\langle 5\rangle_d||1$. Thus, A^2 can be recovered. As it is evident, the attack can be mounted deterministically.

Extending to Messages of Arbitrary Length. Consider the scenario when $C = C_1||\cdots||C_m||C_*$ where $C_1, \cdots, C_m$ are n-bit blocks and C_* contains arbitrary number of bits between 1 to n. Thus $M^1 = M_1^1||\cdots||M_m^1||M_*^1$ and we need to find a M^2 such that $|M^2| = |M^1|$.

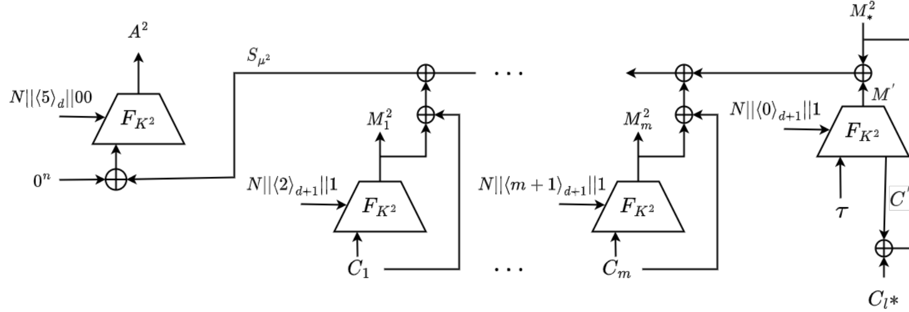


Figure 9: Key Committing Attack on Jolteon with Arbitrary Message Size

We refer the readers to Fig. 9 for the illustration of the attack. The last block M_*^2 can be recovered in the similar way as done in the previous case (attack for n-bit ciphertext). However, as the last block can be incomplete, we need to find M_*^2 such that $|M_*^2| = |C_*|$. Let $C' = F^R_{K^2,N||\langle 0\rangle_{d+1}||1}(\tau)=c'_0c'_1\cdots c'_n$ where each $c'_i \in \{0,1\}$ (for $0 \leq i \leq n$). Consider $|C_*| = l$ and $C_* = c_0^*c_1^*\cdots c_{l-1}^*$ where each $c_i^* \in \{0,1\}$ (for $0 \leq i \leq l-1$). Then a C_{l^*} is constructed such that $C_{l^*} = c_0^*c_1^*\cdots c_{l-1}^*\overline{c'_l}c'_{l+1}\cdots c'_n$. This ensures that $|M_*^2| = |M_*^1|$. The subsequent n-bit message blocks $M_1^2, \cdots, M_m^2$ can be computed as $M_i^2 = F^{-1}_{K^2,N||\langle i+1\rangle_{d+1}||1}(C_i)$. The intermediate state $S_{\mu^2} = \bigoplus_{i=1}^{m}(C_i \oplus M_i^2) \oplus (pad10(M_*^2) \oplus M')$

where $M' = F^{-1}_{K^2,N||\langle 0\rangle_{d+1}||1}(\tau)$. Then A^2 is computed as $A^2 = F^{-1}_{K^2,N||\langle 5\rangle_d||00}(S_{\mu^2})$. This new set (K^2, N, A^2, M^2) generates exactly the same ciphertext as (K^1, N, A^1, M^1). The attack strategy is outlined in Algorithm 4.

Algorithm 4: Key Committing Attack on Jolteon

Input: A key K^1, nonce N and ciphertext-tag pair $C||\tau$
Output: A key K^2, associated data A^2, message M^2 such that (K^2, N, A^2, M^2) generates the same $C||\tau$

1. $C_1C_2\cdots C_mC_* \xleftarrow{n} C$
2. $K^2 \xleftarrow{\$} \{0,1\}^\kappa$ where $\kappa = |K^1|$
3. $C' \leftarrow F^R_{K^2,N||\langle 0\rangle_{d+1}||1}(\tau)$
4. $M' \leftarrow F^{-1}_{K^2,N||\langle 0\rangle_{d+1}||1}(\tau)$
5. $c'_0c'_1\cdots c'_n \xleftarrow{1} C'$
6. $c^*_0c^*_1\cdots c^*_{l-1} \xleftarrow{1} C_*$ where $|C_*| = l$
7. $C_{l^*} \leftarrow c^*_0c^*_1\cdots c^*_{l-1}\overline{c'_l}c'_{l+1}\cdots c'_n$
8. **for** $i = 1$ **to** m **do**
9. $\quad M^2_i \leftarrow F^{-1}_{K^2,N||\langle i+1\rangle_{d+1}||1}(C_i)$
10. $M^2_* \leftarrow Tr^l(C' \oplus C_{l^*} \oplus M')$
11. $S_{\mu^2} \leftarrow \bigoplus_{i=1}^{m}(C_i \oplus M_i) \oplus (C' \oplus C_{l^*} \oplus M')$
12. $A^2 \leftarrow F^{-1}_{K^2,N||\langle 5\rangle_d||00}(S_{\mu^2})$
13. **return** K^2, A^2 and M^2;

3.2.2 Finding (K^2, A^2, M^2) for Espeon and Umbreon.

By following the similar strategy as the one for Jolteon, K^2, A^2 and M^2 can be recovered for both Espeon and Umbreon. We refer the readers to Fig. 10 and Fig. 11 for the attack on Espeon and Umbreon, respectively.

Let, $C = C_1||\cdots||C_m||C_*$ where $|C_1| = \cdots = |C_m| = n$ and $1 \leq |C_*| \leq n$. For Espeon, we compute $C' = F^R_{K^2,Trim^{t-2}(C_m||C_{m-1})||10}(\tau)$ whereas for Umbreon $C' = F^R_{K^2,N||\langle 0\rangle_{d+1}||1}(\tau)$. Consider $C' = c'_0\cdots c'_{n-1}$ and $C_* = c^*_0\cdots c^*_{l-1}$. We construct C_{l^*} as $c^*_0\cdots c^*_{l-1}\overline{c'_l}c'_{l+1}\cdots c'_{n-1}$.

For Espeon, each M^2_i (for $2 \leq i \leq m$) can be independently computed from C_i, C_{i-1} and C_{i-2}. The n-bit $M^2_1 = \bigoplus_{i=2}^{m} M^2_i \oplus (C_{l^*} \oplus C' \oplus M')$ where $M' = F^{-1}_{K^2,N||\langle 0\rangle_{d+1}||1}(\tau)$. The intermediate state $S_{\mu^2} = M^2_1 \oplus F^{-1}_{K^2,N||\langle 4\rangle_d||00}(C_1)$.

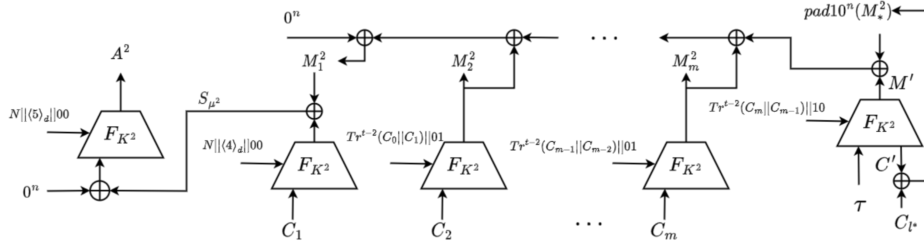


Figure 10: Key Committing Attack on Espeon

Algorithm 5: Key Committing Attack on Espeon

Input: A key K^1, nonce N and ciphertext-tag pair $C||\tau$
Output: A key K^2, associated data A^2, message M^2 such that (K^2, N, A^2, M^2) generates the same $C||\tau$

1 $C_1 C_2 \cdots C_m C_* \xleftarrow{n} C$
2 $K^2 \xleftarrow{\$} \{0,1\}^\kappa$ where $\kappa = |K^1|$ and $K^2 \neq K^1$
3 $C' \leftarrow F^R_{K^2, Trim^{t-2}(C_m||C_{m-1})||10}(\tau)$
4 $M' \leftarrow F^{-1}_{K^2, Trim^{t-2}(C_m||C_{m-1})||10}(\tau)$
5 $c'_0 c'_1 \cdots c'_n \xleftarrow{1} C'$
6 $c^*_0 c^*_1 \cdots c^*_{l-1} \xleftarrow{1} C_*$ where $|C_*| = l$
7 $C_{l^*} \leftarrow c^*_0 c^*_1 \cdots c^*_{l-1} \overline{c'_l} c'_{l+1} \cdots c'_n$
8 $C_0 \leftarrow N||0^{n-|N|}$
9 **for** $i = 2$ **to** m **do**
10 $\quad M^2_i \leftarrow F^{-1}_{K^2, Trim^{t-2}(C_i||C_{i-1})||01}(C_i)$
11 $M^2_1 \leftarrow \bigoplus_{i=2}^{m} M^2_i \oplus (C' \oplus C_{l^*} \oplus M')$
12 $M^2_* \leftarrow Tr^l(C' \oplus C_{l^*} \oplus M')$
13 $S_{\mu^2} \leftarrow M^2_i \oplus F^{-1}_{K^2, N||\langle 4 \rangle_d||00}(C_1)$
14 $A^2 \leftarrow F^{-1}_{K^2, N||\langle 5 \rangle_d||00}(S_{\mu^2})$
15 **return** K^2, A^2 and M^2;

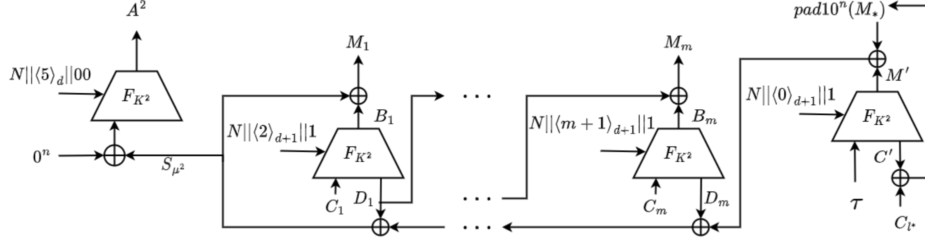


Figure 11: Key Committing Attack on Umbreon

In the case Umbreon, the M^2_i's can not be computed independently. First, D_i and B_i (for $1 \leq i \leq m$) are computed where $D_i = F^R_{K_2, N||\langle i+1 \rangle_{d+1}||1}(C_i)$ and $B_i = F^{-1}_{K_2, N||\langle i+1 \rangle_{d+1}||1}(C_i)$. Then, the values of M^2_2, M^2_3, $\cdots$, M^2_m can be fixed by taking the XOR of corresponding D_{i-1}'s and B_i. The intermediate state $S_{\mu^2} = \bigoplus_{i=1}^{m} D_i \oplus C_{l^*} \oplus C' \oplus M'$ where $M' = F^{-1}_{K^2, N||\langle 0 \rangle_{d+1}||1}(\tau)$.

For recovering A^2, the strategy similar to the one used for Jolteon is followed. Primarily, finding a n-bit A^2 is sufficient for mounting the attack. Thus the value of A^2 is determined by decrypting S_{μ^2} using the key K^2 and the tweak $N||\langle 5 \rangle_d||00$. The attack on Espeon and Umbreon are outlined in Algorithm 5 and Algorithm 6, respectively.

3.3 Key-committing Attacks on PAEF, and SAEF

Here, we discuss about the committing attacks on the forkcipher-based AEADs, PAEF and SAEF. We refer the readers to Fig. 12 for an overview of the analysis. We briefly describe the analysis on PAEF.

Algorithm 6: Key Committing Attack on Umbreon

Input: A key K^1, nonce N and ciphertext-tag pair $C||\tau$
Output: A key K^2, associated data A^2, message M^2 such that (K^2, N, A^2, M^2) generates the same $C||\tau$

1 $C_1C_2 \cdots C_mC_* \xleftarrow{n} C$
2 $K^2 \xleftarrow{\$} \{0,1\}^\kappa$ where $\kappa = |K^1|$ and $K^2 \neq K^1$
3 $C' \leftarrow F^R_{K^2,N||\langle 0\rangle_{d+1}||1}(\tau)$
4 $M' \leftarrow F^{-1}_{K^2,N||\langle 0\rangle_{d+1}||1}(\tau)$
5 $c'_0c'_1 \cdots c'_n \xleftarrow{1} C'$
6 $c^*_0c^*_1 \cdots c^*_{l-1} \xleftarrow{1} C_*$ where $|C_*| = l$
7 $C_{l^*} \leftarrow c^*_0c^*_1 \cdots c^*_{l-1}\overline{c'_l}c'_{l+1} \cdots c'_n$
8 **for** $i = 1$ **to** m **do**
9 $\quad B_i \leftarrow F^{-1}_{K^2,N||\langle i+1\rangle_{d+1}||1}(C_i)$
10 $\quad D_i \leftarrow F^R_{K^2,N||\langle i+1\rangle_{d+1}||1}(C_i)$
11 **for** $i = 2$ **to** m **do**
12 $\quad M^2_i \leftarrow D_{i-1} \oplus B_i$
13 $M^2_* \leftarrow Tr^l(C' \oplus C_{l^*} \oplus M')$
14 $S_{\mu^2} \leftarrow \bigoplus_{i=1}^{m} D_i \oplus (C' \oplus C_{l^*} \oplus M')$
15 $M^2_1 \leftarrow S_{\mu^2} \oplus B_1$
16 $A^2 \leftarrow F^{-1}_{K^2,N||\langle 5\rangle_d||00}(S_{\mu^2})$
17 **return** K^2, A^2 and M^2;

Suppose that a ciphertext-tag pair $C||\tau$ is obtained by querying the PAEF oracle (K^1, N, A^1, M^1) where K^1, N, A^1 and M^1 are the key, nonce, associated data and message, respectively and $C = C_1|| \cdots ||C_m||C_*$. Now, we need to determine a key K^2, associated data A^2 and message M^2 such that when (K^2, N, A^2, M^2) is queried the same ciphertext-tag $C||\tau$ is generated.

First, we use τ to make a reconstruction query using the key K^2 to obtain C' and M^2_*. Similarly, each C_i $(1 \leq i \leq m)$ is used to make a reconstruction query to obtain M^2_i and D_i. $\bigoplus_{i=1}^m D_i \oplus C' \oplus C_*$ is used to obtain the respective A^2. In a similar way, we can find a set (K^2, N, A^2, M^2) for SAEF also. However, unlike PAEF in which each M^2_i can be recovered in parallel, in the case SAEF each block of message are recovered in a sequential manner. The attack algorithms for PAEF and SAEF are outlined in Algorithm 7 and Algorithm 8, respectively. As evident from the algorithms, the attacks can be mounted in $O(1)$ complexity.

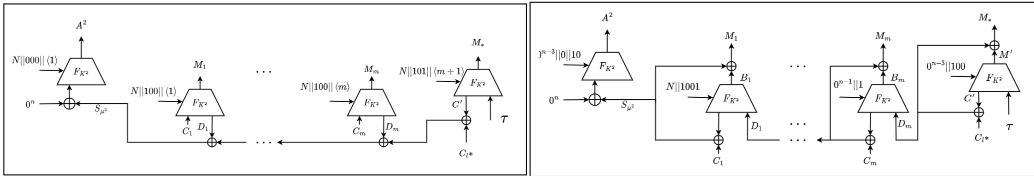
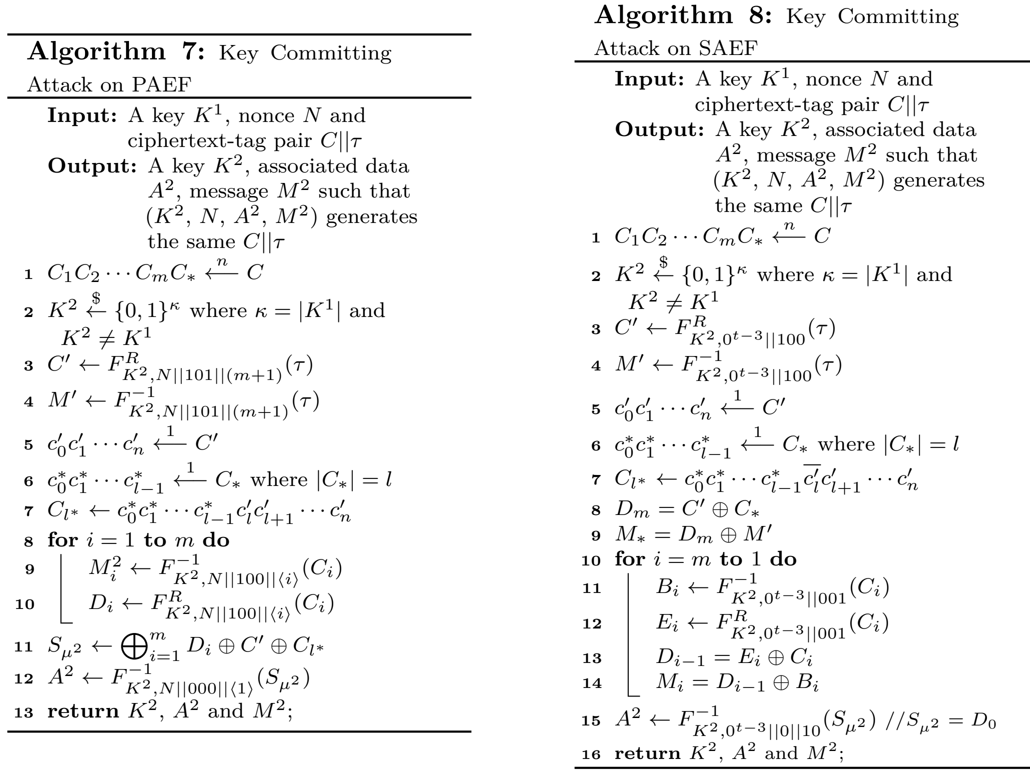


Figure 12: Key Committing Attack on PAEF and SAEF

Note that using a similar strategy, key-committing attacks cannot be mounted on RPAEF. We refer the readers to Step 16 in Algorithm 3. In RPAEF, each message block M_i $(1 \leq i \leq m)$ is XORed with S before this S is introduced into the tweak T. This tweak is then used to encrypt the final complete/incomplete message block M_*. Consequently, in



Algorithm 7: Key Committing Attack on PAEF

Input: A key K^1, nonce N and ciphertext-tag pair $C||\tau$

Output: A key K^2, associated data A^2, message M^2 such that (K^2, N, A^2, M^2) generates the same $C||\tau$

1 $C_1C_2\cdots C_mC_* \xleftarrow{n} C$

2 $K^2 \xleftarrow{\$} \{0,1\}^\kappa$ where $\kappa = |K^1|$ and $K^2 \neq K^1$

3 $C' \leftarrow F^R_{K^2,N||101||(m+1)}(\tau)$

4 $M' \leftarrow F^{-1}_{K^2,N||101||(m+1)}(\tau)$

5 $c'_0c'_1\cdots c'_n \xleftarrow{1} C'$

6 $c^*_0c^*_1\cdots c^*_{l-1} \xleftarrow{1} C_*$ where $|C_*| = l$

7 $C_{l*} \leftarrow c^*_0c^*_1\cdots c^*_{l-1}c'_lc'_{l+1}\cdots c'_n$

8 **for** $i = 1$ **to** m **do**

9 $\quad M^2_i \leftarrow F^{-1}_{K^2,N||100||\langle i\rangle}(C_i)$

10 $\quad D_i \leftarrow F^R_{K^2,N||100||\langle i\rangle}(C_i)$

11 $S_{\mu^2} \leftarrow \bigoplus_{i=1}^m D_i \oplus C' \oplus C_{l*}$

12 $A^2 \leftarrow F^{-1}_{K^2,N||000||\langle 1\rangle}(S_{\mu^2})$

13 **return** K^2, A^2 and M^2;

Algorithm 8: Key Committing Attack on SAEF

Input: A key K^1, nonce N and ciphertext-tag pair $C||\tau$

Output: A key K^2, associated data A^2, message M^2 such that (K^2, N, A^2, M^2) generates the same $C||\tau$

1 $C_1C_2\cdots C_mC_* \xleftarrow{n} C$

2 $K^2 \xleftarrow{\$} \{0,1\}^\kappa$ where $\kappa = |K^1|$ and $K^2 \neq K^1$

3 $C' \leftarrow F^R_{K^2,0^{t-3}||100}(\tau)$

4 $M' \leftarrow F^{-1}_{K^2,0^{t-3}||100}(\tau)$

5 $c'_0c'_1\cdots c'_n \xleftarrow{1} C'$

6 $c^*_0c^*_1\cdots c^*_{l-1} \xleftarrow{1} C_*$ where $|C_*| = l$

7 $C_{l*} \leftarrow c^*_0c^*_1\cdots c^*_{l-1}\overline{c'_l}c'_{l+1}\cdots c'_n$

8 $D_m = C' \oplus C_*$

9 $M_* = D_m \oplus M'$

10 **for** $i = m$ **to** 1 **do**

11 $\quad B_i \leftarrow F^{-1}_{K^2,0^{t-3}||001}(C_i)$

12 $\quad E_i \leftarrow F^R_{K^2,0^{t-3}||001}(C_i)$

13 $\quad D_{i-1} = E_i \oplus C_i$

14 $\quad M_i = D_{i-1} \oplus B_i$

15 $A^2 \leftarrow F^{-1}_{K^2,0^{t-3}||0||10}(S_{\mu^2})$ $//S_{\mu^2} = D_0$

16 **return** K^2, A^2 and M^2;

Figure 13: Key Committing Attack on PAEF (left) and SAEF (right)

a key-committing setting, when attempting to find the same tag and ciphertext using a different key, a collision on C_* is required. This effectively raises the attack complexity to that of a generic forgery attack.

4 Countermeasures and Discussions

To transform these AEAD schemes into commitment-secure AEAD, one straightforward approach is to apply an existing algorithm such as CTX [CR22] or CTY [BH24], as none of these AEAD schemes currently have tag-dependent encryption or decryption algorithms. For such a transformation, we would need an additional universal hash function.

However, when attempting a dedicated approach to make these AEAD schemes commitment-secure (CMT-secure), several challenges arise. Since the adversary has access to ideal cipher queries, they can exploit this access to compute the AEAD construction by combining

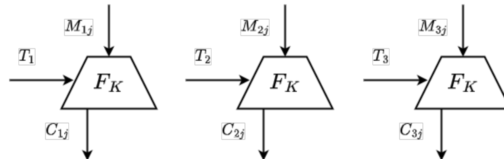


Figure 14: Primitive calls on Forkcipher, where (M_{ij}, T_i) are the message tag pairs and C_{ij} are the ciphertexts for $i \in \{1, 2, 3\}$ and $j \in \{1, 2\}$

several ideal cipher queries. Specifically, an adversary making $\mathcal{O}(p)$ ideal cipher queries can generate $\mathcal{O}(p^l)$ construction queries, where l is the maximum message length. For example, as shown in Fig. 14, adversary makes M_{ij} as primitive query with tweak T_i for $i \in \{1,2,3\}$ and $j \in \{1,2\}$. So, there are such 6 many primitive queries. But using these he can construct $2^3 = 8$ many construction queries as $M_{1i}\|M_{2j}\|M_{3k}$ for $i,j,k \in \{1,2\}$. So, if the number of primitive queries increases linearly, the number of construction queries from these primitive queries increases exponentially.

In these forkcipher based AEAD, the computation of each associated data block is performed independently of the others. This independence, while contributing to efficiency and parallelism, creates a vulnerability that adversaries can exploit. Specifically, the adversary can manipulate the associated data to achieve any desired intermediate state $I_{K,N,A}$ (corresponding to Fig. 4). Even if we add some checksum or include the right output of the forkcipher in the associated data processing, it will not prevent this issue. The adversary can still produce associated data for every 2^n value of $I_{K,N,A}$ with only $\mathcal{O}(n)$ queries.

Also, in the message processing part, in all these schemes, the computation of each block is independent of other messages or associated data blocks. In a forkcipher, message can be reconstructed from either of the ciphertexts and one ciphertext block can be reconstructed from the other. Using these properties of the forkcipher and by making inverse calls to the forkcipher the adversary can easily find primitive queries for any desired ciphertext, except for the last block. For the last ciphertext block, the adversary can always choose a suitable message block to obtain the desired ciphertext.

Therefore, it is difficult to transform these schemes in to commitment-secure AEAD introducing significant overheads which includes extra calls to either a hash function or a pseudo-random function. The generic countermeasures based on hash functions [CR22] or pseudo-random functions [ADG+22, BH22] can be employed to make these schemes key-committing. For example, in [CR22], a key K, nonce N, associated data A, and a message M are first encrypted using an AEAD scheme to generate a ciphertext-tag pair $C\|T$. Then, a hash function is applied to K, N, A, and T to generate T^*. Finally, $C\|T^*$ is communicated instead of $C\|T$. This scheme is proven to be a committing one; however, it requires an additional call to a hash function. Similarly, other schemes in [ADG+22, BH22] require additional calls to pseudo-random functions.

5 Conclusion

Our investigation into the key-committing security of forkcipher-based AEAD modes has revealed significant vulnerabilities. By demonstrating a key-committing attack within the FROB game framework, we have shown that an adversary can exploit the processing of associated data and plaintext to generate tag collisions with a complexity of $O(1)$. This attack is effective even in less strict frameworks such as CMT-1 and CMT-4, highlighting the importance of ensuring robustness in AEAD schemes. Based on our findings, it is recommended that unless explicit robustness is required, the use of such forkcipher-based AEAD modes should be carefully evaluated.

Acknowledgments

The authors are thankful to the anonymous reviewers of IACR CiC for their valuable comments and suggestions. Mostafizar and Samir would like to thank Sougata Mandal for his valuable suggestions regarding the Section 4 of the paper. The results are obtained from the commissioned research (JPJ012368C05801) by the National Institute of Information and Communications Technology (NICT), Japan. This work is also supported by JST

AIP Acceleration Research JPMJCR24U1 Japan and JSPS KAKENHI Grant Number JP24H00696.

References

[ABPV21] Elena Andreeva, Amit Singh Bhati, Bart Preneel, and Damian Vizár. 1, 2, 3, fork: Counter mode variants based on a generalized forkcipher. *IACR Trans. Symmetric Cryptol.*, 2021(3):1–35, 2021. doi:10.46586/TOSC.V2021.I3.1-35.

[ADG+22] Ange Albertini, Thai Duong, Shay Gueron, Stefan Kölbl, Atul Luykx, and Sophie Schmieg. How to Abuse and Fix Authenticated Encryption Without Key Commitment. In Kevin R. B. Butler and Kurt Thomas, editors, *31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022*, pages 3291–3308. USENIX Association, 2022.

[ALP+19] 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 *Advances in Cryptology - ASIACRYPT 2019 - 25th International Conference on the Theory and Application of Cryptology and Information Security, Kobe, Japan, December 8-12, 2019, Proceedings, Part II*, volume 11922 of *Lecture Notes in Computer Science*, pages 153–182. Springer, 2019. doi:10.1007/978-3-030-34621-8_6.

[ARVV18] Elena Andreeva, Reza Reyhanitabar, Kerem Varici, and Damian Vizár. Forking a blockcipher for authenticated encryption of very short messages. *IACR Cryptol. ePrint Arch.*, page 916, 2018. URL: https://eprint.iacr.org/2018/916.

[AW23] Elena Andreeva and Andreas Weninger. A forkcipher-based pseudo-random number generator. In Mehdi Tibouchi and Xiaofeng Wang, editors, *Applied Cryptography and Network Security - 21st International Conference, ACNS 2023, Kyoto, Japan, June 19-22, 2023, Proceedings, Part II*, volume 13906 of *Lecture Notes in Computer Science*, pages 3–31. Springer, 2023. doi:10.1007/978-3-031-33491-7_1.

[BH22] Mihir Bellare and Viet Tung Hoang. Efficient Schemes for Committing Authenticated Encryption. In Orr Dunkelman and Stefan Dziembowski, editors, *Advances in Cryptology - EUROCRYPT 2022 - 41st Annual International Conference on the Theory and Applications of Cryptographic Techniques, Trondheim, Norway, May 30 - June 3, 2022, Proceedings, Part II*, volume 13276 of *Lecture Notes in Computer Science*, pages 845–875. Springer, 2022. doi:10.1007/978-3-031-07085-3_29.

[BH24] Mihir Bellare and Viet Tung Hoang. Succinctly-committing authenticated encryption. In Leonid Reyzin and Douglas Stebila, editors, *Advances in Cryptology - CRYPTO 2024 - 44th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2024, Proceedings, Part IV*, volume 14923 of *Lecture Notes in Computer Science*, pages 305–339. Springer, 2024. doi:10.1007/978-3-031-68385-5_10.

[BJK+16] Christof Beierle, Jérémy Jean, Stefan Kölbl, Gregor Leander, Amir Moradi, Thomas Peyrin, Yu Sasaki, Pascal Sasdrich, and Siang Meng Sim. The SKINNY family of block ciphers and its low-latency variant MANTIS. In Matthew Robshaw and Jonathan Katz, editors, *Advances in Cryptology -*

CRYPTO 2016 - 36th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2016, Proceedings, Part II, volume 9815 of *Lecture Notes in Computer Science*, pages 123–153. Springer, 2016. doi:10.1007/978-3-662-53008-5_5.

[BPA+23] Amit Singh Bhati, Erik Pohle, Aysajan Abidin, Elena Andreeva, and Bart Preneel. Let's go eevee! A friendly and suitable family of AEAD modes for iot-to-cloud secure computation. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, CCS 2023, Copenhagen, Denmark, November 26-30, 2023*, pages 2546–2560. ACM, 2023. doi:10.1145/3576915.3623091.

[CFI+23] Yu Long Chen, Antonio Flórez-Gutiérrez, Akiko Inoue, Ryoma Ito, Tetsu Iwata, Kazuhiko Minematsu, Nicky Mouha, Yusuke Naito, Ferdinand Sibleyras, and Yosuke Todo. Key committing security of AEZ and more. *IACR Trans. Symmetric Cryptol.*, 2023(4):452–488, 2023. URL: https://doi.org/10.46586/tosc.v2023.i4.452-488, doi:10.46586/TOSC.V2023.I4.452-488.

[CR22] John Chan and Phillip Rogaway. On Committing Authenticated-Encryption. In Vijayalakshmi Atluri, Roberto Di Pietro, Christian Damsgaard Jensen, and Weizhi Meng, editors, *Computer Security - ESORICS 2022 - 27th European Symposium on Research in Computer Security, Copenhagen, Denmark, September 26-30, 2022, Proceedings, Part II*, volume 13555 of *Lecture Notes in Computer Science*, pages 275–294. Springer, 2022. doi:10.1007/978-3-031-17146-8_14.

[DDLM24] Nilanjan Datta, Avijit Dutta, Eik List, and Sougata Mandal. FEDT: forkcipher-based leakage-resilient beyond-birthday-secure AE. *IACR Commun. Cryptol.*, 1(2):21, 2024. URL: https://doi.org/10.62056/akgyl86bm, doi:10.62056/AKGYL86BM.

[DFI+24] Patrick Derbez, Pierre-Alain Fouque, Takanori Isobe, Mostafizar Rahman, and André Schrottenloher. Key committing attacks against aes-based AEAD schemes. *IACR Trans. Symmetric Cryptol.*, 2024(1):135–157, 2024. URL: https://doi.org/10.46586/tosc.v2024.i1.135-157, doi:10.46586/TOSC.V2024.I1.135-157.

[DGRW18] Yevgeniy Dodis, Paul Grubbs, Thomas Ristenpart, and Joanne Woodage. Fast Message Franking: From Invisible Salamanders to Encryptment. In Hovav Shacham and Alexandra Boldyreva, editors, *Advances in Cryptology - CRYPTO 2018 - 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2018, Proceedings, Part I*, volume 10991 of *Lecture Notes in Computer Science*, pages 155–186. Springer, 2018. doi:10.1007/978-3-319-96884-1_6.

[Fac16] Facebook. *Messenger Secret Conversations technical whitepaper*. https://fbnewsroomus.files.wordpress, 2016.

[FOR17] Pooya Farshim, Claudio Orlandi, and Razvan Rosie. Security of Symmetric Primitives under Incorrect Usage of Keys. *IACR Trans. Symmetric Cryptol.*, 2017(1):449–473, 2017. doi:10.13154/tosc.v2017.i1.449-473.

[GLR17] Paul Grubbs, Jiahui Lu, and Thomas Ristenpart. Message Franking via Committing Authenticated Encryption. In Jonathan Katz and Hovav Shacham, editors, *Advances in Cryptology - CRYPTO 2017 - 37th Annual International*

Cryptology Conference, Santa Barbara, CA, USA, August 20-24, 2017, Proceedings, Part III, volume 10403 of *Lecture Notes in Computer Science*, pages 66–97. Springer, 2017. doi:10.1007/978-3-319-63697-9_3.

[JKX18] Stanislaw Jarecki, Hugo Krawczyk, and Jiayu Xu. OPAQUE: an asymmetric PAKE protocol secure against pre-computation attacks. In Jesper Buus Nielsen and Vincent Rijmen, editors, *Advances in Cryptology - EUROCRYPT 2018 - 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, April 29 - May 3, 2018 Proceedings, Part III*, volume 10822 of *Lecture Notes in Computer Science*, pages 456–486. Springer, 2018. doi:10.1007/978-3-319-78372-7_15.

[KLL20] Hwigyeom Kim, Yeongmin Lee, and Jooyoung Lee. Forking tweakable even-mansour ciphers. *IACR Trans. Symmetric Cryptol.*, 2020(4):71–87, 2020. URL: https://doi.org/10.46586/tosc.v2020.i4.71-87, doi:10.46586/TOSC.V2020.I4.71-87.

[LGR21] Julia Len, Paul Grubbs, and Thomas Ristenpart. Partitioning Oracle Attacks. In Michael Bailey and Rachel Greenstadt, editors, *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, pages 195–212. USENIX Association, 2021.

[Man24] Sougata Mandal. Tweakable forkcipher from ideal block cipher. *IACR Commun. Cryptol.*, 1(3):42, 2024. doi:10.62056/AEY4FBN2HD.

[Mil17] Jon Millican. *Challenges of E2E Encryption in Facebook Messenger*. Real World Cryptography conference, 2017.

[MLGR23] Sanketh Menda, Julia Len, Paul Grubbs, and Thomas Ristenpart. Context Discovery and Commitment Attacks - How to Break CCM, EAX, SIV, and More. In Carmit Hazay and Martijn Stam, editors, *Advances in Cryptology - EUROCRYPT 2023 - 42nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Lyon, France, April 23-27, 2023, Proceedings, Part IV*, volume 14007 of *Lecture Notes in Computer Science*, pages 379–407. Springer, 2023. doi:10.1007/978-3-031-30634-1_13.

[NSS23] Yusuke Naito, Yu Sasaki, and Takeshi Sugawara. Committing security of ascon: Cryptanalysis on primitive and proof on mode. *IACR Trans. Symmetric Cryptol.*, 2023(4):420–451, 2023. URL: https://doi.org/10.46586/tosc.v2023.i4.420-451, doi:10.46586/TOSC.V2023.I4.420-451.

A Experimental Verification and Attack Vectors

We have experimentally verified the proposed attacks. In the attack vectors, C and τ are the ciphertext and tag, respectively. K^1, K^2 are the two secret keys, N is the nonce, A^1, A^2 are two associated data and M^1, M^2 are two messages. We provide here the C, τ, K^1, K^2, N, A^1, A^2, M^1 and M^2 such that $\Sigma_{Enc}(K^1, N^1, A^1, P^1) = \Sigma_{Enc}(K^2, N^2, A^2, P^2) = C||\tau$.

We provide here the attack vectors corresponding to PAEF and Umbreon. The verification codes are available online [1] (codes in https://github.com/byt3bit/forkae/tree/master/software/ref/paefforkskinnyb128t192n48v1/ref are reused and modified). In the vectors provided, the leftmost bit is the least significant bit (LSB). Consider a 16-bit string $b_0 \cdots b_{15}$ where b_0 is the LSB and b_{15} is the most significant bit (MSB). Using the vectors, the above string is denoted as $[b_0 \cdots b_7 \quad b_8 \cdots b_{15}]$. We have used hexadecimal numbers to denote each 8-bit number.

A.1 Attack Vector for PAEF

$C=$	[0x10	0x94	0x39	0x03	0x80	0xF8	0xCB	0xBC]
$\tau=$	[0xC1	0x67	0x19	0xE5	0x6A	0x96	0x14	0x01]
$K^1=$	[0x00	0x01	0x02	0x03	0x04	0x05	0x06	0x07
	0x08	0x09	0x0A	0x0B	0x0C	0x0D	0x0E	0x0F]
$N=$	[0x00	0x01	0x02	0x03	0x04	0x05]		
$A^1=$	[0x00	0x01	0x02	0x03	0x04	0x05	0x06	0x07]
$M^1=$	[0x00	0x01	0x02	0x03	0x04	0x05	0x06	0x07]
$K^2=$	[0x01	0x11	0x12	0x13	0x14	0x15	0x16	0x17
	0x18	0x19	0x1A	0x1B	0x1C	0x1D	0x1E	0x1F]
$A^2=$	[0x41	0xC0	0xCF	0x33	0x4D	0xA8	0x06	0xD2]
$M^2=$	[0xF6	0x58	0xEF	0x48	0x72	0xC9	0xB4	0xEE]

A.2 Attack Vector for Umbreon

$C=$	[0x75	0x28	0xC8	0xC4	0xE7	0xD8	0x52	0x6A]
$\tau=$	[0x5A	0x3A	0x4A	0x3C	0x52	0x05	0xEE	0xBF]
$K^1=$	[0x00	0x01	0x02	0x03	0x04	0x05	0x06	0x07
	0x08	0x09	0x0A	0x0B	0x0C	0x0D	0x0E	0x0F]
$N=$	[0x00	0x01	0x02	0x03	0x04	0x05]		
$A^1=$	[0x00	0x01	0x02	0x03	0x04	0x05	0x06	0x07]
$M^1=$	[0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00]
$K^2=$	[0x01	0x11	0x12	0x13	0x14	0x15	0x16	0x17
	0x18	0x19	0x1A	0x1B	0x1C	0x1D	0x1E	0x1F]
$A^2=$	[0xC9	0x4B	0x01	0x6C	0xBD	0xAC	0xC5	0x76]
$M^2=$	[0x96	0x55	0x2A	0xFD	0x5C	0x84	0xA9	0xF7]

[1] https://github.com/mrahman454/ForkAE_CiC_2024_4