**BigProject in Cybersecurity**

# Cryptanalysis of ForkAES-*-2-2

*Tutor: Arnab Roy* *Student: Luca Campa*

## 1 Introduction

Andreeva et al. proposed ForkAES, a tweakable AES-based forkcipher that splits the state after five out of ten rounds. That schema was studied with different cryptanalysis techniques which brought to the construction of practical attacks against round reduced ForkAES ciphers. In this article ForkAES-*-2-2 will be deeply analyzed with the application of differential reflection trail techniques and it will be shown how to slightly reduce the complexity of the attack thanks to the exploitation of key scheduling features.

## 2 ForkAES

### 2.1 What is a forkcipher

A forkcipher is a cryptographic primitive which, starting from fixed input length, is able to output more bits than it receves as input divided into fixed length blocks. Its main aim is to provide Authenticated Encryption for smaller devices whose capabilities are not enough to support more complex algorithms (e.g. some IOT devices). ForkAES is an example of forkcipher which makes use of one of the most famous cryptographic primitives: AES. At the beginning, it was thought that the security of this schema would be inherited by the security of AES after many rounds. Unfortunately, the specification of ForkAES leaved some characteristics to be exploited that can be used to break the cipher with low complexity.
A forkcipher should implement three algorithms:

- an encryption algorithm: given a key $K$, a tweak $T$, a plaintext $P \in \{0,1\}^n$, the function will generate $2n$ bits, meaning an output whose length is double the size of the input:

$$Encr : K \times T \times \{0,1\}^n \rightarrow \{0,1\}^{2n}$$

- a decryption algorithm: given a key $K$, a tweak $T$, one of the blocks of the output and a bit indicating what of the two output blocks it is, the function will generate the corresponding plaintext:

$$Decr : K \times T \times \{0,1\}^n \times \{0,1\} \rightarrow \{0,1\}^n$$

- a tag-reconstruction algorithm: given a key $K$, a tweak $T$, one of the blocks of the output and a bit indicating what of the two output blocks it is, the function will generate the other output block:

$$TagRecon : K \times T \times \{0,1\}^n \times \{0,1\} \rightarrow \{0,1\}^n$$

### 2.2 The ForkAES specification

ForkAES is a deterministic AES-based tweakable forkcipher proposed and formalized by Andreeva et al. whose internal state is forked after half of the rounds to produce two 128-bit output blocks. The specification is shown in Figure 1. The following analysis will take into account the version with 128 bits key and both 128 and 64 bits tweak. In real application, modern IoT systems tend to use to latter one cause to the restricted resources they are supplied with.

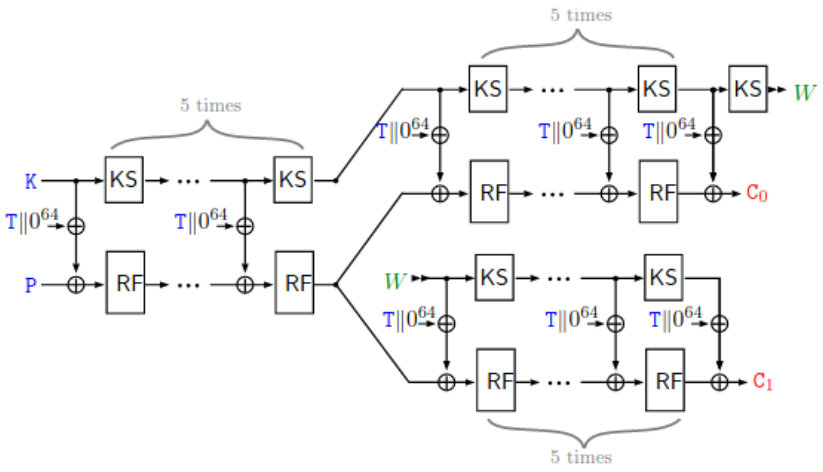


Figure 1: (taken from [1], p. 3) A 128 bit plaintext P, a 128 bit key K and 64 bit tweak T (all in blue) are used to compute a 256 bit ciphertext C = C0‖C1 (in red). RF denotes a single iteration of the AES round function and KS denotes a single iteration of the AES keyschedule.[1]

## 3 Previous Results

Although it is a recent implementation, it was studied and analysed with different techniques [2][3]. In particular, differential reflection trail was applied against `ForkAES-5-3-3` and `ForkAES-5-4-4` and yoyo tricks were applied to `ForkAES-5-3-3` with good results which ended up in the construction of very practical attacks [2].

## 4 Attack on ForkAES-*-2-2 with Reflection Trails (128 bits Tweak)

The attack[2] can work for an arbitrary number of rounds before the state is forked. Our analysis and diagrams refer to 5 rounds before the fork and 2 rounds after that, then the keys are numbered from 0 to 9.
The analysis will make use of the diagram in Figure 2 because only the two rounds after the fork will be considered[1].

[1]Notice that the values indicated with the apex in the diagrams are shown with a ˜ in the text

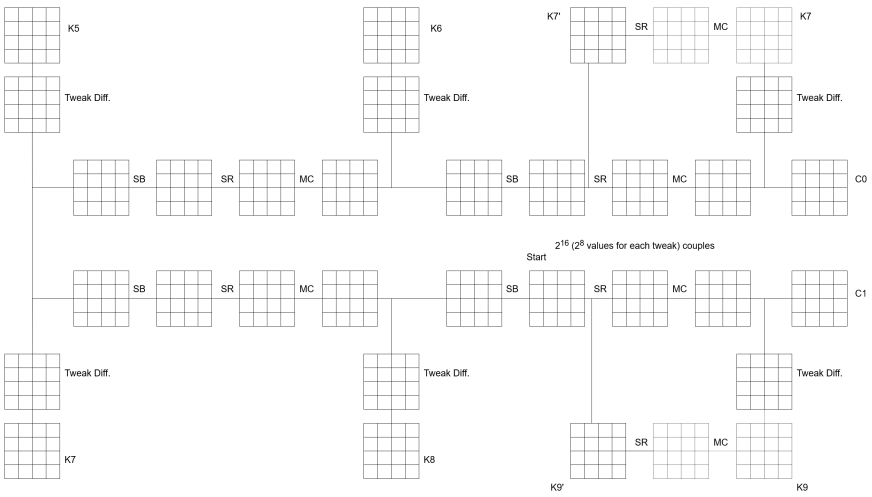


Figure 2: Considered part of the cipher

**Reflection Trails** The analysis will make use of a modified version of the Reflection property which considers the differences between the states of two different encryptions. In particular, we are searching for a differential characteristic which makes possible to find one of the keys involved in the tag-reconstruction operation.

**Proposition: Differential Reflection Trail** If there exists a differential $F_r$ for the $r$-round transformation that propagates a difference $\Delta I$ to $\Delta O$ with probability $p$, there exists a differential for the $2r$-round transformation $(F^{-1})^r \circ F^r$ that propagates a difference $\Delta I$ to $\Delta O$ with probability at least $p^2$. This property holds for any choice of round keys and constants in the $2r$ rounds[2].

We can apply Reflection trails technique on the tag-reconstruction procedure because it is a backward and forward computation of the same round function with different round-keys and constants. The addition of constants and round-keys does not interfere with the property we are going to use.

**The characteristic** Thanks to the mix column operation, a single active byte propagates to 4 active bytes in the next round and to 16 in the second one ($1 \xrightarrow{F} 4 \xrightarrow{F} 16 \xrightarrow{F} 16$). Tweak injection helps us to get better trails. Notice that, if we decide a specific tweak difference, we can find a couple of states whose difference from the fork is $1 \xrightarrow{F} 0 \xrightarrow{F} 1 \xrightarrow{F} 4$. That means that whilst the common diffusion is reduced, the probability of guessing the key increases.

**The reflection trail** From the above differential characteristic, we can build a reflection trail which enables the attacker to break the cipher efficiently. The trail is: $4 \xrightarrow{F^{-1}} 1 \xrightarrow{F^{-1}} 0 \xrightarrow{F^{-1}} 1 \xrightarrow{F} 0 \xrightarrow{F} 1 \xrightarrow{F} 4$.

**Attack** The probability that the one byte difference in $\tilde{C_1}$ becomes equal to the tweak difference after the inverse sub bytes operation in order to make it disappear is $2^8$. This means that for each byte of the key we can obtain a very little number of possibilities (1-2) by only considering $2^8 \times 2^8$ couples of ciphertexts $C_0$.
Such trail does not work with 3 or 4 rounds after the fork because it will imply that the states at the fifth round should be equal, which is slightly impossible.

**Attack Procedure for one key byte** With the following procedure we are going to find one byte of $\tilde{K_7}$.

**1)** Choose tweaks $T_0$ and $T_1$ with the chosen fixed difference $\Delta T$. This means that you will end up with $2^8$ couples $(T_0, T_1)$.

**2)** For each tweaks couple:

- For each tweak $T$ in the couple:
  - take the $2^8$ distinct values of $\tilde{C_1}$ for the same byte number in which the tweak difference is considered. Meaning that, if the tweak difference is at byte 0, we will take all the possible values of the byte 0 of $\tilde{C_1}$. Fix the other 15 bytes to constant values and compute the corresponding $C_1$
  - You will obtain $2^8$ values of $C_1$. Query each of those values to the tag-reconstruction procedure in order to obtain the corresponding $C_0$ values. For each of those $2^8$ $C_0$ values, compute the corresponding $\tilde{C_0}$.
- We obtain $2^8$ values computed with $T_0$ and $2^8$ values computed with $T_1$.
- From the $2^{16}$ couples between different tweaks, pick the one with 15 inactive bytes in the byte positions different from the one of the tweak difference. If such one couple is not found, skip to the next tweak couple, otherwise pick such couple and go to step 3.

**3)** We expect at least one right pair. Initialize a counter for each possible byte value of the key we are trying to find. For each possible byte value of $\tilde{K_7}$:

- For each value in the couple:
  - add that key byte, add the corresponding tweak ($T_0$ for the first value in the couple and $T_1$ for the second one), apply inverse mix-column and inverse shift-row.
- If there is no difference between the two computed states increase the counter of the tried key byte value.

**4)** Consider only the values whose counter value is equal to the maximum. We expect at most 2 values with a probability of the 95,4% (computed empirically).

**Full Attack** Iterate the procedure above for each key byte (the figures in Appendix show the trails for the key bytes of $\tilde{K_7}$ column 0). After having computed the possibilities for each key byte, compute all the possible keys $\tilde{K_7}$ and test them progressively. Once the possibilities are reduced to one key candidate, compute the corresponding $K_7$. From the computed $K_7$, generate all the other round keys by applying the key-scheduling and the inverse-key-scheduling algorithm.

**Complexity**

- **Time**:
  - Encryptions: we need to distinguish two cases
    - If we validate the keys by checking differential characteristic (as done in the provided code), the number of encryptions is 1.
    - If we validate the keys by exhaustive searching the right key from all the found possibilities, the number of encryptions is $\simeq 2^{16}$. Rarely this number could be slightly greater.
  - Memory Accesses: $2^{13}$ if the validation is done with exhaustive search, $2^{14}$ with the other method.
- **Data**: the attack requires $16 \times (2^8 + 2^8) = 2^{13}$ tag-reconstruction queries if the validation is performed with the exhaustive search technique. Otherwise, the attack requires $2^{14}$ queries.
- **Memory**: we need to store $2^9$ AES states, meaning the $2^9$ values of $\tilde{C}_0$.

Here a brief summary:

| **Validation Technique Applied** | **Encr** | **Mem. Acc.** | **Data** | **Store (Memory)** |
|---|---|---|---|---|
| Exhaustive Search | $2^{16}$ | $2^{13}$ | $2^{13}$ | $2^9$ |
| Checking with differential characteristic | 1 | $2^{14}$ | $2^{14}$ | $2^9$ |

### 4.1 Possible Countermeasures

What it makes the attack easier to apply is the injection of the tweak, which is never modified during the process. That characteristic helps us to reduced the complexity of the differential trail. Then, a possible countermeasure could be to modify the tweak during the process in such a way that the tweaks applied in the two branches are always different. In particular, a permutation of the bytes could be applied. In our example, 5+2 rounds ForkAES, we would need to generate tweaks from $T_0$ to $T_9$ as done with the round keys. The application of the round tweaks should be done by adding the round tweak $i$ to the round $i$ state and so on.
A possible simple implementation can be found on the repository. In particular, the file `utilities.c` contains a procedure called `TweakExpansion` which permutes the value within the initial tweak.
The countermeasure should work also for 3, 4 and 5 rounds after the fork. What is important is to implement a scheduling function for the tweak which is able to avoid cycles after the fork in order to prevent the exploitation of similarities between the two branches.

## 5 Attack on ForkAES-*-2-2 with reflection Trails (64 bits Tweak)

The attack works in the same way as the previous one, then, for an arbitrary number of rounds before the state is forked and 2 rounds after the fork, we are able to recover some bytes of the keys. With respect to the previous attack, in which we were able to find all the bytes of $\tilde{K}_7$, such scenario will give us the possibility to find only half of the bytes (all the bytes with $index = 0 + 4i$ and $index = 1 + 4i$ $\forall i \in \{0 \dots 3\}$).
We can apply the same attack procedure as before with the same differential reflection trail.

**Results** We are able to restrict the possibilities to 2-4 candidates for each half of the column, meaning that we need to test at most $2^2 \times 2^2 \times 2^2 \times 2^2 = 2^{16}$ upper half keys. Sometimes, it is possible to validate such possibilities, reducing them to almost $2^{14}$. The other 8 bytes should be exhaustive searched with a complexity of $2^{64}$, making the attack infeasible. Indeed, the overall complexity becomes $2^{64} \times 2^{16} = 2^{80}$. By using such trail, the only way to reduce the attack complexity would be to exploit key-scheduling features. Remember that the attack can be performed from $C_0$ to $C_1$ and vice versa, meaning that we are able to find key bytes both for $\tilde{K}_9$ and $\tilde{K}_7$. We know that in a certain way, those bytes are linked together by the fact that they are involved in the key scheduling algorithm: is it possible to retrieve the other bytes by only knowing the upper half of 2 almost consecutive keys?
Previous studies ([3], [2]) computed the complexity of the attack applied to 3 and 4 rounds after the fork with the extended trail. The following table compares the attack complexity of the 2 rounds scenario with the latter ones.

| | **2 rounds** | **3 rounds** | **4 rounds** |
|---|---|---|---|
| **Encryptions** | $2^{80}$ | $2^{29.7}$ | $2^{28}$ |
| **Memory Accesses** | $2^{12}$ | $2^{19}$ | $2^{19}$ |
| **Data** | $2^{12}$ | $2^{19}$ | $2^{35}$ |
| **Store (Memory)** | $2^{9}$ | $2^{17}$ | $2^{33}$ |

The main outcome is that ForkAES-*-2-2 guarantees more security than the other cases with respect to time complexity (number of encryptions).

**Possible improvements** As pointed out in the previous paragraph, we can apply the above technique from $C_0$ to $C_1$ and vice versa, meaning that we can retrieve 8 bytes (the upper half) of $\tilde{K}_7$ and 8 bytes of $\tilde{K}_9$. To improve the attack we can study the correlations introduced by the key scheduling algorithm and make some notes on how many bytes I need to know in order to discover the other ones both in the context of real keys ($K_i$ (no tilde) etc...), both in reversed keys context ($\tilde{K}_i$).

# 6 Exploiting Key-Scheduling features for ForkAES-*-2-2

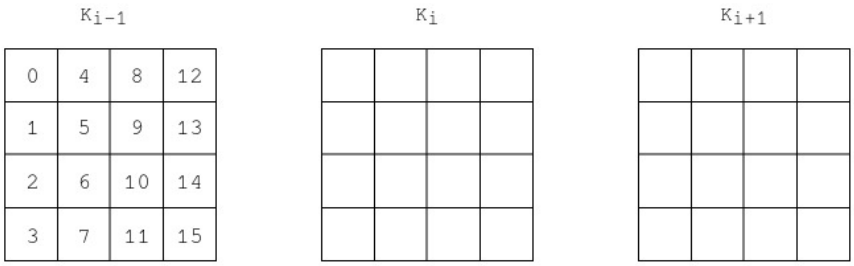


Figure 3: Calling convention

In the next subsections we will take into account 3 consecutive keys. Note that in our analysis we are able to find the upper half of $K_7$ and $K_9$, then we will consider $K_7$ as $K_{i-1}$, $K_8$ as $K_i$ and $K_9$ as $K_{i+1}$ (as shown in Figure 3). Moreover, we will assume that such bytes are exactly known, without considering the fact that we are not sure about their exact value because we found a number of possibilities, not only one possibility.
In the rest of the report we will refer to keys before the application of inverse mix-column operation as `Real Keys` and to keys after its application as `Reversed Keys`.

### 6.1 Real Keys context

**Context:** we have half bytes of $K_7$ and half bytes of $K_9$ (the upper half). Can we reduce the complexity of a complete exhaustive search?
By exploiting key scheduling features we can discover some bytes of $K_i$ from half bytes of $K_{i+1}$ and $K_{i-1}$. The following equations will help to find bytes $0, 1, 4, 5, 8, 9, 12, 13$ of $K_i$:

| **Equation** |
|---|
| $K_i[12] = K_{i+1}[8] \bigoplus K_{i+1}[12]$ |
| $K_i[8] = K_{i+1}[4] \bigoplus K_{i+1}[8]$ |
| $K_i[4] = K_{i+1}[0] \bigoplus K_{i+1}[4]$ |
| $K_i[0] = K_{i-1}[4] \bigoplus K_i[4]$ |
| $K_i[13] = K_{i+1}[9] \bigoplus K_{i+1}[13]$ |
| $K_i[9] = K_{i+1}[5] \bigoplus K_{i+1}[9]$ |
| $K_i[5] = K_{i+1}[1] \bigoplus K_{i+1}[5]$ |
| $K_i[1] = K_{i-1}[5] \bigoplus K_i[5]$ |

Cause we are in the Real Keys context, we can apply the SBOX as done in the key scheduling algorithm. From that we can easily find other 2 bytes of $K_i$.

| **Equation** |
|---|
| $K_i[14] = INV\_SBOX(K_{i+1}[1] \bigoplus K_i[1])$ |
| $K_{i-1}[14] = INV\_SBOX(K_i[1] \bigoplus K_{i-1}[1])$ |
| $K_i[10] = K_{i-1}[14] \bigoplus K_i[14]$ |

At that moment, we discovered 10 bytes of the key and we had reduced the complexity of a brute-force attack from $2^{64}$ to $2^{48}$.

### 6.2 Reversed Keys context

The key scheduling features extend to reversed keys too. The only difference is that the equations involving SBOX will not work in the same way. Cause we are applying the inverse mix-column

operation, it applies to SBOX values too. For example, the difference between $\tilde{K}_i[0]$ and $\tilde{K}_{i+1}[0]$ will not be $SBOX[K_i[13]]$ but a linear combination of $SBOX[K_i[12]]$, $SBOX[K_i[13]]$, $SBOX[K_i[14]]$, $SBOX[K_i[15]]$. It will be shown in the next paragraph.
As done in the previous case, we can easily find bytes $0, 1, 4, 5, 8, 9, 12, 13$ of $\tilde{K}_i$:

| **Equations** |
|---|
| $\tilde{K}_i[12] = \tilde{K}_{i+1}[8] \bigoplus \tilde{K}_{i+1}[12]$ |
| $\tilde{K}_i[8] = \tilde{K}_{i+1}[4] \bigoplus \tilde{K}_{i+1}[8]$ |
| $\tilde{K}_i[4] = \tilde{K}_{i+1}[0] \bigoplus \tilde{K}_{i+1}[4]$ |
| $\tilde{K}_i[0] = \tilde{K}_{i-1}[4] \bigoplus \tilde{K}_i[4]$ |
| $\tilde{K}_i[13] = \tilde{K}_{i+1}[9] \bigoplus \tilde{K}_{i+1}[13]$ |
| $\tilde{K}_i[9] = \tilde{K}_{i+1}[5] \bigoplus \tilde{K}_{i+1}[9]$ |
| $\tilde{K}_i[5] = \tilde{K}_{i+1}[1] \bigoplus \tilde{K}_{i+1}[5]$ |
| $\tilde{K}_i[1] = \tilde{K}_{i-1}[5] \bigoplus \tilde{K}_i[5]$ |

In this case, we cannot find bytes 10, 14 because we are not able to compute the correct difference cause it is a linear relation between the SBOX values of the last column, which we don't entirely know. Vice versa, the difference between $K_i[j]$ and $K_{i+1}[j]$ is a linear combination of the differences between $\tilde{K}_i[j]$ and $\tilde{K}_{i+1}[j]$ for each $j \in \{0 \dots 3\}$. Such correlation will be explained in the next paragraph.
We need to exhaustive search other 16 bytes, meaning that the complexity is $2^{64}$. We can slightly reduce the complexity of the attack exploiting the fact that the differences between $K_i[j]$ and $K_{i-1}[j]$ with $j \in \{0 \dots 3\}$ are linear relations of the SBOX values of the last column of the correspondent real key column which can be computed by applying the mix column operation.

- Compute all the possible values for bytes $2, 3$ of key $\tilde{K}_i$: $2^{16}$
- Compute all the possible values for bytes $2, 3$ of key $\tilde{K}_{i+1}$: $2^{16}$
- We obtained $2^{16} \times 2^{16}$ couples. For each couple:
  - Fill $\tilde{K}_i[2]$ and $\tilde{K}_i[3]$ with one couple.
  - Fill $\tilde{K}_{i+1}[2]$ and $\tilde{K}_{i+1}[3]$ with the other couple.
  - From that point we can follow two ways:

    **I** 1) Compute the Real Keys column 0 of $K_i$ and $K_{i+1}$ by applying the mix-column operation to the column 0 of the reversed keys.

    2) Compute the differences between those bytes which will be the SBOX values and apply the inverse SBOX operation in order to retrieve the last column of $K_i$.

    3) Compute the reversed bytes of the last column by applying the inverse mix-column operation and check whether the first two bytes (12,13) are equal to the known ones. If the column is valid, save the found couples and the retrieved last column, in particular bytes 14 and 15.

**II** 1) Compute the differences

$$\Delta_0 = \tilde{K}_{i+1}[0] \bigoplus \tilde{K}_i[0]$$
$$\Delta_1 = \tilde{K}_{i+1}[1] \bigoplus \tilde{K}_i[1]$$
$$\Delta_2 = \tilde{K}_{i+1}[2] \bigoplus \tilde{K}_i[2]$$
$$\Delta_3 = \tilde{K}_{i+1}[3] \bigoplus \tilde{K}_i[3]$$

2) From such values we can compute the values $SBOX[K_i[j]]$ for each $j \in \{0 \dots 3\}$. Notice that, as regards $\Delta_0$, it involves also the $RCON[i]$ value which need to be xored after having solved the system.
3) From those values we can retrieve $K_i[j]$ for each $j \in \{0 \dots 3\}$ by applying the inverse sub-bytes.
4) We retrieve all the last column of $K_i$. By applying the inverse mix-column operation we can retrieve the last column of $\tilde{K}_i$. We can validate such column by checking if the first two bytes of the column are equal to the already known bytes 12 and 13 of $\tilde{K}_i$.
5) If the column is valid, save the found couples and the retrieved last column, in particular bytes 14 and 15.

**Linear relations involving $\tilde{K}_{i+1}[j] \bigoplus \tilde{K}_i[j]$ for $i \in \{0 \dots 3\}$** In case **(2)**, in order to retrieve the values of each $SBOX[K_i[j]]$ we need to apply the mix-column operation to the vector composed by $\Delta_0, \Delta_1, \Delta_2, \Delta_3$:

$$\begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix} \times \begin{bmatrix} \Delta_0 \\ \Delta_1 \\ \Delta_2 \\ \Delta_3 \end{bmatrix} = \begin{bmatrix} SBOX[K_i[13]] \bigoplus RCON[i] \\ SBOX[K_i[14]] \\ SBOX[K_i[13]] \\ SBOX[K_i[12]] \end{bmatrix}$$

**Applying this method to reduce the attack complexity** By applying that technique we were able to reduce the number of possibilities for the bytes $2, 3, 14, 15$ of $\tilde{K}_i$ from $2^{32}$ to almost $2^{16}$ (in some cases better results are possible). Then, we can follow two ways to retrieve the remaining 4 bytes. The first one is to exhaustive search the remaining 4 bytes, meaning that the overall complexity to find the lower half of the key reduces from $2^{64}$ to almost $2^{48}$. The second one is to take into account also the third key. It can be used to validate more bytes and reduce the complexity to find the lower half of the key from $2^{48}$ to $2^{34}$.
In particular, we will apply the above technique to the couple of keys $\tilde{K}_{i-1}$ and $\tilde{K}_i$ and to the couple of keys $\tilde{K}_i$ and $\tilde{K}_{i+1}$. In this way, we reduced the number of possibilities for the bytes $\tilde{K}_{i-1}[2, 3, 14, 15]$, $\tilde{K}_i[2, 3, 14, 15]$ and $\tilde{K}_{i+1}[2, 3]$. The above algorithm returns validated couples. In particular, by applying it to keys $\tilde{K}_{i-1}$ and $\tilde{K}_i$ we will obtain couples of the type ($\tilde{K}_{i-1}[0, 1, 2, 3]$, $\tilde{K}_i[0, 1, 2, 3]$ ) and, by applying the same technique to keys $\tilde{K}_i$ and $\tilde{K}_{i+1}$ we will obtain couples of the type ($\tilde{K}_i[0, 1, 2, 3]$, $\tilde{K}_{i+1}[0, 1, 2, 3]$ ). You can notice that the second element of the first type of couples and the first one of the second type should be identical because they are referring to the same bytes. That means that we can perform the intersection between those types of couples and retrieve triples of the type ($\tilde{K}_{i-1}[0, 1, 2, 3]$, $\tilde{K}_i[0, 1, 2, 3]$, $\tilde{K}_{i+1}[0, 1, 2, 3]$). These triples represent the column 0 of the three keys and they can be validated by the following algorithm:

1) Compute column 0 of $K_{i-1}$, $K_i$, $K_{i+1}$ by applying the `Mix Column` operation.

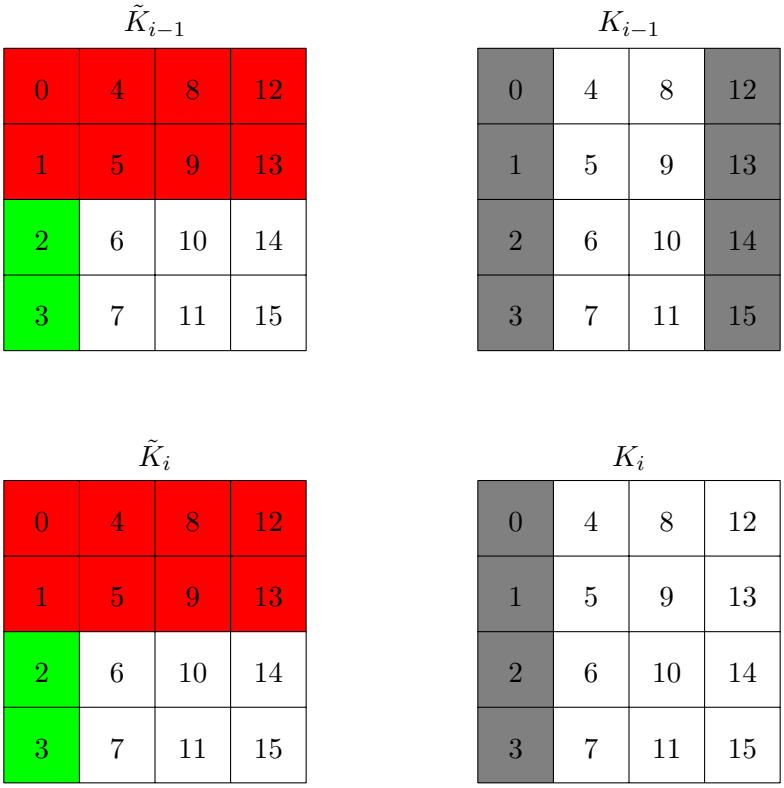


Figure 4: First exhaustive search
- Exhaustive searched
- Known bytes
- Discovered bytes

2) From column 0 of $K_{i-1}$ and $K_i$ compute column 3 of $K_{i-1}$ (Figure **??**).

3) From column 0 of $K_i$ and $K_{i+1}$ compute column 3 of $K_i$ (the same as done in Figure **??**).

4) From the just computed column 3 of $K_{i-1}$ and $K_i$ compute column 2 of $K_i$ (Figure **??**).

5) Apply the `Inverse Mix Column` operation to the just computed column 2 of $K_i$ and check whether the bytes $\tilde{K}_i[8, 9]$ we already know are identical to the bytes $8, 9$ of the previously inverted column.

6) If yes, save the triple, otherwise discard it.

At the end of the computation, we should obtain almost $2^{17}$ triples. We left only 2 bytes ($\tilde{K}_i[6, 7]$) which will need to be exhaustive searched ($2^{16}$ possibilities). The time complexity is given by the intersection ($O(2^{34})$), whilst the number of key possibilities is $2^{16} \times 2^{17} = 2^{33}$ instead of $2^{64}$

# 7 Complexity

The table in Section 5 can be rewritten by taking into account the above results. In particular, the number of encryption was reduced.

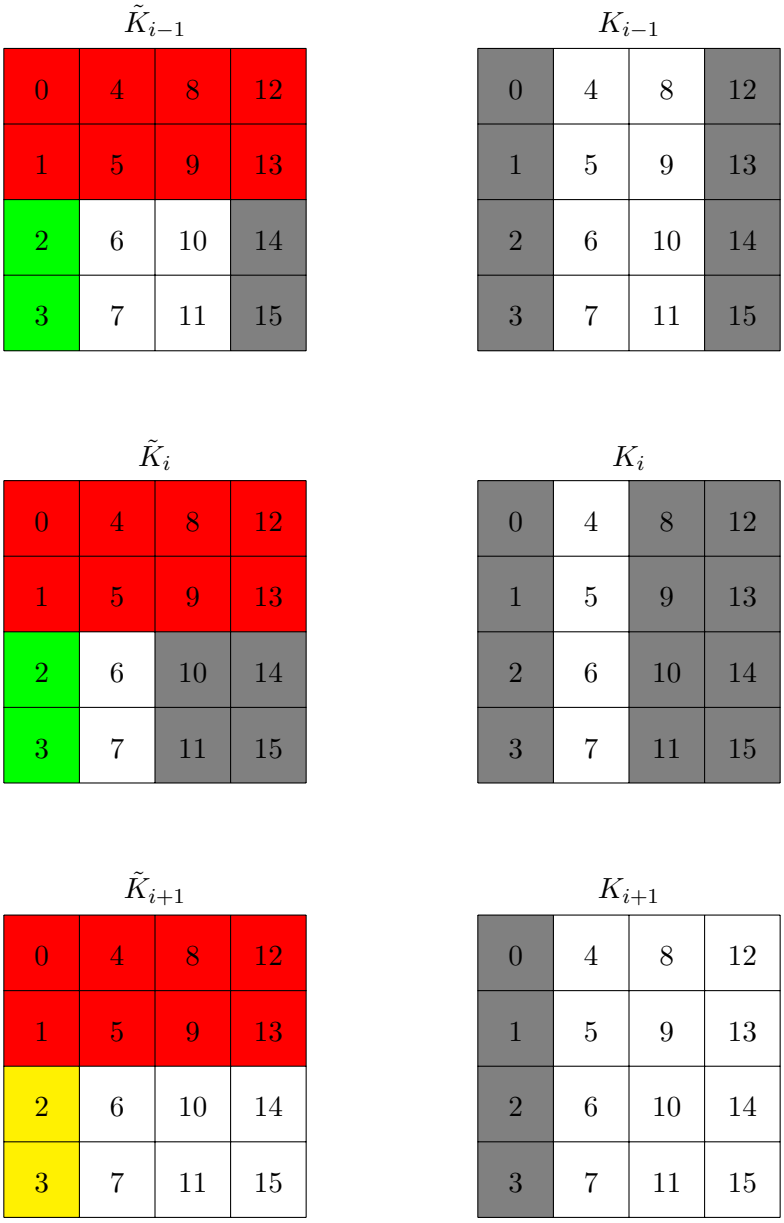


Figure 5: Validation and bytes that can be found

- Exhaustive searched
- Exhaustive searched
- Known bytes
- Discovered bytes

- The attack gives us $2^{16}$ possibilities for the upper half of $\tilde{K}_7$ which can be reduced by validation to almost $2^{14}$
- The same attack gives us $2^{16}$ possibilities for the upper half of $\tilde{K}_9$ which can be reduced by validation to almost $2^{14}$
- For each couple of possibilities, we can directly compute the upper half $\tilde{K}_8$.
- For each couple of possibilities, we can apply the above technique to find the possibilities for the lower half of $\tilde{K}_8$.

With the previous technique the number of encryption would be $2^{16} \times 2^{64} = 2^{80}$. By applying the above method the number decreases. In particular,in the worst case it is almost $(2^{16} \times 2^{16}) \times 2^{33} = 2^{65}$.

In this way we were able to reduce the number of possible keys to be tested. Unfortunately, it requires more memory accesses, store and data.

| | **2 rounds** | **3 rounds** | **4 rounds** |
|---|---|---|---|
| **Encryptions** | $2^{65}$ | $2^{29.7}$ | $2^{28}$ |
| **Memory Accesses** | $2^{49}$ | $2^{19}$ | $2^{19}$ |
| **Data** | $2^{18}$ | $2^{19}$ | $2^{35}$ |
| **Store (Memory)** | $2^{17}$ | $2^{17}$ | $2^{33}$ |

# 8 Conclusions

This report considers the attack on 2 rounds after the fork which was never considered by previous works. The analysis shows two main outcomes. First of all, that ForkAES-*-2-2 is stronger than ForkAES-*-3-3 and ForkAES-*-4-4 with respect to the application of reflection trail techniques.
Secondly, we showed how in such scenario the 64 bit tweak is much more secure than a 128 bit tweak because it cannot be crafted as wanted and it cannot be used to retrieve all the bytes of the key. Indeed, nevertheless the improvement in the attack to ForkAES with 64 bits tweak, the number of possible keys to test keeps to be greater then the attacks against 3 and 4 rounds ForkAES and the attack against 2 rounds ForkAES with 128 bits tweak. ForkAES-*-2-2 with 128 bits tweak can be broken with a number of encryptions equal to $2^{16}$ and memory accesses equal to $2^{13}$. On the other hand, it was shown that the version with 64 bits tweak is more secure against such type of attack because it requires a number of encryptions equal to $2^{65}$ and memory accesses equal to $2^{49}$. As a result, in contrast with the common thinking, using less bytes of tweak within ForkAES results in a stronger version of the cipher. That fact coincides with the capabilities requirements of IoT devices, which need to limit resource consumption, in this case the memory usage.
Future researches on the key scheduling correlations than can be exploited in such scenarios could affect the performance of the attacks by reducing the number of bytes to be exhaustive searched, making the attack as feasible as possible. Moreover, taking into account other kind of cryptanalysis

techniques in order to check if the outcomes about the tweak dimension are the same could be a possible research line.

# 9 Appendix

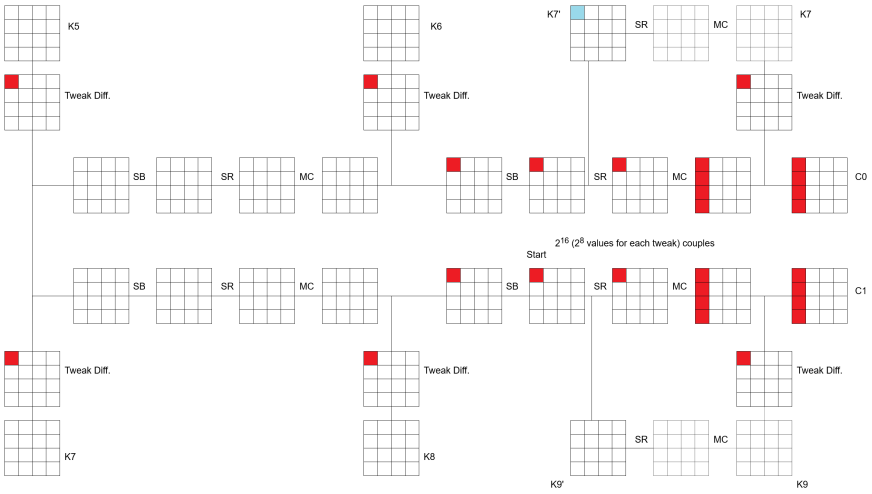


Figure 6: Differential Reflection Trail: Byte 0

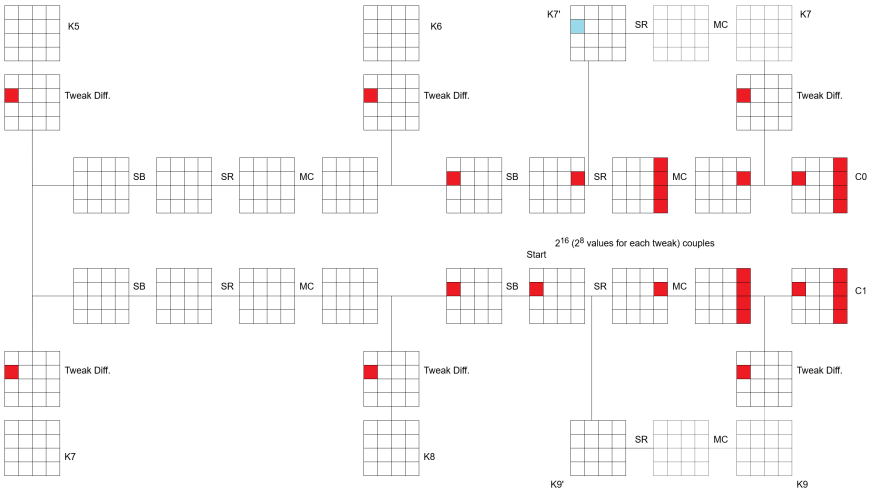


Figure 7: Differential Reflection Trail: Byte 1

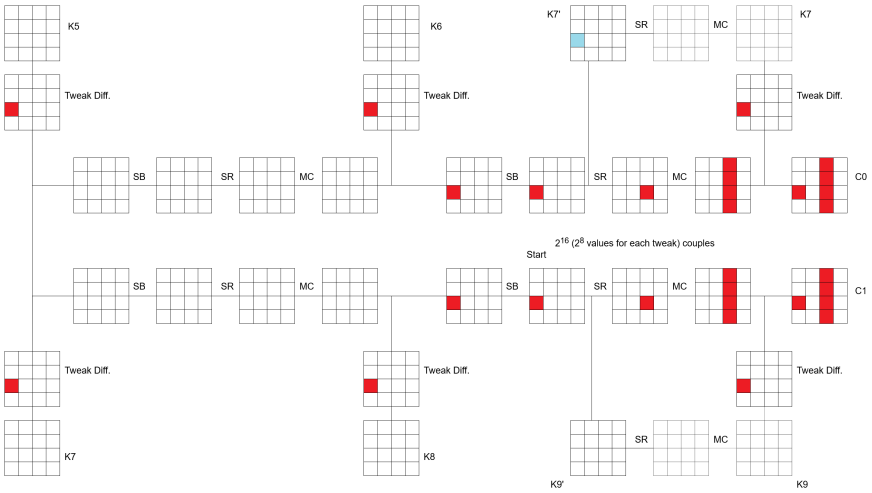


Figure 8: Differential Reflection Trail: Byte 2

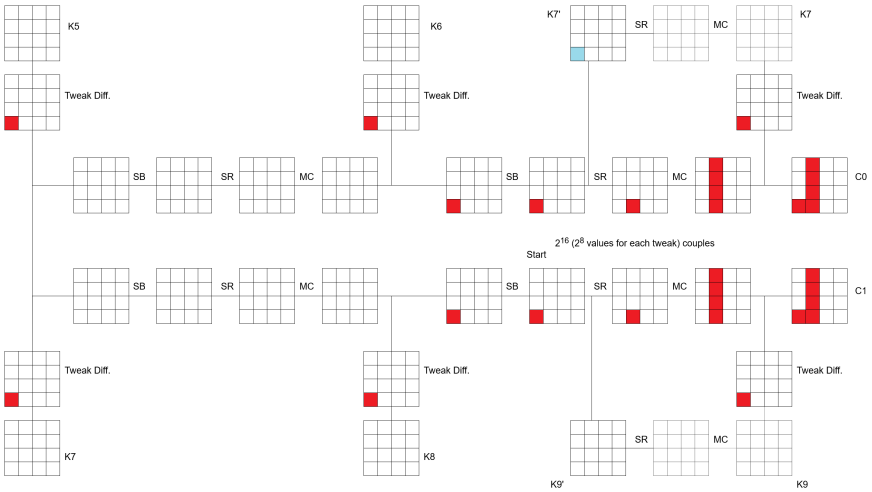


Figure 9: Differential Reflection Trail: Byte 3

## References


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

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

[3] Eik List Jannis Bossert and Stefan Lucks. Rectangle and impossible-differential cryptanalysis on versions of forkaes, 2018. https://eprint.iacr.org/2018/1075.