Daniel Kales
Graz University of Technology
daniel.kales@student.tugraz.at

Robustness of CAESAR candidates

Master Project

Graz University of Technology
IAIK – Institute for Applied Information Processing and Communications

Advisors: Maria Eichlseder and Florian Mendel

Graz, January 30, 2017

Abstract

Nonces are used in cryptographic primitives to induce randomness and freshness. Authenticated ciphers rely on the uniqueness of the nonces to meet their security goals. In this work, we investigate the implications of reusing nonces for several third-round candidates of the ongoing CAESAR competition, namely Tiaoxin, AEGIS and MORUS. We show that an attacker that is able to force nonces to be reused can reduce the security of the ciphers with results ranging from full key-recovery to forgeries (with practical complexity).

I. Introduction

The CAESAR authenticated encryption competition was initiated to encourage the design and analysis of authenticated encryption ciphers. Almost all authenticated encryption schemes are nonce-based schemes [6], as these constructions have been proven easy to describe and provide security proofs for. A nonce is a public, unique and (usually) fixed length number, which is necessary to hide plaintext equality and introduce freshness into internal state parts. While nonces are not necessarily random or unpredictable, these are two desirable properties, as they can further reduce attack vectors. It is most important that a nonce is used only once for a specific key. If this property is violated, many new attacks vectors can surface. For example, a single known plaintext-ciphertext pair with the same nonce is enough to forge valid ciphertext-tag pairs for arbitrary plaintexts when using AES-GCM [2].

In this work, we analyze the impact of using a repeated nonce for three of the third round candidates of the CAESAR Authenticated Encryption competition. Most of their security claims are based on the correct use of the public nonce.

Attack Scenario. Our attack scenario for the ciphers is a chosen-plaintext attack, where we also fix the public nonce to be constant during the attack. The targeted results of the attack include forgeries, state recovery and key recovery.

Attack Targets. We chose the three third-round candidates Tiaoxin [5], AEGIS [9] and MORUS [8] for our analysis. In Section II, we will shortly present the structure of each of the chosen ciphers, whereas a more detailed description can be found in the respective design documents. Having a similar structure, these three ciphers also present similar angles of attack.

Attack Results. We show that all three candidates investigated in this work are vulnerable to the proposed attack scenario. A detailed description of the attacks can be found in Section III, and a summary of the results can be found in Section IV. Moreover, all presented attacks were proposed and implemented by the author of this work.

II. Description of CAESAR Candidates

We chose the following three second-round candidates: Tiaoxin [5], AEGIS [9] and MORUS [8]. In this section, we will shortly describe the structure and encryption process for each of the three ciphers.

i. Tiaoxin

Tiaoxin is a family of authenticated encryption algorithms designed by Nikolić [5]. We focus on the Tiaoxin-346 version of the algorithm, but the idea of the attack can be easily extended to the other versions. The internal state consists of 13 words of 16 bytes each. The 13 words are divided into three groups of 3, 4 and 6 words each (this is also the reason for the name Tiaoxin-346). The state update function for Tiaoxin-346 absorbs a message block of 32 bytes and produces a new internal state, as illustrated in Figure 1.

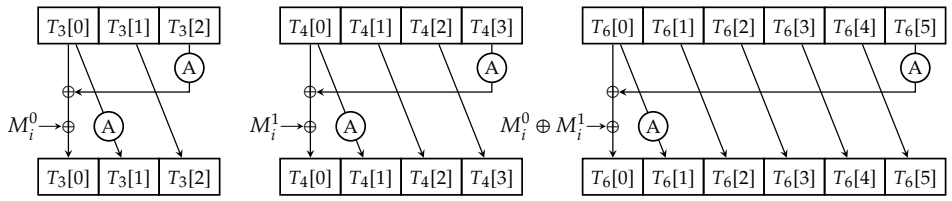

Figure 1: The *Update* operation (round function) of Tiaoxin-346. The circled A represents one round of AES. A more detailed explanation can be found in the design document [5].

Each of the three groups of the internal state is initialized using the secret key and the public nonce. Afterwards, a number of calls to the state update function (with constants in place of the message) is performed, resulting in the initial state for the encryption phase. We will now briefly summarize the encryption phase of Tiaoxin-346: The message is padded and then divided into m blocks of 32 bytes each. Each message block M_i is divided into two parts ($M_i = M_i^0 || M_i^1$) and the result is a ciphertext block $C_i = C_i^0 || C_i^1$. The encryption is performed as follows:

for $i = 1$ to m
$\quad$ Update$(T_3, T_4, T_6, M_i^0, M_i^1, M_i^0 \oplus M_i^1)$
$\quad C_i^0 = T_3[0] \oplus T_3[2] \oplus T_4[1] \oplus (T_6[3] \wedge T_4[3])$
$\quad C_i^1 = T_6[0] \oplus T_4[2] \oplus T_3[1] \oplus (T_6[5] \wedge T_3[2])$
end for

Due to its structure, the round update function is fully invertible given the plaintext. This means that if we can recover the internal state (either $T3$, $T4$ or $T6$) in our attack, we can apply the inverse round function and recover the corresponding initial state after the initialization. Furthermore, since

the initialization phase of Tiaoxin-346 also entirely consists of calls to the round update function, we can completely recover the initial state before the initialization phase, which contains the secret key. Therefore, recovering the internal state for a known plaintext is equivalent to recovering the secret key.

ii. AEGIS

AEGIS is a family of authenticated encryption algorithms designed by Wu et al. [9]. We only focus on the AEGIS-128 version of the algorithm, but again, the idea of the attack can be extended easily to other versions. The internal state of AEGIS-128 consists of 5 words of 16 bytes each. The state update function consumes a plaintext block of 16 bytes and produces the new internal state. A graphical representation of the state update function can be seen in Figure 2.

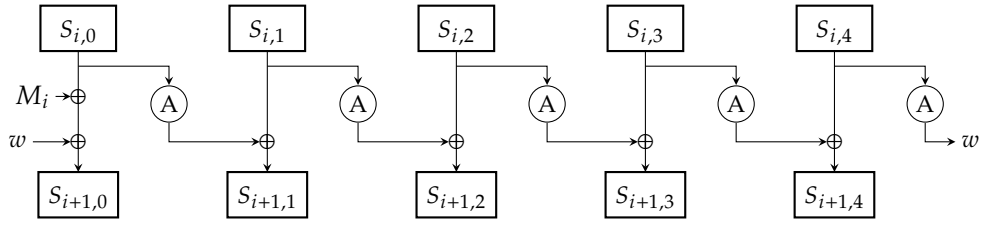

Figure 2: The state update function of AEGIS-128. The circled A represents one round of AES. A more detailed explanation can be found in the design document [9].

Like Tiaoxin, the internal state is also initialized with the secret key and the public nonce. Then the state is updated with the state update function a fixed number of times, using constants in place of the message words. This process results in the initial state used for the encryption phase. We will now shortly summarize the encryption phase of AEGIS-128:

$$
\begin{aligned}
&\text{for } i = 1 \text{ to } m \\
&\quad C_i = M_i \oplus S_{i,1} \oplus S_{i,4} \oplus (S_{i,2} \wedge S_{i,3}) \\
&\quad S_{i+1} = \text{StateUpdate}(S_i, M_i) \\
&\text{end for}
\end{aligned}
$$

In contrast to the state update function of Tiaoxin-346 (Figure 1), the state update function of AEGIS-128 is not easily invertible. This means that recovering the internal state does not allow us to recover the secret key.

iii. MORUS

MORUS is a family of authenticated encryption ciphers designed by Wu et al. [8]. We only focus on the MORUS-640-128 version of the cipher, but as

with the two other ciphers before, the ideas of the attack are easily extended to the other versions of the cipher. The internal state of MORUS-640-128 consists of 5 words of 16 bytes each. The state update function consumes a plaintext block of 16 bytes and updates the internal state. A graphical representation of the state update function of MORUS-640-128 can be seen in Figure 3. In contrast to Tiaoxin-346 and AEGIS-128, the state update function is not based on AES, but instead is a custom construction using ANDs, rotations and XORs (ARX). It uses two kind of rotations internally: A rotation of one full message word with different constant rotation offsets (denoted by $\lll w_i$) and a second kind of rotation, where one 16 byte internal state is split into four 4-byte blocks, which are rotated independently (denoted by Rotl_128_32(b_i)).

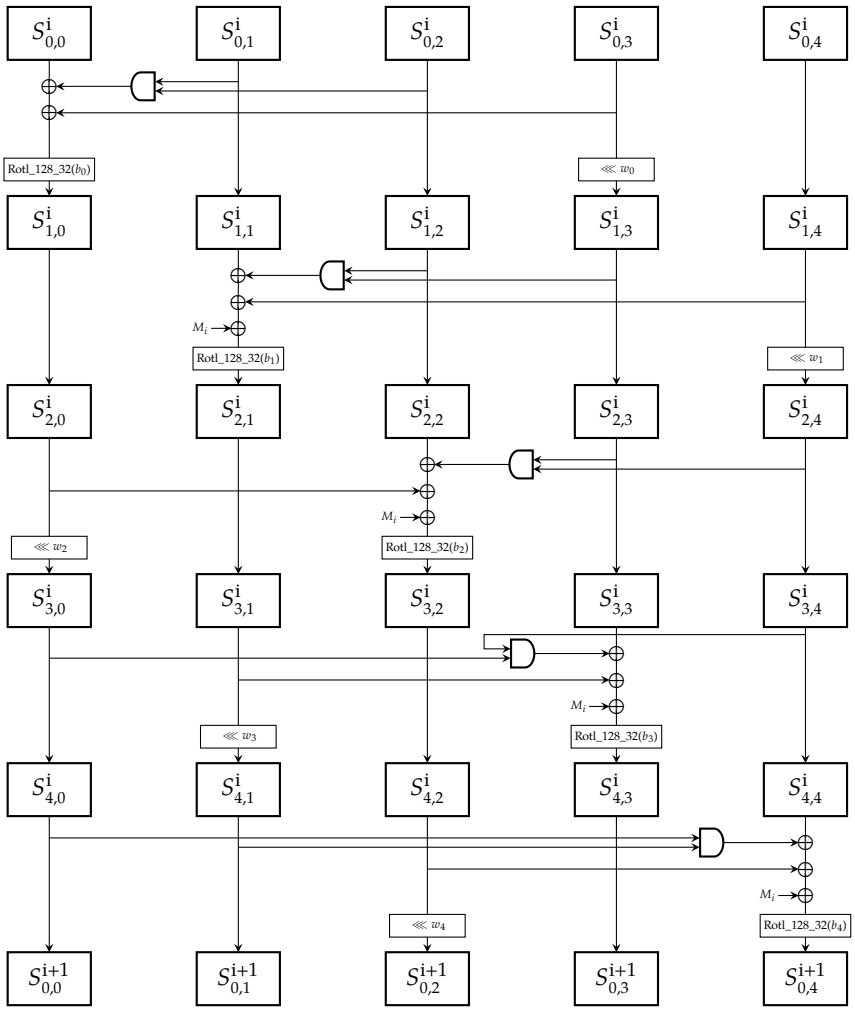

Figure 3: The state update function of MORUS-640-128. A more detailed explanation can be found in the design document [8].

As with the two previous ciphers, the internal state is initialized with the secret key and the public nonce. The internal state is then updated with a fixed number of calls to the internal state function (with constants in place of the message words). After the final initialization round, the secret key is

again XORed to one part of the internal state. This results in the initial state used for encryption. We will now shortly summarize the encryption phase of MORUS-640-128:

$$
\begin{aligned}
&\text{for } i = 1 \text{ to } m \\
&\quad C_i = M_i \oplus S_0^i \oplus (S_1^i \lll 96) \oplus (S_2^i \wedge S_3^i) \\
&\quad S_{i+1} = \text{StateUpdate}(S_i, M_i) \\
&\text{end for}
\end{aligned}
$$

Like the state update function of Tiaoxin-346, the state update function of MORUS is invertible given the internal state. The initialization phase, however, is not, due to the fact that the key is XORed to a part of the state after the initialization rounds.

iv. Similarities of the Three Candidates

All of the candidates analyzed in this work use the following way to generate a key stream block: The internal state is initialized with the IV and the key and a few rounds of the state update function are applied. After the initial initialization, parts of the internal state (a, b, c and d) are combined with a mix of linear and non-linear operations. The resulting keystream KS is then XORed with the plaintext P to generate the ciphertext C (Equations (1), (2)).

$$KS = a \oplus b \oplus (c \wedge d) \tag{1}$$

$$C = P \oplus KS \tag{2}$$

After the encryption phase, the message tag is generated by applying a few rounds of the state update function, before finally combining parts of the internal state. By keeping a, b and c constant and varying d, we can recover c with the following observation. Since XOR and AND operate on single bits, if we manage to get the values d to include both 0 and 1 for a range of trials, we can observe the following property for one bit of the xor of two trials:

$$
\begin{aligned}
KS_0 \oplus KS_1 &= a \oplus b \oplus (c \wedge d_0) \oplus a \oplus b \oplus (c \wedge d_1) \\
&= (c \wedge d_0) \oplus (c \wedge d_1) = (c \wedge 0) \oplus (c \wedge 1) = 0 \oplus c \\
&= c\,.
\end{aligned}
$$

If the values for the bit of d are equal, the result will always be 0, since all inputs to the xor are equal. This means we need at least one pair of trials that have different values for d.

In the following section, we present attacks that make use of these properties and allow recovery of the internal state.

III. Attack on the CAESAR Candidates

i. Tiaoxin

The encryption of Tiaoxin follows the structure in Equation (1):

$$C_i^1 = T_6[0] \oplus T_4[2] \oplus T_3[1] \oplus (T_6[5] \wedge T_3[2]) .$$

We attack the generation of C_2^1 with the following conditions: The state words $T_6[0]$, $T_4[2]$ and $T_3[1]$ of S_3 are kept constant, while a difference is introduced in $T_3[2]$. A graphical representation of the attack can be seen in Figure 4.

However, due to the AES call in the state update function, we cannot exactly predict the final difference in $T_3[2]$ that is introduced by our chosen plaintext difference Δ. This means we need to perform a higher number of trials to achieve a high probability that at least one pair of message differences has a difference in one bit of d. To get a reliable result for the attack, we use 128 random plaintexts. Since the input to the AES block is random, and AES is not biased with respect to its output, we get a probability of $\frac{1}{2}$ for a single bit to be either 0 or 1. This means the probability that one bit of the output is the same for all 128 messages is 2^{-127}. Combining the results for each of the 128 bits of the AES output block gives us a total success probability of

$$p_{\text{success}} = \left(1 - \frac{1}{2^{127}}\right)^{128} \approx 1 . \tag{3}$$

With smart choice of the message words, we can achieve the required properties of the attack. If we choose a difference (referred to as Δ in the following) for M_0^0 and no difference for M_0^1, this results in the same difference Δ in M_0^2. If we use the same values for the next message block M_1^0, M_1^1 and M_1^2, the differences cancel out for the most part. For the third round state S_3, only $T_3[2]$ and $T_6[2]$ contain any differences (see Figure 4).

Using this attack outline, we can recover one part of the internal state T_6 after three rounds of encryption. Furthermore, we can repeat our attack for later rounds: If we execute n rounds of encryption with a constant plaintext before our attack, we can recover $T_6[5]$ of state S_{3+n}. Additionally, the structure of the update function tells us that $T_6[5]$ of S_4 equals $T_6[4]$ of S_3 and a similar relation can be found for all parts of T_6 of S_3. So we can repeat our initial attack six times with offsets $0, \ldots, 5$ and recover the internal state T_6. As explained in Section II, this allows us to fully recover the secret key.

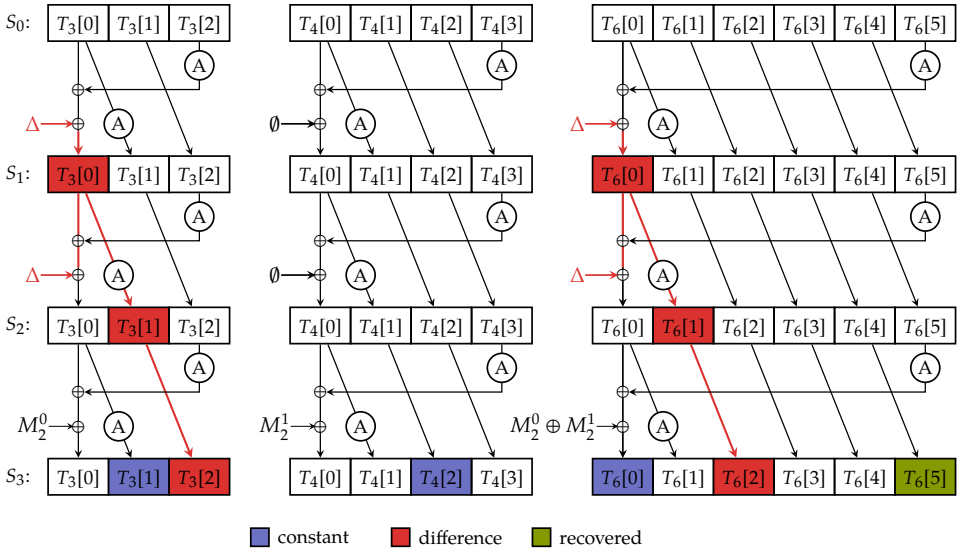

Figure 4: Propagation of differences in the attack scenario for Tiaoxin.

ii. AEGIS

The attack against AEGIS works according to the same general principle as the attack against Tiaoxin. Again, the structure of the encryption of AEGIS follows the structure in Equation 1:

$$C_i = M_i \oplus S_{i,1} \oplus S_{i,4} \oplus (S_{i,2} \wedge S_{i,3})\,.$$

Figure 5 depicts the structure of the attack for AEGIS. We introduce a difference Δ in the message word P_1 and use the same difference Δ in P_2 to cancel most of the effect. The only remaining difference propagates to the state word $S_{2,1}$ and can be recovered from the encryption equation of C_2, since the state words $S_{2,2}$, $S_{2,3}$ and $S_{2,4}$ are constant. The difference propagates to $S_{3,2}$ and since the difference in $S_{3,1}$ is equal to the previously recovered difference $S_{2,1}$ we can use our approach to recover the state $S_{3,3}$.

To recover more parts of the state, we have to repeat our attack one round later. Recovering $S_{4,3}$ in addition to $S_{3,3}$ enables us to calculate $S_{3,2}$ as can be seen in Figure 6. However, the message word P_1 has an influence on $S_{3,2}$ and therefore needs to be constant. This means we need to choose a fixed P_1 for our attack and can only recover a state for this choice of P_1. Repeating this attack for $S_{5,3}$ and $S_{6,3}$ in order to recover $S_{3,1}$ and $S_{3,0}$ in turn means we also have to fix P_2 and P_3 for our attack, meaning we can recover the state for a 3-block chosen-plaintext prefix. This enables us to create forged ciphertexts for messages starting with the same 3-block chosen-plaintext prefix and the fixed nonce used in the attack.

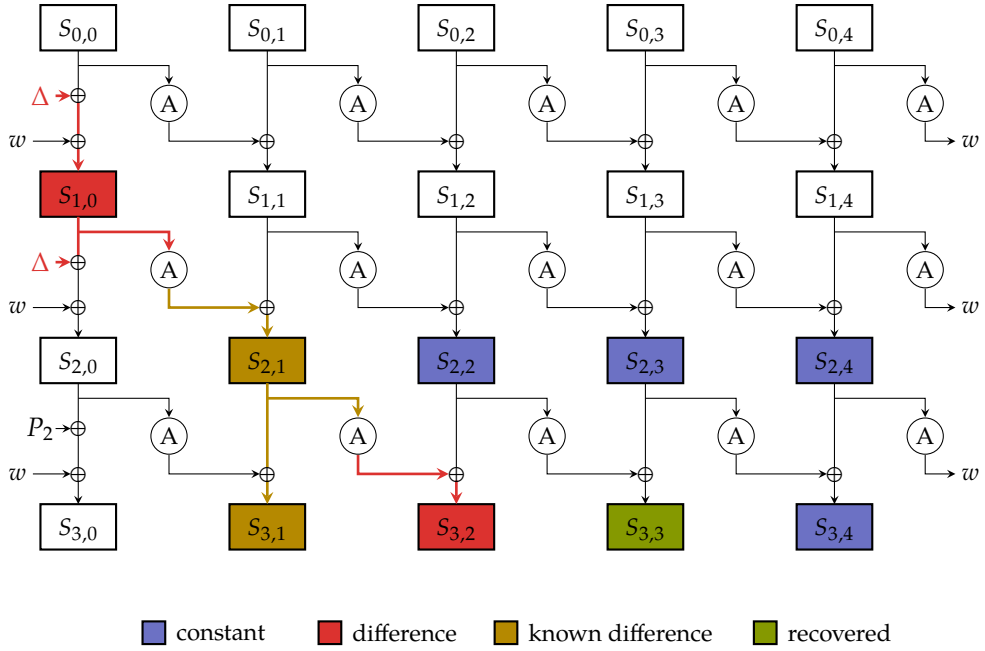

Figure 5: Propagation of differences in the attack scenario for AEGIS.

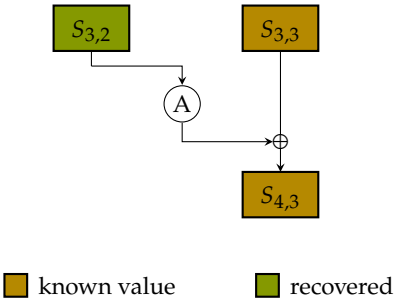

Figure 6: Combining two recovered states to recover more parts of the internal state.

iii. MORUS

The attack on MORUS is based on the same principle as the previous attacks. Again, the structure of the encryption of MORUS follows the structure in Equation 1:

$$C_i = M_i \oplus S_0^i \oplus (S_1^i \lll 96) \oplus (S_2^i \wedge S_3^i)\,.$$

Since the state update function does not use a round of AES, but XOR, AND and rotation operations instead, we can propagate desirable differences more easily. Instead of using about 2^7 encryption oracle calls for one part of the state, we can reduce the overall encryption oracle calls to below 32.

To recover the first part of the internal state, we choose a difference of $\Delta = 1^{128}$ for the first plaintext block. This difference spreads to $S_{0,1}^1$ and $S_{0,2}^1$ while $S_{0,3}^1$ stays constant, because the difference introduced by Δ is canceled between $S_{3,3}^0$ and $S_{4,3}^0$ (see Figure 7). This allows us to recover $S_{0,3}^1$ with only 2 encryptions.

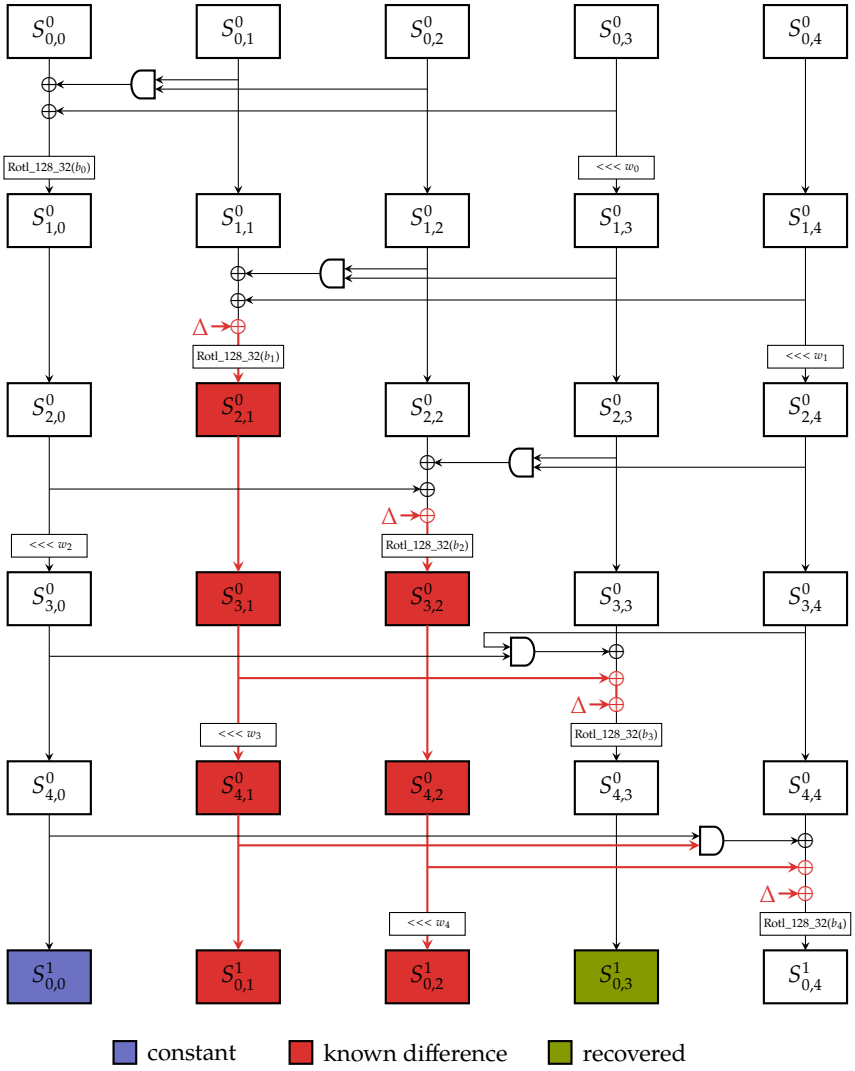

Figure 7: Propagation of differences to recover first state part in the attack on MORUS.

We can now recover an additional part of the state in a similar fashion. Taking advantage of the rotations in the cipher, we are able to generate a plaintext block that keeps half of $S^1_{0,2}$ constant, while introducing a difference in the corresponding half of $S^1_{0,3}$ (see Figure 8). Repeating this process for the other half allows us to fully recover $S^1_{0,2}$ with only 2 additional encryptions.

For the other state blocks we have to attack the second round of encryption. With the knowledge of the recovered state $S^1_{0,2}$ we can now calculate a plaintext block M_1, so that the resulting state of $S^1_{0,2}$ is equal to 0^{128}. This disables the AND gate in the first sub-step of the round update, meaning $S^1_{0,1}$ has no influence on $S^2_{0,0}$. If we now calculate a second plaintext M_1, so that $S^1_{0,2}$ is equal to $1^{32}||0^{96}$, we only allow one quarter of $S^1_{0,1}$ to be propagated into $S^2_{0,0}$. Due to the nature of the state rotation and its different rotation values for each state part, the newly introduced quarter block of ones only influences a specific

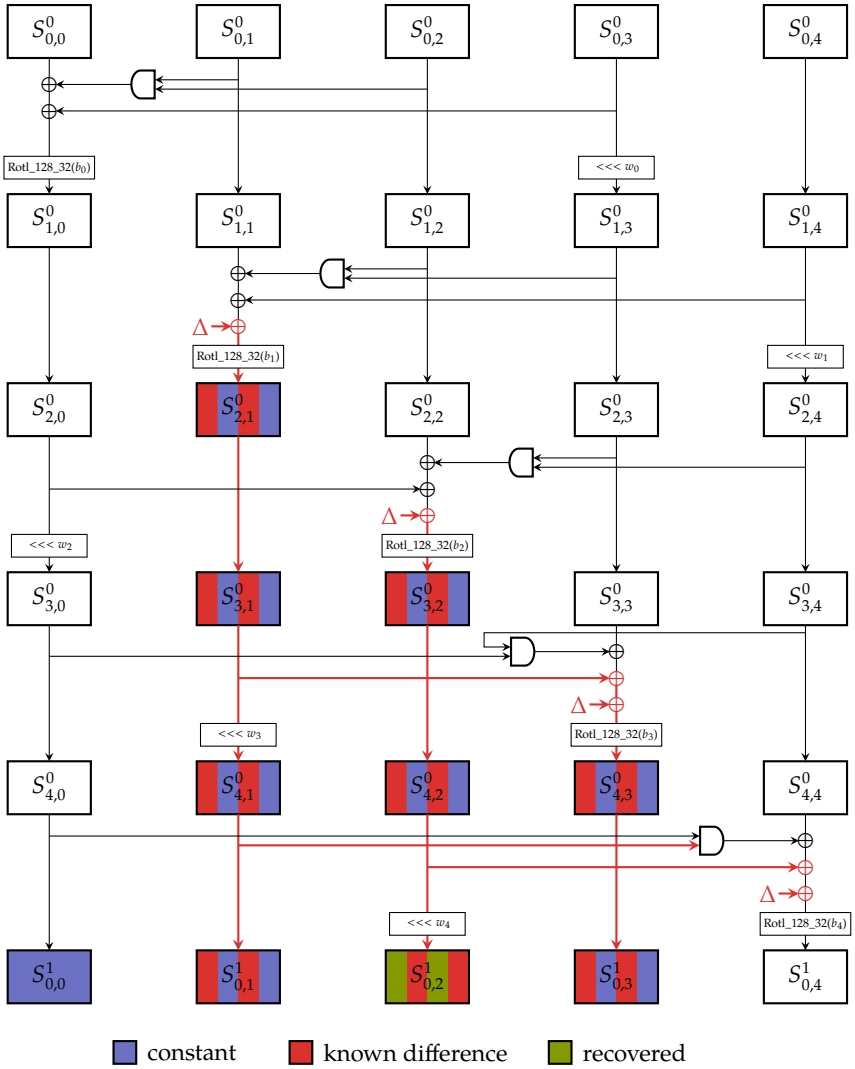

Figure 8: Propagation of differences to recover second state part in the attack on MORUS.

quarter of $S^2_{0,0}$, but does not influence the same quarter in the other state parts $S^2_{0,1-3}$. This means by comparing the output of the second-round encryptions of these two plaintexts, we can recover one quarter of $S^1_{0,1}$, and repeating this process 3 more times while setting different quarters of $S^1_{0,2}$ to 1^{32} allows us to fully recover the state $S^1_{0,1}$.

The state $S^1_{0,0}$ is then easily recovered with one known plaintext-ciphertext pair, since it is the only unknown part of the encryption equation of MORUS.

The final state part $S^1_{0,4}$ is then recovered by looking at the first and second round of encryption. We can recover the state $S^2_{0,1}$ by performing our attacks with an offset of one constant plaintext block. Now we can just look at the

equation given by the second column of the state update function:

$$S^2_{0,1} = S^1_{0,1} \oplus (S^1_{0,2} \wedge (S^1_{0,3} \lll w_0)) \oplus S^1_{0,4} \,.$$

Since we already know the value of every variable of the equation except $S^1_{0,4}$, we can easily calculate it by rearranging the equation.

We can then apply the inverse round update function to get the state S^0, but we cannot recover the secret key, since the initialization phase cannot be reversed without previous knowledge of the secret key. Nevertheless, knowledge of the internal state allows forgery of ciphertext-tag pairs for arbitrary plaintexts.

iv. Practical implementation

We implemented all of the above attacks and the results were verified using the designer-provided reference implementations of the respective cipher. The run-time of all of the three attack implementations is negligibly short (< 1 second each) and the success probability is essentially 100%. We did not encounter any issues during the implementation that caused us to change the theoretical attacks presented in Section III in any way. The source code for the attacks is available at GitHub [4].

IV. Results and Conclusion

The results of the attacks are summarized in Table 1.

Cipher	Encryption Oracle Calls	Result
Tiaoxin	2^{10}	Key recovery
AEGIS	2^9	Almost universal forgery
MORUS	2^5	Universal forgery

Table 1: Summary of attack complexity and results.

These results show the importance of correct nonce usage. The designers of all three algorithms stress that their security claims are invalid in case of nonce misuse and this work confirms it. This puts a burden on the developers implementing the system, as misconfiguration or wrong usage of the cipher could weaken the security immensely.

In the context of the CAESAR competition, these results are not showing any groundbreaking attacks or mistakes of the designers of the respective ciphers. The proposed attack scenario is not practical in an environment that was set up correctly and as previously said is even explicitly mentioned to void all security claims.

It is possible for a cipher to be resistant against these types of attacks. One solution would be to use the SIV scheme, which provides full nonce misuse

resistance [7]. However, full nonce misuse resistant schemes cannot be online, which is a desirable property for an authenticated encryption scheme. In the third round of the CAESAR competition there is only one cipher, AEZ [3], left providing full nonce misuse resistance. Additionally, there are other candidates in the third round that provide weaker notions of nonce misuse resistance. One example for such a cipher is COLM [1], which guarantees its full security up to the common prefix. In most cases, the additional overhead and loss of flexibility are a higher priority than the additional security provided by such schemes.

References

[1] E. Andreeva, A. Bogdanov, N. Datta, A. Luykx, B. Mennink, M. Nandi, E. Tischhauser, and K. Yasuda. *COLM v1*. Submission to the CAESAR Competition. 2016. https://competitions.cr.yp.to/round3/colmv1.pdf.

[2] H. Böck, A. Zauner, S. Devlin, J. Somorovsky, and P. Jovanovic. "Nonce-Disrespecting Adversaries: Practical Forgery Attacks on GCM in TLS". In: *USENIX Workshop on Offensive Technologies – WOOT 16*. USENIX Association, 2016.

[3] V. T. Hoang, T. Krovetz, and P. Rogaway. "Robust Authenticated-Encryption AEZ and the Problem That It Solves". In: *Advances in Cryptology – EUROCRYPT 2015*. Ed. by E. Oswald and M. Fischlin. Vol. 9056. LNCS. Springer, 2015, pp. 15–44. DOI: 10.1007/978-3-662-46800-5_2.

[4] D. Kales. *Implementation of the nonce-misuse attacks for the CAESAR competition candidates AEGIS, Tiaoxin and MORUS*. GitHub repository. 2017. https://github.com/dkales/caesar-nonce-misuse.

[5] I. Nikolić. *Tiaoxin-346*. Submission to the CAESAR Competition. 2016. http://competitions.cr.yp.to/round3/tiaoxinv21.pdf.

[6] P. Rogaway. "Nonce-Based Symmetric Encryption". In: *Fast Software Encryption – FSE 2004*. Ed. by B. K. Roy and W. Meier. Vol. 3017. LNCS. Springer, 2004, pp. 348–359. DOI: 10.1007/978-3-540-25937-4_22.

[7] P. Rogaway and T. Shrimpton. "A Provable-Security Treatment of the Key-Wrap Problem". In: *Advances in Cryptology – EUROCRYPT 2006*. Ed. by S. Vaudenay. Vol. 4004. LNCS. Springer, 2006, pp. 373–390. DOI: 10.1007/11761679_23.

[8] H. Wu and T. Huang. *The Authenticated Cipher MORUS (v2)*. Submission to the CAESAR Competition. 2016. http://competitions.cr.yp.to/round3/morusv2.pdf.

[9] H. Wu and B. Preneel. *AEGIS: A Fast Authenticated Encryption Algorithm (v1.1)*. Submission to the CAESAR Competition. 2016. http://competitions.cr.yp.to/round3/aegisv11.pdf.