Square Attack on 7-Round Kiasu-BC

Christoph Dobraunig, Maria Eichlseder, and Florian Mendel

IAIK, Graz University of Technology, Austria
christoph.dobraunig@iaik.tugraz.at, maria.eichlseder@iaik.tugraz.at, florian.mendel@iaik.tugraz.at

Abstract. Kiasu-BC is a tweakable block cipher presented within the TWEAKEY framework at AsiaCrypt 2014. Kiasu-BC is almost identical to AES-128, the only difference to AES-128 is the tweak addition, where the 64-bit tweak is xored to the first two rows of every round-key. The security analysis of the designers focuses primarily on related-key related-tweak differential characteristics and meet-in-the-middle attacks. For other attacks, they conclude that the security level of Kiasu-BC is similar to AES-128. In this work, we provide the first third-party analysis of Kiasu-BC. We show that we can mount Square attacks on up to 7-round Kiasu-BC with a complexity of about $2^{48.5}$ encryptions, which improves upon the best published 7-round attacks for AES-128. Furthermore, we show that such attacks are applicable to the round-reduced ΘCB3-like mode of the CAESAR candidate Kiasu≠. To be specific, we show a key-recovery attack on 7-round Kiasu≠ with a complexity of about 2^{82} encryptions.

Keywords: Cryptanalysis · TWEAKEY · Kiasu · Square Attack

1 Introduction

In contrast to standard block ciphers, tweakable block ciphers provide an additional input called tweak. This tweak is usually public and is used to select one specific instance of the block cipher. The concept of tweakable block ciphers was first formalized by Liskov et al. [15,16]. Since then, tweakable block ciphers have proven to be a valuable building block of cryptographic schemes for various applications, like encryption, authentication, or authenticated encryption. For example, several of the authenticated encryption schemes in the ongoing CAESAR competition [19] are based on tweakable block ciphers [8,12,13].

Recently, Jean et al. presented the TWEAKEY framework [10] for designing tweakable block ciphers. The extended version of their paper [11] specifies three instances: Deoxys-BC, Joltik-BC, and Kiasu-BC. Kiasu-BC is a tweakable variant of AES-128, accepting a 64-bit tweak T in addition to the 128-bit key and 128-bit data block. The specification of Kiasu-BC is essentially identical to AES-128, except that T is xored to the first two rows of every round key. Hence, Kiasu-BC exactly matches AES-128 if $T = 0$. This has several advantages. First of all, it allows easy reuse or updates of existing implementations of AES-128. Moreover, the trust of the industry and academia in AES-128 has been steadily growing over the past years and it might be easier in practice to promote the use of AES-128 with slight modifications instead of proposing new tweakable block ciphers. Another advantage of the similarity of Kiasu-BC and AES-128 is that AES-128 has been very thoroughly analyzed due to its prominence and widespread adoption. Since Kiasu-BC corresponds to AES-128 if $T = 0$, existing and also new analysis results for AES-128 directly carry over to Kiasu-BC. However, it is not trivial to determine the effects of the tweak on the security of the design. Therefore, we provide—to the best of our knowledge—the first third-party analysis of Kiasu-BC.

The existing cryptanalysis of Kiasu-BC by its designers [9,11] focuses mainly on meet-in-the-middle attacks and related-key related-tweak differential attacks. The designers argue that the existing meet-in-the-middle attacks for AES-128 also apply to Kiasu-BC. Regarding related-key related-tweak differential characteristics, the designers were able to show that the minimum number of active S-boxes for 7 rounds of Kiasu-BC is 22 and thus, an upper bound for the probability is 2^{-132}. Since this bound is not tight, the designers conclude that Kiasu-BC suffers at most one round security loss compared to AES [9] in the framework of related-key related-tweak differential attacks. For the remaining types of attacks, the designers claim: *"As we keep the original round function and key schedule of AES, we believe that the security level of KIASU-BC against the remaining types of attacks stays the same"* [9]. In Table 1, we have listed some of these remaining attacks. The best-performing attacks that cover 7 rounds of AES-128 fall into the category of impossible differential and meet-in-the-middle attacks. Our goal is to find stronger attacks than these.

Table 1. Excerpt of best attacks on AES-128.

Rounds	Type	Data (CP)	Time	Ref
6	Partial sum	$2^{34.6}$	2^{44}	[6]
7	Partial sum	$2^{128-\varepsilon}$	2^{120}	[6]
7	Collisions	2^{32}	$2^{128-\varepsilon}$	[7]
7	Impossible differential	$2^{112.2}$	$2^{117.2}$ MA	[17]
7	Meet-in-the-middle	2^{80}	2^{123}	[4]
7	Impossible differential	$2^{106.2}$	$2^{110.2}$	[18]
7	Meet-in-the-middle	2^{97}	2^{99}	[5]

MA – memory accesses

All our attacks are based on the Square attack [1]. In the attack, a so-called Λ-set of 256 different plaintexts is observed during the encryption. In the case of AES, it is possible to construct 3-round distinguishers based on the Square property [2,3]. This leads to efficient 6-round key-recovery attacks on AES-128 by prepending 1 round and appending 2 rounds to the 3-round distinguisher [6]. To extend these attacks, we use the additional freedom introduced by the tweak of Kiasu-BC to create a Square-based distinguisher covering 4 rounds. This leads to

7-round attacks on Kiasu-BC (shown in Table 2), which are significantly better than the best published attacks on 7 rounds of AES-128 (see Table 1 for an overview of attacks on AES-128). Furthermore, we show that variants of our Square attack are also applicable to round-reduced variants of an authenticated encryption mode of the CAESAR candidate Kiasu [9]. To be more specific, we target a round-reduced variant of Kiasu$\neq$, which uses 7-round Kiasu-BC in a ΘCB3-like [14] mode of operation. The attacks on round-reduced Kiasu$\neq$ are performed in a nonce-respecting scenario, and also comply with the very low data complexity limits imposed by Kiasu$\neq$.

Table 2. Dedicated attacks on round-reduced Kiasu-BC and Kiasu$\neq$.

Target	Rounds	Type	Data (CP)	Time	Ref
Kiasu-BC	7/10	Square	2^{40}	2^{82}	4.1
	7/10	Square	$2^{43.6}$	$2^{48.5}$	4.2
Kiasu$\neq$	7/10	Square	$2^{28} \times 2^{16}$	2^{82}	5.2

The remainder of the paper is organized as follows. First, we describe the design of Kiasu-BC in Section 2. Afterwards, we construct a 4-round distinguisher based on the Square attack (Section 3), followed by two key-recovery attacks on 7-round Kiasu-BC in Section 4. Next, we demonstrate the applicability of variants of the key-recovery attacks on the mode of operation Kiasu$\neq$ in Section 5. Finally, we conclude in Section 6.

2 Description of Kiasu-BC

The tweakable block cipher Kiasu-BC was introduced as building block of the Kiasu authenticated cipher family [9], a candidate in the CAESAR competition [19]. Kiasu-BC is an instantiation of the TWEAKEY framework [10], a general construction framework for tweakable block ciphers. For each 128-bit key and public 64-bit tweak, Kiasu-BC defines a 128-bit permutation.

Kiasu-BC is essentially identical to AES, except that the 64-bit tweak value is xored to the state in each round after the round-key addition. Thus, like for AES, the 128-bit Kiasu-BC state S is represented as a 4×4 matrix of bytes, labeled $x_0, \ldots, x_{15}$:

$$S = \begin{array}{|c|c|c|c|} \hline x_0 & x_4 & x_8 & x_{12} \\ \hline x_1 & x_5 & x_9 & x_{13} \\ \hline x_2 & x_6 & x_{10} & x_{14} \\ \hline x_3 & x_7 & x_{11} & x_{15} \\ \hline \end{array} \, .$$

In each of Kiasu-BC's 10 rounds, the round operations $\mathsf{SubBytes}$, $\mathsf{ShiftRows}$, $\mathsf{MixColumns}$ and $\mathsf{AddRoundTweakey}$ are applied to the state in turn. Except for $\mathsf{AddRoundTweakey}$, they are identical to the AES round operations:

– $\mathsf{SubBytes}$: Applies the 8-bit AES S-box $\mathcal{S}$ to each of the 16 state bytes.
– $\mathsf{ShiftRows}$: Rotates row i of the state, $0 \leq i \leq 3$, by i bytes to the left.
– $\mathsf{MixColumns}$: Multiplies each byte column of the state by the MDS-matrix M over $\mathbb{K} = \mathbb{F}_2[\alpha]/(\alpha^8 + \alpha^4 + \alpha^3 + \alpha + 1)$,

$$M = \begin{pmatrix} \alpha & \alpha+1 & 1 & 1 \\ 1 & \alpha & \alpha+1 & 1 \\ 1 & 1 & \alpha & \alpha+1 \\ \alpha+1 & 1 & 1 & \alpha \end{pmatrix} = \begin{pmatrix} \mathsf{02} & \mathsf{03} & \mathsf{01} & \mathsf{01} \\ \mathsf{01} & \mathsf{02} & \mathsf{03} & \mathsf{01} \\ \mathsf{01} & \mathsf{01} & \mathsf{02} & \mathsf{03} \\ \mathsf{03} & \mathsf{01} & \mathsf{01} & \mathsf{02} \end{pmatrix}$$

– $\mathsf{AddRoundTweakey}$: In round i, xors the 128-bit round key RK_i and the tweak T to the state, where

$$\mathsf{RK}_i = \begin{array}{|c|c|c|c|}\hline \mathsf{RK}_{i,0} & \mathsf{RK}_{i,4} & \mathsf{RK}_{i,8} & \mathsf{RK}_{i,12} \\ \hline \mathsf{RK}_{i,1} & \mathsf{RK}_{i,5} & \mathsf{RK}_{i,9} & \mathsf{RK}_{i,13} \\ \hline \mathsf{RK}_{i,2} & \mathsf{RK}_{i,6} & \mathsf{RK}_{i,10} & \mathsf{RK}_{i,14} \\ \hline \mathsf{RK}_{i,3} & \mathsf{RK}_{i,7} & \mathsf{RK}_{i,11} & \mathsf{RK}_{i,15} \\ \hline \end{array}\,, \qquad T = \begin{array}{|c|c|c|c|}\hline T_0 & T_2 & T_4 & T_6 \\ \hline T_1 & T_3 & T_5 & T_7 \\ \hline & & & \\ \hline & & & \\ \hline \end{array}\,.$$

We omit the details of the AES key schedule that derives the round subkeys RK_i from the key K, since they are not relevant for our attack. Note that there is no tweak schedule, i.e., the same tweak T is xored in each round. So for the all-zero tweak $T = 0$, Kiasu-BC is equivalent to AES-128.

To refer to intermediate states of Kiasu-BC, we denote by S_i the state after i rounds: $S_0 = P \oplus T \oplus \mathsf{RK}_0$, S_1, $\ldots$, $S_{10} = C$. In addition, the state after $\mathsf{SubBytes}$ of round i is denoted S_i^{SB}, after $\mathsf{ShiftRows}$ S_i^{SR}, after $\mathsf{MixColumns}$ S_i^{MC}, and after $\mathsf{AddRoundTweakey}$ $S_i^{\mathsf{AK}} = S_i$. So the states of full-round Kiasu-BC are

$$\begin{aligned} &P \xrightarrow{\mathsf{AK}} S_0 \xrightarrow{\mathsf{SB}} S_1^{\mathsf{SB}} \xrightarrow{\mathsf{SR}} S_1^{\mathsf{SR}} \xrightarrow{\mathsf{MC}} S_1^{\mathsf{MC}} \xrightarrow{\mathsf{AK}} S_1 \\ &\qquad\vdots \\ &S_9 \xrightarrow{\mathsf{SB}} S_{10}^{\mathsf{SB}} \xrightarrow{\mathsf{SR}} S_{10}^{\mathsf{SR}} \xrightarrow{\mathsf{AK}} S_{10} = C\ . \end{aligned}$$

3 Distinguisher for 4 rounds of Kiasu-BC

The distinguisher presented in this section is based on the Square attack. This attack, originally demonstrated for the block cipher Square [1], is also applicable to AES [2, 3]. As in the Square attack on AES, we will observe a Λ-set of 256 different plaintexts through the encryption. By making use of the tweak input of Kiasu-BC, we show that a distinguisher for 4 rounds can be created. This is one round more than the distinguisher used in the Square attack on AES. Before giving the distinguisher, we recall the effect of the round functions of AES on Λ-sets.

3.1 Preliminaries

For the Square attack, we will make statements about the 256 values for single byte positions x_i of a Λ-set. We index the individual byte value of byte position i in Λ-set element k as $x_i[k]$, where the index k is in the range from 0 to 255. We call a byte of a Λ-set active (`A`) if it takes all possible 256 values; constant (`C`) if all 256 values are equal; balanced (`B`) if the sum of all 256 values is 0; or unknown (`?`) if we cannot make any statements about the 256 values for this byte position.

SubBytes. SubBytes affects each byte of the state individually. Therefore, we can put our focus on the effects of the S-box on our four different byte states: active, constant, balanced, and unknown. The AES S-box is a permutation. Hence, if the input of the S-box iterates over all 256 possible values, then so will the output. Thus, an active byte remains active after SubBytes. Since the AES S-box is deterministic, a certain value at the input of the S-box will always map to the same value at the output. This means a constant byte remains constant after SubBytes. However, a balanced byte becomes unknown, because the S-box is non-linear. An unknown byte remains, of course, unknown.

ShiftRows. The ShiftRows operation works on byte-level. To be more concrete, it simply reorders the bytes of the state. Hence, our statements about the bytes remain the same, just the position differs after ShiftRows.

MixColumns. MixColumns is a linear transformation that mixes the single bytes of one column. Clearly, an all-constant input set will be mapped to an all-constant output set. Furthermore, if at least one of the input byte positions of the set is unknown, the entire output will be unknown.

Since MixColumns is based on an MDS matrix, it has a branch number of 5. This implies that if two input columns differ only in one byte, the output will differ in all 4 bytes. In particular, if the 4 input byte positions of a set are all constant except for one active byte, then all output bytes will be active. (Assume that one byte is not active, but takes one particular value twice. The corresponding pair of inputs will have a difference in only 1 input byte and at most 3 output bytes, violating the branch number property.) The same reasoning also clearly applies for the inverse operation of MixColumns.

AddRoundTweakey. Here, the specific round key as well as the tweak are xored to the state. Our attacks are performed in the single-key setting, so each key byte is constant. This means that an active byte of the state remains active, a constant byte constant, a balanced byte balanced (since the constant key is added an even number of times and cancels out), and an unknown byte remains unknown.

The situation changes if we take a look at the tweak addition. For the distinguisher, we want to use Λ-sets where one byte of the tweak is active, so we have to consider the following situations. The xor of an active byte with an active byte definitely results in a balanced byte. If the tweak byte as well as the state byte are active and $T_i[k] \oplus x_i[k] = c$ for each k, the byte gets constant. The xor of an active byte with a balanced byte results in a balanced byte.

3.2 The 4-round distinguisher

The distinguisher used in the Square attack against AES [2, 3] spans over 3 rounds. It starts with a Λ-set that is active in one byte of the plaintext and constant in the rest of the state. The distinguisher ends after the key addition of the third round with an all-balanced state. By introducing an active tweak byte, we are able to extend the distinguisher by one round. However, the condition we get after round 4 is slightly more difficult to exploit (see Fig. 1).

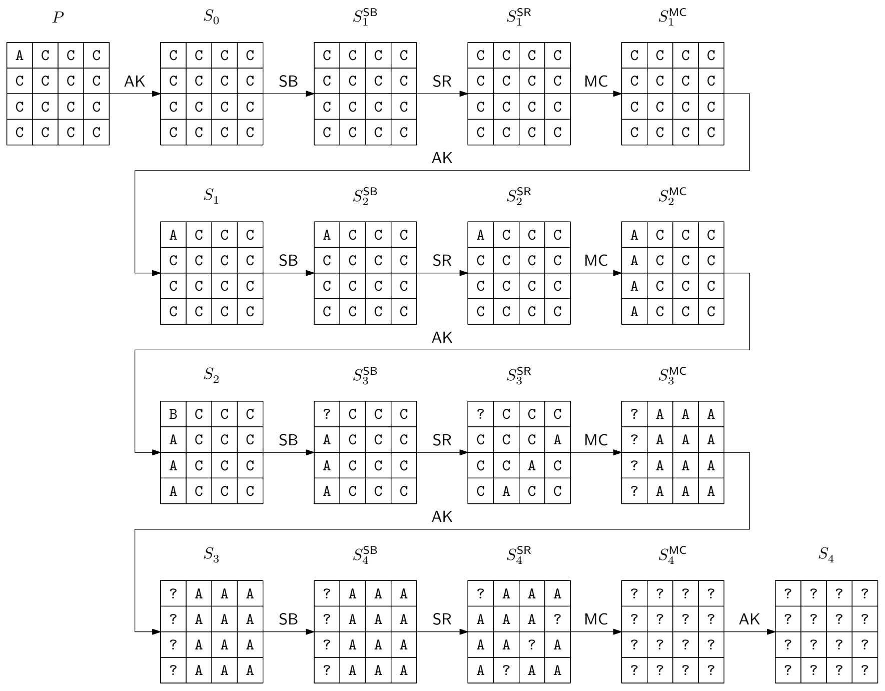

Fig. 1. Distinguisher for 4 rounds of Kiasu-BC.

As shown in Fig. 1, we start with a Λ-set of 256 plaintexts P, where one byte is active and the others remain constant. Additionally, we require that byte T_0 of the tweak is active as well. Since always the same tweak is xored to every round key, every resulting round key xored with the tweak can be describe as a Λ-set that is active at byte 0 and constant in the rest of the bytes. The tweak and plaintext values have to be chosen in a way that the xor of the tweak and the plaintext is constant. For instance, $T_0[k]$ can always be chosen to be equal to the first byte of the plaintext $x_0[k]$ for all 256 values of k. In this way, it is ensured that state S_0 is constant at every byte position. The state remains constant until S_1, where byte x_0 becomes active again due to the addition of the tweak.

The second round of our distinguisher for Kiasu-BC corresponds to the first round of the distinguisher used in the AES Square attack, except for the addition of the active tweak byte at the end. Since SubBytes and ShiftRows affect neither active nor constant bytes, we get to state S_2^{SR}, where still only the byte at position 0 remains active. The rest of the state is still constant. The next MixColumns operation leads to an active column in state S_2^{MC}. In contrast to the first tweak addition, the tweak addition at the end of round 2 leads to a balanced byte at position 0. We get a balanced byte here, because we cannot make any assumption on the concrete ordering of the 256 values of x_0 of state S_2^{MC}.

In the third round, we have one balanced byte before SubBytes. This byte becomes unknown after the S-box application. The ShiftRows operation shifts the active bytes away from the first column. So we have at state S_3^{SR} one unknown, and three constant bytes in the first column and one active, and 3 constant bytes in every other column. This leads to one completely unknown first column, and three completely active columns in state S_3^{MC}. The next tweak addition does not change anything.

For the fourth round, we only go with active or unknown bytes through the S-box layer, thus SubBytes does not influence our knowledge about the Λ-set at this point. ShiftRows shifts one unknown byte to every column, so we get a completely unknown state S_4^{MC} if we only limit our view to single byte positions. Hence, we have to take a closer look at the MixColumns operation. To do so, we represent the bytes of S_4^{SR} as x_i and the bytes of S_4^{MC} as y_i. Now, let us take a look at what happens if we xor y_1 with y_2:

$$\begin{aligned} y_1 \oplus y_2 &= \mathtt{01}\cdot x_0 \oplus \mathtt{02}\cdot x_1 \oplus \mathtt{03}\cdot x_2 \oplus \mathtt{01}\cdot x_3 \oplus \mathtt{01}\cdot x_0 \oplus \mathtt{01}\cdot x_1 \oplus \mathtt{02}\cdot x_2 \oplus \mathtt{03}\cdot x_3 \\ &= \mathtt{03}\cdot x_1 \oplus \mathtt{01}\cdot x_2 \oplus \mathtt{02}\cdot x_3 \end{aligned} \tag{1}$$

As shown in (1), x_0 cancels and thus does not influence $y_1 \oplus y_2$. In the first column of S_4^{SR}, x_0 is the only byte which is unknown. The rest of the bytes are active. Since (1) only contains active coefficients, $y_1 \oplus y_2$ is balanced. The next key and tweak addition is an addition with constant bytes. This addition with constant values does not influence the balanced property and therefore, also the xor of byte 1 and 2 of state S_4 is balanced.

4 Attacking 7 Rounds of Kiasu-BC

For attacking 7 rounds of Kiasu-BC, we extend the distinguisher by one round in the backward and two rounds in the forward direction. At first we present a basic version of the attack. Then, we improve the attack by using partial sums in a similar way as Ferguson et al. [6].

4.1 Basic Square attack

The key-recovery attack is based on a set of plaintexts with differences only on one of the diagonals of the state, combined with a set of tweaks with differences only in the top left byte T_0. Fig. 2 shows the trail we use to attack 7 rounds of Kiasu-BC, where rounds 2 to 5 correspond to the distinguisher explained in Section 3. To perform this attack, we first collect the encryption of all 2^{32} plaintexts P where the diagonal bytes $(x_0, x_5, x_{10}, x_{15})$ loop through all possible values, whereas the remaining 12 bytes are fixed to some constant. Each of these plaintexts is encrypted under all 2^8 possible tweaks where all bytes except T_0 are fixed to some constant, and T_0 loops through all values. Thus, in total, we require the ciphertext for $2^8 \cdot 2^{32} = 2^{40}$ plaintext-tweak combinations.

Building Λ-sets. Next, we want to group this data into suitable Λ-sets, so that the previously introduced distinguisher can be applied to state S_1. This has to be done separately for each possible key guess of the 32 key bits $\mathsf{RK}_{0,0}$, $\mathsf{RK}_{0,5}$, $\mathsf{RK}_{0,10}$, and $\mathsf{RK}_{0,15}$, which determine the values of the first column of state S_1^{MC}. What we want to achieve is that this first column has only 1 active byte in x_0, and that this activity is canceled by $\mathsf{AddRoundTweakey}$. Thus, we can fix the 3 constant bytes x_1, x_2, x_3 in S_1^{MC} to some arbitrary value, and set $x_0 = T_0$ for each of the 2^8 tweaks. If we decrypt these 2^8 set elements by 1 round, $\mathsf{MixColumns}$ will produce 4 active S-boxes in S_1^{SR}, which will be shifted to active S-boxes in x_0, x_5, x_{10}, x_{15} in state S_0. Depending on the different tweak bytes T_0 and the current key guess for the partial first-round key RK_0, we get a Λ-set of 2^8 plaintexts. We can repeat this procedure for a few different constant values in S_1^{MC} in order to build 16 Λ-sets for each of the 2^{32} key guesses of RK_0. For the correct key guess, all 16 Λ-sets will follow the 4-round distinguisher from Section 3.

Applying the distinguisher. We now want to partially decrypt all ciphertexts of each Λ-set back to state S_5, in order to verify the distinguishing property. Remember that we are interested in computing the xor sum $y_1 \oplus y_2$ of each Λ-set, marked in black (y_1) and gray (y_2) in Fig. 2. To do so, we have to calculate all intermediate values marked in black and gray in Fig. 2. We can do this for the black and the gray trail separately, requiring to guess 5 byte of key material for each trail. Note that we swapped the order of $\mathsf{MixColumns}$ and $\mathsf{AddRoundTweakey}$ in round 6, so that we only have to guess 1 byte of an equivalent round-key $\mathsf{RK}_6^{\mathrm{equ}} = \mathsf{MC}^{-1}(\mathsf{RK})$, rather than 4 bytes of the original RK_6.

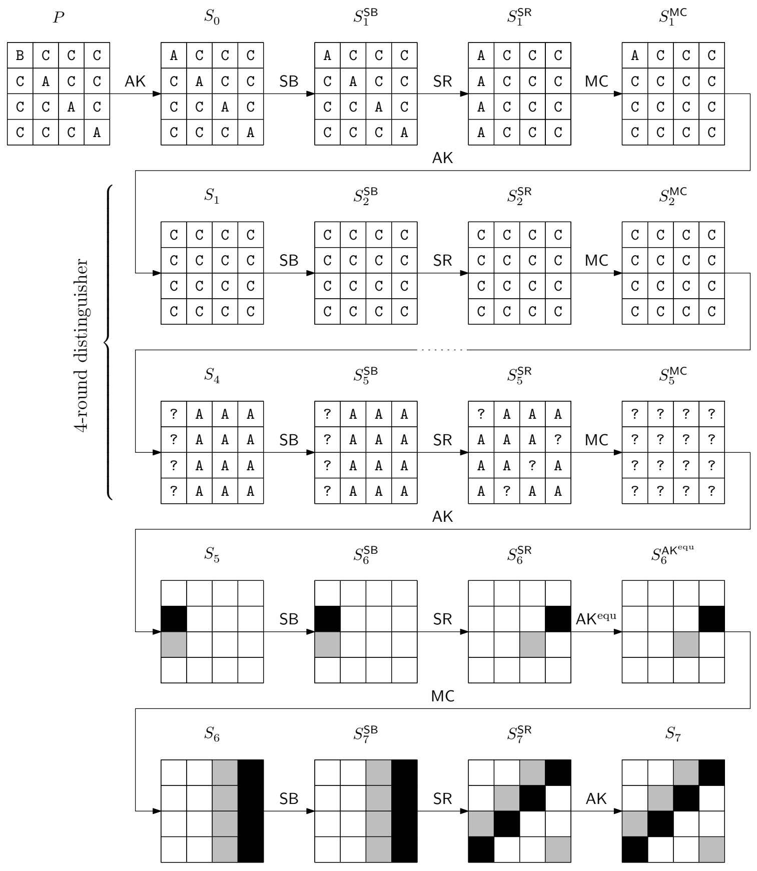

Fig. 2. Square attack for 7 rounds of Kiasu-BC.

We end up building two lists L_1 and L_2 (per key guess of RK_0). Each list has 2^{40} entries of 16-byte length each. For L_1, each entry represents the 16 xor sums of y_1 that result when decrypting the 16 Λ-sets for one guess of $\mathsf{RK}_{7,3}$, $\mathsf{RK}_{7,6}$, $\mathsf{RK}_{7,9}$, $\mathsf{RK}_{7,12}$, and $\mathsf{RK}^{\text{equ}}_{6,13}$. In the case of L_2, each entry represents the 16 xor sums of y_2 that result when decrypting the 16 Λ-sets for one guess of $\mathsf{RK}_{7,2}$, $\mathsf{RK}_{7,5}$, $\mathsf{RK}_{7,8}$, $\mathsf{RK}_{7,15}$, and $\mathsf{RK}^{\text{equ}}_{6,10}$.

As explained in Section 3, y_1 and y_2 of state S_5 sum to 0 for the correct key guess. Hence, we have to search for matching 16-byte entries between lists L_1 and L_2. A match indicates a key guess combination for the 10 guessed bytes of $\mathsf{RK}^{\text{equ}}_6$, RK_7 that satisfies the distinguishing property for all 16 Λ-sets of one key guess for 4 bytes of RK_0; that is, a candidate for 14 bytes (or 112 bits) of key material for the correct key. The probability that a wrong key fulfills our 16-byte distinguisher is 2^{-128} (distinguishing property is the zero value for 128 bits, all other values reveal wrong keys). Therefore, we expect that only one candidate for the correct key bytes remains.

Attack complexity. To determine the overall complexity of this attack, we first take a look at the complexity per first-round key guess (guess of $\mathsf{RK}_{0,0}$, $\mathsf{RK}_{0,5}$, $\mathsf{RK}_{0,10}$, and $\mathsf{RK}_{0,15}$). To generate the 16 Λ-sets, we have to partially decrypt $16 \cdot 2^8$ plaintext-tweakey combinations for one round, for one column of the state. This will allow us to select suitable Λ-sets from the 2^{40} chosen-plaintext queries encrypted under the target key. Then, we have to create our two lists L_1 and L_2. For creating one list, we have to decrypt $2^4 \cdot 2^8$ ciphertexts for 2^{40} key guesses 2 rounds back to one byte at S_5. Since we decrypt for 2 rounds to one byte, we only have to look at one column of the state. Hence, we estimate the costs for such a partial decryption with half a Kiasu-BC round. So, creating one list has approximately the complexity of $2^4 \cdot 2^8 \cdot 2^{40} = 2^{52}$ half-round decryptions, which corresponds to less than 2^{49} 7-round Kiasu-BC decryptions. For creating both lists, we require about 2^{50} 7-round Kiasu-BC decryptions. This complexity dominates both the complexity of 2^{12} one-round encryptions for creating the 16 Λ-sets and the complexity for finding a match between the two lists, which is approximately $40 \cdot 2^{40}$ comparison operations for sorting one list and $40 \cdot 2^{40}$ memory look-ups for finding a match.

Since we have to build the two lists for each of the 2^{32} first-round key guesses, we end up having a total attack complexity of 2^{82} 7-round Kiasu-BC encryptions. For carrying out this attack, we have to query $2^8 \cdot 2^{32} = 2^{40}$ chosen plaintexts. In addition to the plaintext-ciphertext pairs, we have to store our two lists L_1 and L_2. One entry of the lists corresponds to the memory complexity of storing one plaintext. Thus, we have an additional memory requirement of roughly 2^{41} Kiasu-BC states.

4.2 Improvements using partial sums

Ferguson et al. [6] showed that the complexity of the Square attack on AES can be significantly improved by using the partial sum technique. Their first observation is that for AES, the effort of guessing the 32 bits of RK_0 can be traded for summing over larger sets (of all 2^{32} plaintexts, rather than only 2^8 Λ-set messages), thus reducing the complexity by a factor of 2^8. Then, as a second improvement, the increased number of operations necessary for evaluating the distinguisher can be rearranged into partial sums to significantly cut down the computational complexity. In this section, we will show that a similar reasoning applies to Kiasu-BC, and that the techniques of Ferguson et al. [6] can be adapted to improve the complexity of the attack on 7-round Kiasu-BC significantly.

Summing all messages. In the basic attack, we had to guess 4 bytes of the first round key RK_0 in order to select a suitable Λ-set of 2^8 plaintext-tweak combinations and apply the distinguisher. For such a Λ-set, which is characterized by a single active byte x_0 in state S_1^{MC} and a constant difference between this byte and tweak byte T_0 (e.g., $x_0 = T_0$), we know that in S_5, the values $y_1 \oplus y_2$ sum to 0. Clearly, the same distinguishing property also applies if we sum not just over one, but over several Λ-sets.

Now consider again our set of $2^8 \cdot 2^{32}$ plaintext-tweak combinations. This set can actually be grouped into $2^8 \cdot 2^{24}$ Λ-sets as follows. For every value of T_0, the state bytes x_0, x_1, x_2, x_3 in state S_1^{MC} take all 2^{32} values. Therefore, for each of the 2^{24} fixed constant values of x_1, x_2, x_3 and each fixed value $x_0 \oplus T_0$ in state S_1^{MC}, we can find exactly 2^8 plaintext-tweak combinations that map to this state, where x_1, x_2, x_3 and $x_0 \oplus T_0$ are constant. Each of these 2^8 plaintext-tweak combinations fulfills our conditions for a Λ-set. Thus, if we sum over all plaintext-tweak combinations, we actually sum over many Λ-sets, so the distinguishing property for $y_1 \oplus y_2$ will apply – and we do not have to guess the round key RK_0 in order to evaluate it. In other words, we can trade guessing the 32 key bits of RK_0 for summing over 2^{40} instead of 2^8 messages. Unfortunately, in contrast to the original attack on AES [6], this first improvement described so far does not, by itself, decrease the attack complexity, since we have to sum over all values of T_0. However, as we will show next, this modified distinguisher can be evaluated in an optimized way by reorganizing the order of summation.

Adapting the distinguisher. To evaluate the distinguisher, we now need to decrypt our 2^{40} ciphertexts back to y_1 and y_2. To identify valid key candidates, we calculate the sum in y_1 for each key guess of $\mathsf{RK}_{7,3}$, $\mathsf{RK}_{7,6}$, $\mathsf{RK}_{7,9}$, $\mathsf{RK}_{7,12}$, and $\mathsf{RK}_{6,13}$, storing the result in L_1 (indexed by the key guess); and we do the same for y_2 in L_2, based on all guesses of $\mathsf{RK}_{7,2}$, $\mathsf{RK}_{7,5}$, $\mathsf{RK}_{7,8}$, $\mathsf{RK}_{7,15}$, and $\mathsf{RK}_{6,10}$. Since we guess in total 10 bytes of key material, a 1-byte distinguisher is not enough to filter all wrong key guesses. Hence, we repeat the whole procedure for a total of 12 collections (of 2^{40} ciphertexts each), so that L_1 and L_2 are in the end populated with 12-byte entries (and indexed by 5-byte key guesses). We expect only one 12-byte match between L_1 and L_2, providing us with the correct 10 bytes of key material.

We now want to optimize the costs for calculating the entries of L_1 and L_2, which dominate the overall runtime by making use of the partial-sum technique described by Ferguson et al. [6]. They show that the cost for computing the 2^{40} sums (for each key guess) of one byte located 2 AES rounds before the end (similar to our case, y_1 or y_2 of State S_5), using 2^{32} ciphertexts, can be reduced to approximately 2^{50} S-box applications. Assuming that one encryption under a new key is equivalent to 2^8 S-box applications, the overall cost is only about 2^{42} encryptions. In contrast to the original attack, we actually want to sum over 2^{40} values, and additionally have to consider the tweak input. However, it turns out that the original partial-sum technique can be adapted to allow this with no significant computational overhead.

First, observe that in each $\mathsf{AddRoundTweakey}$ step, the different values of T_0 only influence the first byte x_0 of the state; and in the $\mathsf{AddRoundTweakey}^{\text{equ}}$ step that we apply in round 6, T_0 modifies the equivalent round key of the first column (state bytes x_0, x_1, x_2, x_3). As illustrated in Fig. 2, neither L_1 nor L_2 depend on these state bytes, so we do not need to know T_0 in order to partially decrypt. Second, note that for building L_1 (or L_2), we are only interested in 32 bits of each of the 2^{40} encrypted messages (per collection). Thus, instead of decrypting each message with each key guess, we can count how often each possible 32-bit value occurs among the encrypted messages, and then only decrypt based on each 32-bit value once. Furthermore, since the effects of two occurrences of the same 32-bit value will simply cancel out in the final xor-sum, it is sufficient to count occurrences modulo 2. We can store the counters in a 2^{32}-bit vector $\delta^{\mathrm{cccc}} = (\delta_0^{\mathrm{cccc}}, \dots, \delta_{2^{32}-1}^{\mathrm{cccc}})$, indexed by the possible values $x = x_0 \| x_1 \| x_2 \| x_3$.

Equipped with these two observations, we can now directly apply Ferguson et al.'s partial-sum technique, which we summarize below.

Ferguson et al.'s partial sums [6]. Consider the byte y_1 we need to evaluate for one entry of L_1, i.e., the sum over the 2^{40} messages of one collection. If we denote the 4 relevant (black) ciphertext bytes of message i in state S_7 by $c_{i,0}, \dots, c_{i,3}$ and the 5 guessed round-key bytes (after xoring the known tweak) by $k_0, \dots, k_4$, and summarize the inverse SubBytes in round 7 and the constant multiplications by MixColumns in round 6 in the bytewise functions $\mathcal{S}_0, \dots, \mathcal{S}_3$, then the value we want to compute is

$$\begin{aligned}\sigma &= \bigoplus_{i=0}^{2^{40}-1} \mathcal{S}^{-1}[\mathcal{S}_0[c_{i,0} \oplus k_0] \oplus \mathcal{S}_1[c_{i,1} \oplus k_1] \oplus \mathcal{S}_2[c_{i,2} \oplus k_2] \oplus \mathcal{S}_3[c_{i,3} \oplus k_3] \oplus k_4] \\ &= \bigoplus_{x=0}^{2^{32}-1} \delta_x^{\mathrm{cccc}} \cdot \mathcal{S}^{-1}[\mathcal{S}_0[x_0 \oplus k_0] \oplus \mathcal{S}_1[x_1 \oplus k_1] \oplus \mathcal{S}_2[x_2 \oplus k_2] \oplus \mathcal{S}_3[x_3 \oplus k_3] \oplus k_4].\end{aligned}$$

To optimize this computation, we first count for every key guess of k_0 and k_1 the modulo-2 frequency of the values $(\mathcal{S}_0[c_{i,0} \oplus k_0] \oplus \mathcal{S}_1[c_{i,1} \oplus k_1], c_{i,2}, c_{i,3})$ and store it in the 2^{24}-bit vector δ^{scc}. This vector can easily be computed from δ^{cccc} as

$$\delta_{x_0,x_1,x_2}^{\mathrm{scc}} = \bigoplus_{s=0}^{2^8-1} \delta_{s,\mathcal{S}_1^{-1}[x_0 \oplus \mathcal{S}_0[s \oplus k_0]] \oplus k_1, x_1, x_2}^{\mathrm{cccc}} . \tag{2}$$

Similarly, after guessing k_2 and subsequently k_3, we can compute the frequency δ^{sc} of $(\mathcal{S}_0[c_{i,0} \oplus k_0] \oplus \mathcal{S}_1[c_{i,1} \oplus k_1] \oplus \mathcal{S}_2[c_{i,2} \oplus k_2], c_{i,3})$ (2^{16} entries) and then δ^{s} of $(\mathcal{S}_0[c_{i,0} \oplus k_0] \oplus \mathcal{S}_1[c_{i,1} \oplus k_1] \oplus \mathcal{S}_2[c_{i,2} \oplus k_2] \oplus \mathcal{S}_3[c_{i,3} \oplus k_3])$ (2^8 entries) via

$$\delta_{x_0,x_1}^{\mathrm{sc}} = \bigoplus_{s=0}^{2^8-1} \delta_{s,\mathcal{S}_2^{-1}[x_0 \oplus s] \oplus k_2, x_1}^{\mathrm{scc}} , \tag{3}$$

$$\delta_{x_0}^{\mathrm{s}} = \bigoplus_{s=0}^{2^8-1} \delta_{s,\mathcal{S}_3^{-1}[x_0 \oplus s] \oplus k_3}^{\mathrm{sc}} . \tag{4}$$

Finally, we guess k_4 and compute the desired result byte via

$$\sigma = \bigoplus_{s=0}^{2^8-1} \delta_s^{\mathrm{s}} \cdot \mathcal{S}^{-1}[s \oplus k_4]. \tag{5}$$

The same procedure can be applied to compute the entries of L_2, and needs to be repeated for each of the 12 collections. Afterwards, L_1 and L_2 can be sorted and matched as before to identify the correct partial key for 10 bytes of key material. The remaining 6 bytes of key information can be recovered with a brute-force approach.

Overall complexity. The data complexity for the improved attack is $12 \cdot 2^{40} \approx 2^{43.6}$ chosen plaintext-tweak combinations. Per list and collection, we have the following complexity. The original 2^{32}-bit vector δ^{cccc} can be constructed with negligible overhead to each chosen-plaintext query. The 2^{24}-bit vector δ^{scc} is computed for 2^{16} key guesses, and requires $2 \cdot 2^8 \cdot 2^{24} = 2^{33}$ S-box lookups, so the computations of (2) contribute 2^{49} S-box lookups per list and collection. Similarly, computations (3), (4) and (5) contribute 2^{48} S-box lookups each. Overall, computing lists L_1 and L_2 require $2 \cdot 12 \cdot (2^{49} + 3 \cdot 2^{48}) \approx 2^{54.9}$ S-box lookups, or roughly $2^{46.9}$ 7-round encryptions.

Sorting the 2^{40} entries of L_1 and L_2 can be implemented, for example, with less than $40 \cdot 2^{40} \approx 2^{45.3}$ comparisons (worst-case) and $2 \cdot 2^{40.1}$ Kiasu-BC states of memory per list via MergeSort, or a total of $2^{46.3}$ comparisons and $2^{41.7}$ memory for both lists. Finding all matches between the sorted lists takes a negligible $2 \cdot 2^{40}$ comparisons (worst-case).

We expect to find only one match, and guessing the remaining 6 bytes of key information takes, in the worst case, 2^{48} encryptions (assuming that the known 10 bytes of key information can be combined efficiently). In total, the worst-case attack complexity is about $2^{48.5}$ 7-round Kiasu-BC encryptions, and requires about $2^{41.7}$ Kiasu-BC states of memory, and $2^{43.6}$ chosen-plaintext-tweak queries.

5 Application to Authenticated Cipher Kiasu$\neq$

In this section, we show that variants of the previously presented Square attacks are applicable when Kiasu-BC is used in a ΘCB3-like [14] mode of operation. To be specific, we demonstrate the feasibility of a variant of the attack presented in the previous section on Kiasu$\neq$. Kiasu$\neq$ is one of two proposed modes of the CAESAR candidate Kiasu [9], which only claims security when used in a nonce-respecting way. Thus, the attacks presented in this section follow this restriction and never require the nonce to be equal for queries on the encryption oracle. Before describing the attack, we give a short description of Kiasu$\neq$.

5.1 Description of Kiasu$\neq$

Fig. 3 shows the plaintext processing part of the authenticated encryption scheme Kiasu$\neq$. Here, each plaintext block P_i is encrypted with the help of Kiasu-BC using always a different value for its tweak. The tweak value is constructed by concatenating a 3-bit 0, the 32-bit nonce N and a 29-bit value representing the index i of the plaintext block P_i that is encrypted. To generate the tag T, the sum of the plaintext blocks is encrypted and xored with Auth, which is derived from processing the authenticated data.

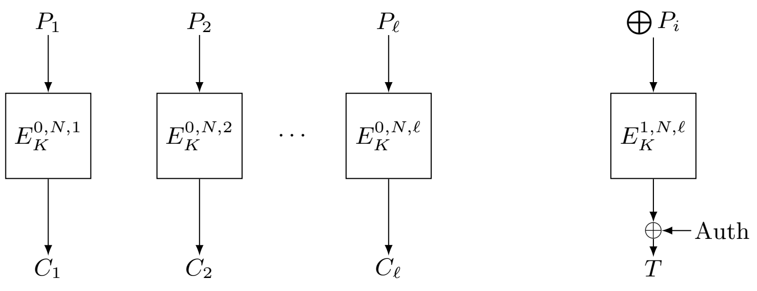

Fig. 3. Plaintext processing for the nonce-respecting mode Kiasu≠ for a multiple of the block length.

5.2 A Key-Recovery Attack on Round-Reduced Kiasu≠

Our attack targets the encryption of the plaintexts blocks. For the attack to be carried out, we need an encryption oracle that encrypts plaintexts chosen by the attacker. We use the block counter to iterate over the tweak byte T_7 to construct our Λ-sets. Since the least significant byte of the block counter is xored to byte 13 of the state, we have to use a slightly different distinguisher, which is shown in Fig. 4. Similar to the attacks presented in Section 4, we prepend one round to the distinguisher and append two rounds. Then, we can apply a slight modification of the Square attack described in Section 4.1.

The attack of Section 4.1 can be partitioned in two phases. The first one is the generation of 16 Λ-sets under a specific guess of 32-bits of RK_0, the second part is the evaluation of the Λ-sets to see if the distinguishing property holds for partial guesses of RK_6 and RK_7. While this evaluation of the Λ-sets works equivalent as in Section 4.1 for the attack on Kiasu≠, we have to change the way we built our Λ-sets. For building the Λ-sets, the attack of Section 4.1 uses the same 2^8 tweak values for every Λ-set. This is no longer an option, since the attacks on round-reduced Kiasu≠ are performed in a nonce-respecting setting. Therefore, we have to build each Λ-set using different tweak values and respecting the data limits of Kiasu≠, which limit the number of encrypted blocks per message to 2^{29}, and the total number of encrypted messages to 2^{32}. Next, we will describe how to select suitable plaintexts to obtain Λ-sets under these constraints.

Observe that for a single multi-block plaintext message, the tweaks used for encrypting the individual plaintext blocks will be constant in the first 35 bits, where 32 bits represent the nonce value. Dependent on the attack model, the nonce may be known before we make an encryption query (e.g., it is implemented as a counter, to avoid collisions of the very short nonces), or the oracle picks a random nonce. Note that one byte of the nonce at tweak position T_1 influences

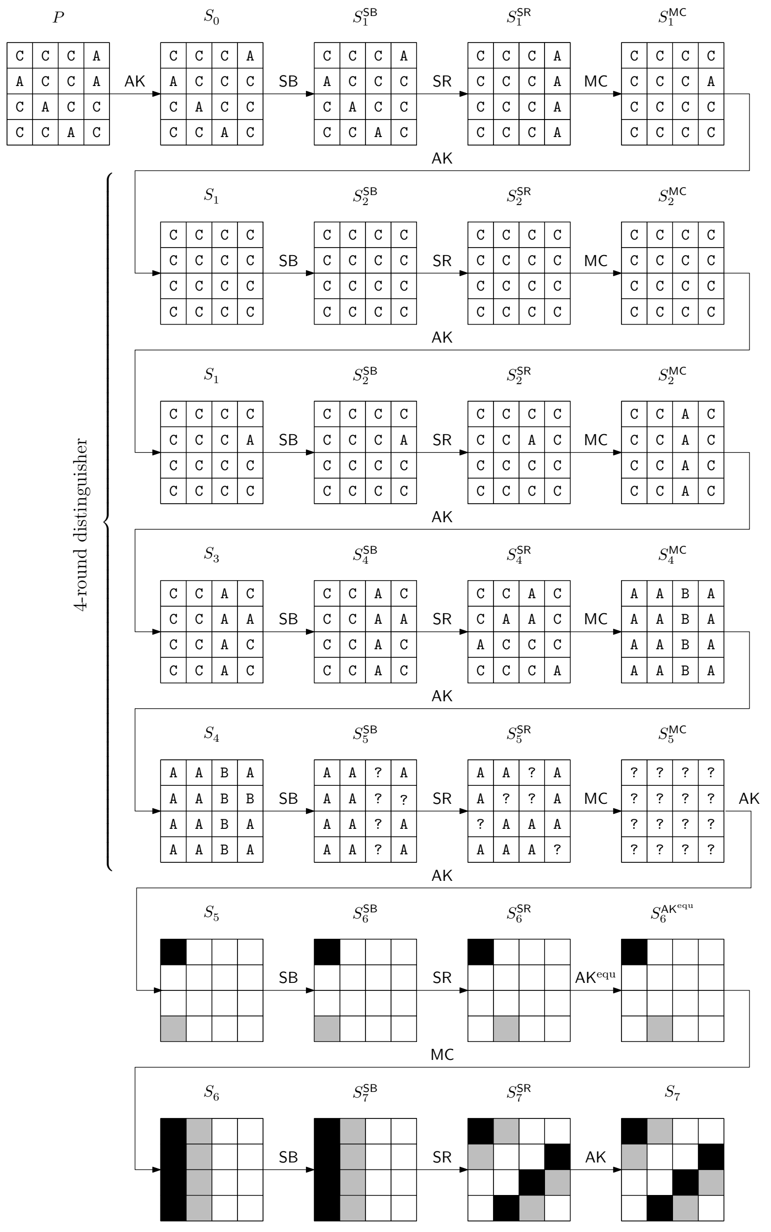

Fig. 4. Attack for 7 rounds of Kiasu≠.

our key guess at $\mathsf{RK}_{0,1}$ in the upcoming attack. Hence, for sake of simplicity, we assume that the nonce value is known before we make each encryption query (we discuss the case of unpredictable nonces at the end of this section). The remaining bits of the tweak represent a 29-bit block counter and are always known in advance. In our attack we want to use the least significant 8 counter bits in T_7 for the active tweak byte. Since the counter starts with a value of 1, we actually can only start building Λ-sets from block 256 on. So the first Λ-set includes blocks $256, \ldots, 511$, i.e., $T_6 = 1$ and T_7 is active. Now, we need to define suitable plaintext blocks to query, so that the ciphertext blocks of one 511-block message will allow us to evaluate the distinguisher.

Let p_i denote the individual state bytes of the plaintext, x_i the bytes of S_1^{MC}, and z_i the bytes of the state right after adding the tweak (but before adding the round key). We start by choosing some arbitrary constant value for the bytes $z_{12}, z_{13}, z_{14}, z_{15}$. Then, we apply the inverse tweak-addition to obtain $x_{12}, \ldots, x_{15}$, which will add a constant value of $T_6 = 1$ to z_{12}, and an active $T_7 = 0, \ldots, 255$ to z_{13}. The inverse first round will map this column to some set of states with 4 active bytes in S_0. For one key guess of $\mathsf{RK}_{0,1}$, $\mathsf{RK}_{0,6}$, $\mathsf{RK}_{0,11}$, and $\mathsf{RK}_{0,12}$, we obtain a set of 256 values for $(p_1, p_6, p_{11}, p_{12})$. The only other active byte, p_{13}, needs to be chosen so that the difference $p_{13} \oplus T_7$ is fixed, e.g., by setting $p_{13} = T_7$. The rest of the state can be chosen as some arbitrary constant. The resulting plaintext blocks have to be encrypted by the encryption oracle at block positions P_{256} to P_{511} and form a Λ-set for the right guess of $\mathsf{RK}_{0,1}$, $\mathsf{RK}_{0,6}$, $\mathsf{RK}_{0,11}$, and $\mathsf{RK}_{0,12}$.

The second part of the attack is evaluating the constructed Λ-sets. Since we changed the position of the active tweak byte from T_0 to T_7 compared to the original attack of Section 4.1, we also need to adapt the distinguishing property and evaluate, for instance, $y_0 \oplus y_3$ in state S_5, instead of $y_1 \oplus y_2$. The indices of the guessed round keys and ciphertext bytes need to be adapted accordingly, but otherwise, the attack procedure remains the same. This modification also has no influence on the attack runtime, so the computational complexity is still a total of 2^{82} encryptions to recover 12 bytes of key information.

Accomodating the data complexity limit. Note that with the above strategy, we would need to encrypt $16 \cdot 2^{32}$ messages to obtain 16 Λ-sets per 32-bit guess of RK_0. Thus, we would exceed the maximum number of messages that can be encrypted per key. However, it is possible and necessary to build more than one Λ-set per message following block 511, so that we do not exceed the maximum number of possible messages in our attack. Assume we construct 2^8 Λ-sets per message. This means the first Λ-set covers blocks $256, \ldots, 511$, so $T_6 = 1$ and T_7 is active, the second Λ-set covers blocks $512, \ldots, 767$, so $T_6 = 2$ and T_7 is active, and so on, until we have 2^8 Λ-sets. Thus, every message we query has a length of $2^{16} + 255$ blocks. This means we need 2^{28} chosen messages sent to the encryption oracle, corresponding to $2^{44} + 2^{36}$ chosen plaintext blocks for the attack.

Adaptation for unpredictable nonces. For simplicity, we assumed that the nonce value for each encryption query is predictable, since we needed the value of the nonce byte at tweak position T_1 in order to derive the plaintext values p_1 for each key guess of $\mathsf{RK}_{0,1}$. However, the attack can also be adapted for cases where the nonce is not known as follows. The attacker assumes $T_1 = 0$ and simply queries one message per guess of RK_0. The actual values of T_1 will be random, so for each value of $\mathsf{RK}_{0,6}, \mathsf{RK}_{0,11}, \mathsf{RK}_{0,12}$, the attacker effectively queried sets for 2^8 random values of $\mathsf{RK}_{0,1}$. Due to possible collisions, these queries will, on average, cover a fraction of about $1 - \frac{1}{e} \approx 63.2\,\%$ of all 2^8 possible values of $\mathsf{RK}_{0,1}$. The attack is only successful if the correct value of $\mathsf{RK}_{0,1}$ is among the covered fraction, so the success probability of the overall attack will be about $63.2\,\%$. This can be improved by asking several queries per key guess, e.g., 4 queries for a success probability of about $1 - \frac{1}{e^4} \approx 98.2\,\%$, at the cost of an increase in data complexity by a factor of 2^2 (but no increase in computational complexity).

An alternative, deterministic approach is to query 2^8 Λ-sets per guess of $\mathsf{RK}_{0,1}$, one for each possible value of T_1. All 2^8 Λ-sets need to be queried in one message, to get a constant nonce value and thus definitely cover the correct guess of T_1. Each message now contains $2^{24} + 255$ blocks, and we query a total of $2^{52} + 2^{36}$ blocks. Again, the computational complexity remains at 2^{82} encryptions.

6 Conclusion

In this work, we presented the first third-party analysis of Kiasu-BC. We showed that the additional tweak input can be exploited to create a distinguisher based on the Square property spanning 4 rounds. This is one more round compared to the distinguisher used in Square attacks on AES-128. Hence, we were able to perform key-recovery attacks on 7-round Kiasu-BC with a computational complexity of only about $2^{48.5}$ encryptions, which is faster than the best 7-round attacks for AES-128. However, we cannot attack more rounds compared to AES-128 and hence our analysis does not contradict with the claim of the designers that Kiasu-BC has a sufficient security margin.

Variants of the Square attacks on Kiasu-BC are also applicable if Kiasu-BC is used in one of its recommended modes of operation. We demonstrated this with a nonce-respecting key-recovery attack on Kiasu$\neq$, a ΘCB3-like mode of the CAESAR candidate Kiasu. The computational complexity of this attack is approximately 2^{82} encryptions for 7-round Kiasu-BC, and the attack also respects the low data query limits.

References

1. Daemen, J., Knudsen, L.R., Rijmen, V.: The block cipher Square. In: Biham, E. (ed.) Fast Software Encryption – FSE '97. LNCS, vol. 1267, pp. 149–165. Springer (1997)

2. Daemen, J., Rijmen, V.: AES proposal: Rijndael. National Institute of Standards and Technology (1998)
3. Daemen, J., Rijmen, V.: The Design of Rijndael: AES – The Advanced Encryption Standard. Information Security and Cryptography, Springer (2002)
4. Demirci, H., Taskin, I., Çoban, M., Baysal, A.: Improved meet-in-the-middle attacks on AES. In: Roy, B.K., Sendrier, N. (eds.) Progress in Cryptology – INDOCRYPT 2009. LNCS, vol. 5922, pp. 144–156. Springer (2009)
5. Derbez, P., Fouque, P., Jean, J.: Improved key recovery attacks on reduced-round AES in the single-key setting. In: Johansson, T., Nguyen, P.Q. (eds.) Advances in Cryptology – EUROCRYPT 2013. LNCS, vol. 7881, pp. 371–387. Springer (2013)
6. Ferguson, N., Kelsey, J., Lucks, S., Schneier, B., Stay, M., Wagner, D., Whiting, D.: Improved cryptanalysis of Rijndael. In: Schneier, B. (ed.) Fast Software Encryption – FSE 2000. LNCS, vol. 1978, pp. 213–230. Springer (2000)
7. Gilbert, H., Minier, M.: A collision attack on 7 rounds of Rijndael. In: AES Candidate Conference. pp. 230–241 (2000)
8. Grosso, V., Leurent, G., Standaert, F., Varici, K., Journault, A., Durvaux, F., Gaspar, L., Kerckhof, S.: SCREAM. Submission to the CAESAR competition: `http://competitions.cr.yp.to/round2/screamv3.pdf` (2015)
9. Jean, J., Nikolic, I., Peyrin, T.: KIASU. Submission to the CAESAR competition: `http://competitions.cr.yp.to/round1/kiasuv1.pdf` (2014)
10. Jean, J., Nikolic, I., Peyrin, T.: Tweaks and keys for block ciphers: The TWEAKEY framework. In: Sarkar, P., Iwata, T. (eds.) Advances in Cryptology – ASIACRYPT 2014, Part II. LNCS, vol. 8874, pp. 274–288. Springer (2014)
11. Jean, J., Nikolic, I., Peyrin, T.: Tweaks and keys for block ciphers: the TWEAKEY framework. IACR Cryptology ePrint Archive 2014, 831 (2014), `http://eprint.iacr.org/2014/831`
12. Jean, J., Nikolic, I., Peyrin, T.: Deoxys. Submission to the CAESAR competition: `http://competitions.cr.yp.to/round2/deoxysv13.pdf` (2015)
13. Jean, J., Nikolic, I., Peyrin, T.: Joltik. Submission to the CAESAR competition: `http://competitions.cr.yp.to/round2/joltikv13.pdf` (2015)
14. Krovetz, T., Rogaway, P.: The software performance of authenticated-encryption modes. In: Joux, A. (ed.) Fast Software Encryption – FSE 2011. LNCS, vol. 6733, pp. 306–327. Springer (2011)
15. Liskov, M., Rivest, R.L., Wagner, D.: Tweakable block ciphers. In: Yung, M. (ed.) Advances in Cryptology – CRYPTO 2002. LNCS, vol. 2442, pp. 31–46. Springer (2002)
16. Liskov, M., Rivest, R.L., Wagner, D.: Tweakable block ciphers. J. Cryptology 24(3), 588–613 (2011)
17. Lu, J., Dunkelman, O., Keller, N., Kim, J.: New impossible differential attacks on AES. In: Chowdhury, D.R., Rijmen, V., Das, A. (eds.) Progress in Cryptology – INDOCRYPT 2008. LNCS, vol. 5365, pp. 279–293. Springer (2008)
18. Mala, H., Dakhilalian, M., Rijmen, V., Modarres-Hashemi, M.: Improved impossible differential cryptanalysis of 7-round AES-128. In: Gong, G., Gupta, K.C. (eds.) Progress in Cryptology – INDOCRYPT 2010. LNCS, vol. 6498, pp. 282–291. Springer (2010)
19. The CAESAR committee: CAESAR: Competition for authenticated encryption: Security, applicability, and robustness (2014), `http://competitions.cr.yp.to/caesar.html`