Birthday-Bound Slide Attacks on TinyJAMBU's Keyed-Permutations for All Key Sizes

Ferdinand Sibleyras(✉), Yu Sasaki(✉), Yosuke Todo(✉), Akinori Hosoyamada(✉), and Kan Yasuda(✉)

NTT Social Informatics Laboratories, Tokyo, Japan
{sibleyras.ferdinand.ez,yu.sasaki.sk,yosuke.todo.xt, akinori.hosoyamada.bh,kan.yasuda.hy}@hco.ntt.co.jp

Abstract. We study the security of the underlying keyed-permutations of NIST LWC finalist TinyJAMBU. Our main findings are key-recovery attacks whose data and time complexities are close to the birthday bound 2^{64}. The attack idea works for all versions of TinyJAMBU permutations having different key sizes, irrespective of the number of rounds repeated in the permutations. Most notably, the attack complexity is only marginally increased even when the key size becomes larger. Concretely, for TinyJAMBU permutations of key sizes 128, 192, and 256 bits, the data/time complexities of our key-recovery attacks are about 2^{65}, 2^{66}, and $2^{69.5}$, respectively. Our attacks are on the underlying permutations and not on the TinyJAMBU AEAD scheme; the TinyJAMBU mode of operation limits the applicability of our attacks. However, our results imply that TinyJAMBU's underlying keyed-permutations cannot be expected to provide the same security levels as robust block ciphers of the corresponding block and key sizes. Furthermore, the provable security of TinyJAMBU AEAD scheme should be carefully revisited, where the underlying permutations have been assumed to be almost ideal.

Keywords: TinyJAMBU · NIST LWC · keyed-permutation · slide attack

1 Introduction

The Lightweight Cryptography standardization by NIST (NIST LWC) [9] is one of the most actively discussed topics recently in the symmetric-key cryptography community. In March 2021, out of 56 candidates NIST kept 10 finalists [10] whose evaluations would take approximately 12 months [11].

In this paper we target TinyJAMBU [16], one of the finalists of NIST LWC. TinyJAMBU was designed by Wu and Huang. Roughly speaking, TinyJAMBU (Fig. 2) can be seen as the duplex construction [2] with its public permutation replaced by a 128-bit keyed-permutations; or again as similar to SAEB [8].

C.-M. Cheng and M. Akiyama (Eds.): IWSEC 2022, LNCS 13504, pp. 107–127, 2022.
https://doi.org/10.1007/978-3-031-15255-9_6

TinyJAMBU has one of the smallest hardware footprints of all the finalists. One reason is its small 128 bits of internal state which is near optimal. Moreover, the round function (Fig. 1) consists of a non-linear feedback shift register (NLFSR) with only four XOR operations and a single NAND operation. To optimize throughput, TinyJAMBU varies the number of rounds of the keyed-permutations throughout the mode. $P1$ denotes a permutation with fewer rounds, and $P2$ the one with more rounds.

Table 1. Summary of attacks. KP, CP, and ACP represent known-plaintexts, chosen-plaintexts, and adaptively-chosen plaintexts, respectively. †: This corresponds to the Type-2 difference with a probability of 2^{-47} [15]. In [16], this analysis was deleted by considering the difficulty of exploiting it through the mode. Because our interest is $P2$ as a standalone primitive without the mode, this analysis is of our interest.

Approach	Rounds	Key size	Setting	Data	Time	Memory	Reference
differential	512†	any	CP	2^{48}	-	-	[14,15]
differential	640		CP	2^{84}	-	-	[16]
linear	512		KP	2^{60}	-	-	[14,16]
slide	infinite	128	KP	2^{65}	2^{65}	2^{64}	Section 3.1
			KP	2^{64}	2^{65}	2^{64}	Section 3.2
			ACP	$2^{72.5}$	$2^{72.5}$	negl.	Section 3.2
		192	ACP	2^{65}	2^{66}	2^{65}	Section 4.4
			CP	2^{67}	2^{69}	2^{66}	Section A.1
		256	ACP	$2^{67.5}$	$2^{69.5}$	$2^{67.5}$	Section 5

Because of its minimalist design, the security of TinyJAMBU needs to be carefully assessed. For instance, the security proof of TinyJAMBU assumes that both $P1$ and $P2$ are ideal keyed-permutations, while deliberately making $P1$ weaker. As a matter of fact, the designers have already increased the number of rounds of $P1$ from 384 to 640 following a forgery attack over a 338 rounds $P1$ by Saha et al. [14]. Unlike the old $P1$, $P2$ seems to resist those cryptanalyses due to a larger number of rounds.

In this paper, we focus on slide attacks on TinyJAMBU keyed-permutation which cannot be thwarted by increasing the number of rounds. As a matter of fact, the designers do not make any claim on sliding property in the single-key setting. Moreover, we are interested in sliding property that leads to actual key-recovery attacks and their total complexity.

1.1 Our Contributions

In this paper, we study the security of the underlying keyed-permutation of TinyJAMBU as a standalone primitive. Particularly, we investigate all the details of the sliding property of the keyed-permutation to show that the sliding property

actually leads to efficient key-recovery attacks for all key sizes. Intuitively, by ignoring constant factors, the keyed-permutation can be attacked with about 2^{64} queries and computational cost. Most notably, the attack complexity is only marginally increased even when the key size becomes larger.

We begin with a slide attack on the keyed-permutation of TinyJAMBU-128 because of its simplicity. Slide attacks need to detect a slid pair by using the birthday paradox, which makes it inevitable to make 2^{64} queries. The simplest attack scenario requires almost no extra overhead from this minimum requirement, which results in the data, time, and memory complexities of 2^{65}, 2^{65}, and 2^{64}, respectively. We then discuss a small observation to halve the data complexity and apply the memoryless meet-in-the-middle attack to achieve the data and time complexities of $2^{72.5}$, while the required memory amount is negligible.

However, the simple attack on TinyJAMBU-128 cannot be trivially applied to a larger key size as the keyed-permutation for a k-bit key has a periodical structure in every k rounds weakening the key materials recovered by a slid pair. Nevertheless, the information loss for a large key can be compensated for by generating more slid pairs. Such a challenge has already been discussed in the pioneering work [4], and the technique of making a chain of queries was proposed. The same technique has been exploited by many following works [1,3,6]. In this paper, we present a new technique called "splitting longer chains" that generates more slid pairs than the previous method.

Then, we recover the key from multiple input and output pairs of P_k by applying linear algebra. In particular, we experimentally verified the correctness of our key-recovery algorithm by assuming an access to several slid pairs. As a result, we show that the keyed-permutation of TinyJAMBU-192 and TinyJAMBU-256 can be attacked with a marginally increased complexity than the case with TinyJAMBU-128. The complexities of our attacks are summarized in Table 1.

Lastly, we show several observations on the keyed-permutation: a combination of probability 1 differential characteristics and slide attacks to avoid adaptively-chosen-plaintext queries, a transformation of P_k to the iterative FX-construction [7], extension of our attacks so that the number of rounds that is not a multiple of the key-length can be attacked, and implication of our attacks to the authenticated-encryption with associated data (AEAD) schemes.

Note that results presented in this paper do not violate the security claim of TinyJAMBU, which is only for the entire scheme including the mode. Nevertheless, security of the keyed-permutation is of interest, because it is assumed to be ideal in the security proof. We believe that the security analysis in this paper will be valuable for the NIST to choose the winner(s) of NIST LWC.

2 Specifications

TinyJAMBU is a family of AEAD schemes that supports the key sizes of 128, 192, and 256 bits. Each version is called TinyJAMBU-128, TinyJAMBU-192, and TinyJAMBU-256, respectively. TinyJAMBU uses an n-round keyed-permutation P_n as a building block.

2.1 Keyed-Permutation P_n

The keyed-permutation P_n uses an internal state of 128 bits for all the key sizes, which is represented by $s_0, s_1, \ldots, s_{127}$. Let $k_0, k_1, \ldots, k_{klen-1}$ denote the $klen$-bit key. The internal state is updated by applying the following NLFSR n times by increasing i from 0 to $n-1$.

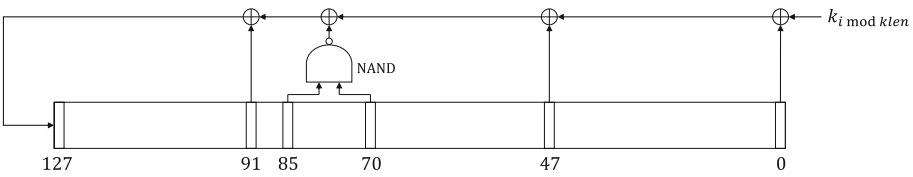

Fig. 1. Step-update function of TinyJAMBU for a $klen$-bit key.

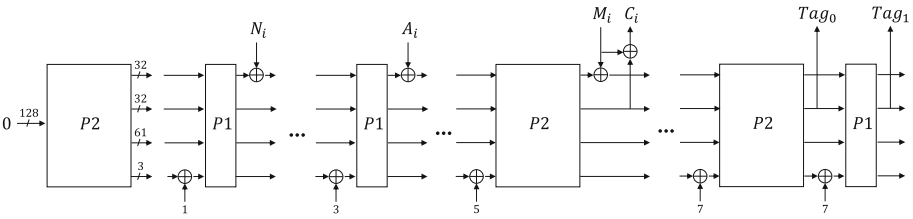

Fig. 2. The mode of TinyJAMBU. $P2$ is P_{1024}, P_{1152}, and P_{1280} for TinyJAMBU-128, TinyJAMBU-192, and TinyJAMBU-256. $P1$ is P_{640}, which was updated from previous P_{384} at the last-round design tweak in NIST LWC.

$$
\begin{aligned}
&\text{feedback} \leftarrow s_0 \oplus s_{47} \oplus (\neg(s_{70} \wedge s_{85})) \oplus s_{91} \oplus k_{i \bmod klen} \\
&\text{for } j \text{ from 0 to 126} : s_j \leftarrow s_{j+1} \\
&s_{127} \leftarrow \text{feedback}
\end{aligned}
$$

where '$\oplus$,' '$\wedge$,' and '$\neg$,' are XOR, AND, and NOT, respectively. The NLFSR is depicted in Fig. 1. Note that the tapping bit-positions were chosen so that 32 rounds of P_n can be computed in parallel on 32-bit CPUs.

2.2 AEAD Mode

The computation structure of TinyJAMBU is described in Fig. 2, which resembles the duplex mode with the keyed-permutation P_n. The details of the mode are omitted in this paper because our target is P_n. To process the nonce, the associated data, and the second half of the tag, the round number n is 640 for all key lengths, which is denoted by $P1$. During the initialization, the encryption, and the first half of the tag, the round number n is 1024, 1152, and 1280 for TinyJAMBU-128, TinyJAMBU-192, and TinyJAMBU-256, respectively, which is denoted by $P2$.

The main reason our attack strategy hardly applies to the AEAD mode is that an attacker can only observe 32 bits out of the 128-bit input and output of any permutation calls to $P1$ and $P2$.

2.3 Security Claim

64-bit security for authentication and 112-, 168-, and 224-bit security for encryption are claimed for TinyJAMBU-128, TinyJAMBU-192, and TinyJAMBU-256 respectively, against nonce-respecting adversaries who make at most 2^{50} bytes of queries.

Security of TinyJAMBU mode was proven assuming that $P1$ and $P2$ are ideal keyed-permutations. Nevertheless, the designers reported the existence of a differential characteristic with a probability of 2^{-47}[1] and a linear characteristic with a bias of 2^{-30} for 512 rounds [15], which is sufficient to conclude that P_{384}, the original round number for $P1$, can be distinguished from an ideal object. Note that no analysis has been known for more than 512 rounds. In particular, it seems that differential and linear cryptanalysis cannot be applied to P_{1024}, P_{1152}, and P_{1280} used in $P2$.

2.4 Self-similarity of P_n

The keyed-permutation P_n does not use any round constant. Moreover, the bits from the key are computed by $k_{i \bmod klen}$. Hence, as mentioned by the designers [16], the state-update function of P_n has some sliding property, which is shown to be exploited with two related keys.[2]

Given that the internal state size is 128 bits, TinyJAMBU-128 shows the best fit because P_n is iterative in every 128 rounds and each state bit is updated exactly once with each key bit. In the following, we first describe the attack for TinyJAMBU-128 and later extend the attack to TinyJAMBU-192 and TinyJAMBU-256.

3 Slide Attacks on TinyJAMBU-128

This section presents a slide attack on TinyJAMBU-128. Because of its simplicity, it bears some similarity with other works, e.g. Bar-on et al. [1, Alg.1]. We first describe a key-recovery attack with 2^{65} known-plaintext queries, 2^{65} offline computations of P_{128}, and a memory to store 2^{64} queries. We then discuss an idea to halve the data complexity and further discuss a memoryless variant of

[1] This corresponds to the Type-2 difference [15]. In [16], the analysis about the Type-2 difference was deleted due to the difficultly of exploiting it through the mode. Our interest is $P2$ as a standalone primitive, so the Type-2 difference is of our interest.

[2] The designers did not give any details of this related-key attack, but when $K' = K \lll 1$, key bits for K' from round 1 to n equal the key bits for K from round 2 to $n+1$. Hence, a plaintext M processed by E_K and a plaintext $P_1^K(M)$ processed by $E_{K'}$ are actually the 1-round slid pair.

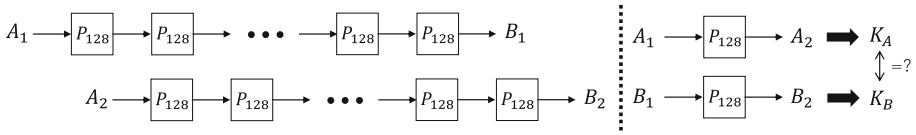

Fig. 3. Overview of slide attacks on the keyed-permutation of TinyJAMBU-128.

the attack. Note that the attack can work for $128t$ rounds for any positive integer $t > 1$ including $P1$ and $P2$ of TinyJAMBU-128, and the attack works in the single-key setting.

3.1 Overview of the Simple Slide Attack

The core of the slide attack (Fig. 3) is to find a slid pair; a pair of plaintext-ciphertext pairs (A_1, B_1) and (A_2, B_2), in which A_2 is the internal state after the first application of P_{128} for A_1, or $A_2 = P_{128}(A_1)$. This simultaneously ensures that $B_2 = P_{128}(B_1)$. A slid pair is generated by using the birthday paradox. The attacker makes 2^{64} queries of A_1 and of A_2 to obtain the corresponding B_1 and B_2. Then, among all the 2^{128} pairs, one pair will be a slid pair with good probability. The slid pair can be detected via a 113-bit filter and a collision-finding algorithm with a computational cost of 2^{64}.

Computing 113-Bit Filter. For a given pair of plaintext-ciphertext pairs (A_1, B_1) and (A_2, B_2), we want to know whether the induced key for $A_2 = P_{128}(A_1)$ and for $B_2 = P_{128}(B_1)$ collides. We do a collision-finding algorithm on values computed separately from (A_1, B_1) and (A_2, B_2), which is denoted by $G_1(A_1, B_1)$ and $G_2(A_2, B_2)$. We denote $G_x(A_x, B_x)$, $x \in \{1, 2\}$ with respect to the i-th bit by $G_x(A_x, B_x)[i]$.

Let $a_0, a_1, \ldots, a_{127}$ and $a_{128}, a_{129}, \ldots, a_{255}$ denote A_1 and A_2, respectively and $b_0, b_1, \ldots, b_{127}$ and $b_{128}, b_{129}, \ldots, b_{255}$ denote B_1 and B_2, respectively. If (A_1, A_2) is a slid pair then so is (B_1, B_2) and k_i is computed as follows:

$$\begin{aligned} k_i &= a_{i+128} \oplus a_i \oplus a_{i+47} \oplus (\neg(a_{i+70} \wedge a_{i+85})) \oplus a_{i+91} \\ &= b_{i+128} \oplus b_i \oplus b_{i+47} \oplus (\neg(b_{i+70} \wedge b_{i+85})) \oplus b_{i+91}. \end{aligned}$$

Bit Positions 0 *to* 36. For $i = 0, 1, \ldots, 36$, we let $G_1(A_1, B_1)[i]$ and $G_2(A_2, B_2)[i]$ be the XOR sum of the terms belonging to (A_1, B_1) and (A_2, B_2) with respect to the i-th bit, respectively, i.e.,

$$\begin{aligned} G_1(A_1, B_1)[i] :=\ & a_i \oplus a_{i+47} \oplus (\neg(a_{i+70} \wedge a_{i+85})) \oplus a_{i+91} \oplus \\ & b_i \oplus b_{i+47} \oplus (\neg(b_{i+70} \wedge b_{i+85})) \oplus b_{i+91}, \\ G_2(A_2, B_2)[i] :=\ & a_{i+128} \oplus b_{i+128}. \end{aligned}$$

$G_1(A_1, B_1)[i]$ and $G_2(A_2, B_2)[i]$ can be computed independently from the other pair, hence a collision on 37 bits of $k_0, k_1, \ldots, k_{36}$ can be observed.

Bit Positions 37 *to* 42. For $i = 37, 38, \ldots, 42$, notice that the term a_{i+91} belongs to A_2. The same applies to B_2. Hence, we have

$$G_1(A_1, B_1)[i] := a_i \oplus a_{i+47} \oplus (\neg(a_{i+70} \wedge a_{i+85})) \oplus b_i \oplus b_{i+47} \oplus (\neg(b_{i+70} \wedge b_{i+85})),$$
$$G_2(A_2, B_2)[i] := a_{i+91} \oplus a_{i+128} \oplus b_{i+91} \oplus b_{i+128}.$$

No Filter for Bit Positions 43 *to* 57. For $i = 43, 44, \ldots, 57$, one of the inputs to the AND operation, a_{i+85} (resp. b_{i+85}), belongs to A_2 (resp. B_2), while the other input bit, a_{i+70} (resp. b_{i+70}), belongs to A_1 (resp. B_1). Hence, the output of the AND operation cannot be computed independently.

Bit Positions 58 *to* 80 *and* 81 *to* 127. Following the same strategy, equations for $i = 58, 59, \ldots, 80$ are defined as

$$G_1(A_1, B_1)[i] := a_i \oplus a_{i+47} \oplus b_i \oplus b_{i+47},$$
$$G_2(A_2, B_2)[i] := (\neg(a_{i+70} \wedge a_{i+85})) \oplus a_{i+91} \oplus a_{i+128} \oplus (\neg(b_{i+70} \wedge b_{i+85})) \oplus b_{i+91} \oplus b_{i+128},$$

and equations for $i = 80, 81, \ldots, 127$ are defined as

$$\begin{aligned} G_1(A_1, B_1)[i] &:= a_i \oplus b_i, \\ G_2(A_2, B_2)[i] &:= a_{i+47} \oplus (\neg(a_{i+70} \wedge a_{i+85})) \oplus a_{i+91} \oplus a_{i+128} \oplus \\ &\quad b_{i+47} \oplus (\neg(b_{i+70} \wedge b_{i+85})) \oplus b_{i+91} \oplus b_{i+128}. \end{aligned}$$

Summary. For each A_1 and its query-output B_1, the attacker can compute a 113-bit value to match with $G_1(A_1, B_1)[i]$ for $i = \in \{0, 1, \ldots, 127\} \backslash \{43, 44, \ldots, 57\}$. Similarly, for (A_2, B_2), the 113-bit value to match can be computed with $G_2(A_2, B_2)$.

Attack Procedure. The pseudo-algorithm to recover the key of TinyJAMBU-128 is described in Algorithm 1. For simplicity, here we assume that a table T of size 2^{64} is available.

Analysis. In the above attack procedure, the attacker makes 2^{64} queries of A_1 and A_2, thus the data complexity is 2^{65} known-plaintexts. The bottleneck of the time complexity is to compute $G_1(A_1, B_1)$ and $G_2(A_2, B_2)$, which is 2^{65} computations of P_{128}. The attack requires a memory of size 2^{64} for the table. (The table for Step 2 can be omitted by checking the collision in an online manner when a value of $G(A_2, B_2)$ is obtained.) In Step 3, 2^{128} pairs are examined and $2^{128-113} = 2^{15}$ pairs will pass this filter, and a valid pair will be detected by matching the remaining 15 bits.

Algorithm 1. A simple slide attack on TinyJAMBU-128 with 2^{64} memory.

1: Generate 2^{64} distinct values for A_1, obtain all the respective B_1 with 2^{64} queries, compute $G_1(A_1, B_1)$ for the 113 bits, and store $(A_1, B_1, G_1(A_1, B_1))$ in the table.
2: Generate 2^{64} distinct values for A_2, obtain all the respective B_2 with 2^{64} queries, compute $G_2(A_2, B_2)$ for the 113 bits, and store $(A_2, B_2, G_2(A_2, B_2))$ in the table.
3: Find collisions of $G_1(A_1, B_1)$ and $G_2(A_2, B_2)$ for all $2^{64} \times 2^{64} = 2^{128}$ pairs.
4: **for** all pairs with $G_1(A_1, B_1) = G_2(A_2, B_2)$ **do**
5: Derive $k_{43}, \ldots, k_{57}$, with $A_2 = P_{128}(A_1)$ and also with $B_2 = P_{128}(B_1)$.
6: **if** $k_{43}, \ldots, k_{57}$ from $A_2 = P_{128}(A_1)$ and from $B_2 = P_{128}(B_1)$ collide **then**
7: **return** K.
8: **end if**
9: **end for**

3.2 Reducing Data or Memory Complexity

Halving Data Complexity. Algorithm 1 assumes that queries in Step 1 correspond to the input to P_{128} and queries in Step 2 correspond to the output from P_{128}. However, we can reuse the data of Step 1 in Step 2 and look for a collision the same way. This would halve the data complexity from 2^{65} to 2^{64}.

A Memoryless Variant. As in [1], the 2^{64} memory requirement of Algorithm 1 can be removed with the standard memoryless collision-finding algorithm [13], which exploits a cycle-detection algorithm for the query chain. To do so, we start with a 113-bit value v_0, pads it to 128 bits to get A_0, and query to obtain B_0. Then, we compute either $G_1(A_0, B_0)$ or $G_2(A_0, B_0)$ depending on a bit of v_0 (LSB for instance). Set the result as v_1 and iterate this procedure to generate the chain of $v_0, v_1, v_2, \ldots$.

On the memory side, we only store some particular values for instance store the 100 values starting with the most 0 bits. When the chain length reaches about $2^{113/2}$, a newly computed v_i will eventually collide with one of the stored values. The exact colliding point can be found by starting from the stored points before the observed collision. If a collision is between $G_1(A_i, B_i)$ and $G_2(A_j, B_j)$, A_i and A_j is a slid pair candidate. If a collision is between $G_b(A_i, B_i)$ and $G_b(A_j, B_j)$ for the same $b \in \{1, 2\}$, the algorithm is repeated from scratch by changing v_0.

The procedure will be repeated twice on average to find a slid pair candidate which makes $2 \times 2^{113/2} = 2^{57.5}$ queries. And the candidate is a slid pair with probability 2^{-15}, thus we need 2^{15} candidates, which makes the total data complexity of $2^{15} \times 2^{57.5} = 2^{72.5}$ adaptively-chosen-plaintext queries. For each query, the attacker computes G_1 or G_2, thus the time complexity is $2^{72.5}$ computations of P_{128}. The memory amount is negligible when there are sufficiently few stored 113-bit values.

The memoryless meet-in-the-middle attack is an extreme case to optimize the memory complexity. A more general tradeoff for data, time, and memory complexities can be achieved by the parallel collision search [12].

4 Attacks Against a Larger Key

The same filter will not work for longer key versions TinyJAMBU-192 and TinyJAMBU-256 as the permutation repeats only after a number of rounds equal to the key length. However, we show that we can build a $113 - \kappa$-bit filter for $128 + \kappa$-bit key permutation and still do a key recovery from a slid pair.

4.1 Building a Filter

Concretely, for a given pair of plaintext-ciphertext pairs (A_1, B_1) and (A_2, B_2), we want to know whether the key for $A_2 = P_{128+\kappa}(A_1)$ and for $B_2 = P_{128+\kappa}(B_1)$ will collide. Hence, just like in Sect. 3.1, we want to compute colliding values separately from (A_1, B_1) and (A_2, B_2) to efficiently look for a collision.

Similarly, let us denote the bit states s_i for $i \in [0, 255 + \kappa]$ such that s_{127} to s_0 is the input, $s_{255+\kappa}$ to $s_{128+\kappa}$ is the output and $s_{127+\kappa}$ to s_{128} are κ bits of internal computations. By definition of the permutation we have:

$$k_i = s_{i+128} \oplus s_i \oplus s_{i+47} \oplus (\neg(s_{i+70} \wedge s_{i+85})) \oplus s_{i+91}.$$

We look for relations of key bits that only depend on input and output bits, that is on s_i for $i \in [0, 127] \cup [128 + \kappa, 255 + \kappa]$.

First, we ignore all key bits whose AND term $(\neg(s_{i+70} \wedge s_{i+85}))$ is not computable given either the input or output bits. There are 113 remaining key bits that are k_i for $i \in [0, 42] \cup [58 + \kappa, 127 + \kappa]$. Indeed, every AND term is unique, so there is no linear combination that can hope to cancel it.

	s_0	s_1	s_2	...	s_{127}	s_{128}	...	$s_{126+\kappa}$	$s_{127+\kappa}$	$s_{128+\kappa}$	...	$s_{255+\kappa}$
k_0	1	0	0	...	0	1	...	0	0	0	...	0
k_1	0	1	0	...	0	0	...	0	0	0	...	0
k_2	0	0	1	...	0	0	...	0	0	0	...	0
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
$k_{126+\kappa}$	0	0	0	...	0	0	...	1	0	0	...	0
$k_{127+\kappa}$	0	0	0	...	0	0	...	0	1	0	...	1

Fig. 4. Construction of the $113 \times (256 + \kappa)$ binary matrix M.

Then, we build a binary matrix M with 113 rows, one row for each considered k_i, and $256 + \kappa$ columns, one column for each state bit s_i. Let $M(i, j) = 1$ if s_j linearly appears in the formula for the ith retained bit key and $M(i, j) = 0$ otherwise, as illustrated in Fig. 4. For instance, $M(0, j) = 1$ for $j \in \{128, 91, 47, 0\}$ and $M(0, j) = 0$ otherwise.

Then, we use row-wise Gaussian elimination on M to put zeroes on the columns 128 to $127 + \kappa$ that correspond to internal computation state bits.

s_0 s_1 s_2 ... s_{127}	s_{128} ... $s_{126+\kappa}$ $s_{127+\kappa}$	$s_{128+\kappa}$ ... $s_{255+\kappa}$
$E^1_{\kappa\times 128}$	$I_{\kappa\times\kappa}$	$S^1_{\kappa\times 128}$
$E^2_{(113-\kappa)\times 128}$	$0_{(113-\kappa)\times(113-\kappa)}$	$S^2_{(113-\kappa)\times 128}$

Fig. 5. The $113\times(256+\kappa)$ binary matrix M after Gaussian elimination. I is the identity matrix, 0 is the zero matrix and E^1, E^2, S^1, and S^2 are binary matrices resulting from the Gaussian elimination.

Assuming the 128 to $127+\kappa$-column submatrix is a full rank $113\times\kappa$ matrix, we will at least recover $113-\kappa$ rows with only zeroes on those columns that naturally correspond to $113-\kappa$ relevant relationships as illustrated in Fig. 5.

The linear part of each relationship is recovered by looking at the other columns of M and the non-linear part must be also added by looking at the corresponding key bits involved. By construction, those $113-\kappa$ relationships are linearly independent and will involve both input and output state bits and only those state bits. Each such row thus implies a relation between key bits, input bits and output bits that can be summarized as $\mathcal{R}(k) = \mathcal{R}_i(A_1) \oplus \mathcal{R}_o(A_2) = \mathcal{R}_i(B_1) \oplus \mathcal{R}_o(B_2)$; implying $\mathcal{R}_i(A_1) \oplus \mathcal{R}_i(B_1) = \mathcal{R}_o(A_2) \oplus \mathcal{R}_o(B_2)$ that can be used to efficiently filter a slid pair among many plaintext-ciphertext.

4.2 Enhancing a Filter with Chains of Queries

With the previous method, to attack P_{240} ($\kappa = 112$), we only have a 1-bit filter which is insufficient. Hence we need a way to leverage on the filter.

Basic Method. We use a technique by Biryukov and Wagner [5] to increase the number of filtering bits by generating more slid pairs. To multiply the number of filtering bits, the attacker can generate a chain of queries. That is, after querying A_1 and receiving B_1, the attacker queries B_1 to obtain C_1, then queries C_1 to obtain D_1, and so on. A similar chain is generated from each A_2. If (A_1, A_2) is a slid pair, then so are (B_1, B_2), (C_1, C_2) and (D_1, D_2). Thus, we have the relationship $\mathcal{R}(k) = \mathcal{R}_i(A_1)\oplus\mathcal{R}_o(A_2) = \mathcal{R}_i(B_1)\oplus\mathcal{R}_o(B_2) = \mathcal{R}_i(C_1)\oplus\mathcal{R}_o(C_2) = \mathcal{R}_i(D_1)\oplus\mathcal{R}_o(D_2) = \cdots$. When the length of the chains is ℓ, the $113-\kappa$-bit filter is applied to ℓ pairs, which achieves a $\ell\cdot(113-\kappa)$-bit filter. For P_{240} ($\kappa = 112$), we set $\ell = 128$ and gets a $128\cdot(113-112) = 128$-bit filter to identify the right slid pair.

Advanced Method: Splitting Longer Chains. We can further chain the queries to efficiently create multiple chains of the required length. Concretely, chains of length $\ell+\beta$ values can be cut into $\beta+1$ chains of length ℓ (Fig. 6).

However, comparing those $\beta + 1$ chains for any reasonable β won't yield any slid pairs since an n-bit permutation won't loop until about $\mathcal{O}(2^n)$ iterations. Nevertheless, comparing two independent chains of length $\ell + \beta$, we can expect to find a solution among the implied $2\beta + 2$ chains with probability $2(2\beta + 1)/2^{128}$ (fixing the first set of chains, there are $2\beta + 1$ starting points for the next set of chains that will provide a slid output and the same amount for a slid input solution). Hence a solution is expected to be found after collecting about $2^{64}/\sqrt{4\beta + 2}$ sets of $\beta + 1$ chains, which makes for a $2^{64}(\ell + \beta)/\sqrt{4\beta + 2}$ data complexity optimized for $\beta = \ell - 1$. For $\ell = 128$, the data complexity becomes $\sqrt{255/2} \cdot 2^{64} \simeq 2^{67.5}$.

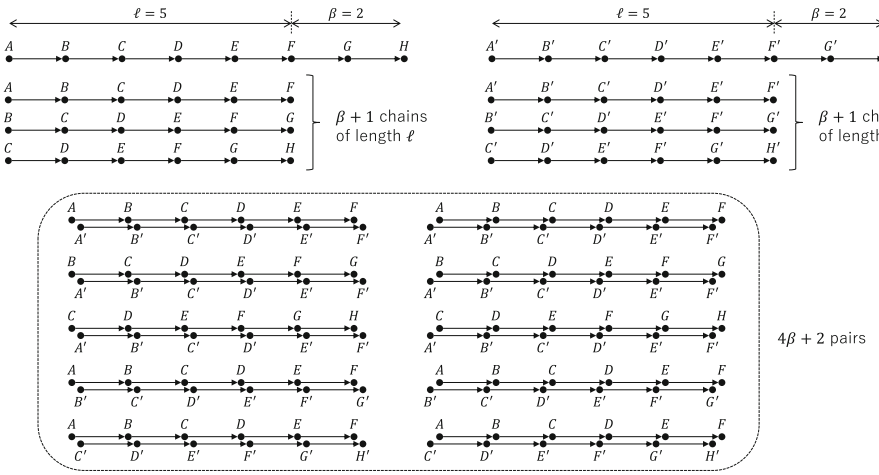

Fig. 6. Schematic representation of splitting longer chains for $\ell = 5$ and $\beta = 2$.

4.3 Key-Recovery from Input/Output Pairs

In this section, we explain how to efficiently extract the $128 + \kappa$-bit key from multiple input/output pairs of $P_{128+\kappa}$ in only about $\kappa \log(\kappa)$ operations where $0 \leq \kappa \leq 113$.

The key-recovery is described in Algorithm 2 which basically guesses the κ unseen bit states one by one. Let us explain the first iteration of the algorithm. We start by taking the matrix M after Gaussian elimination (Fig. 5) that was allegedly used to filter the pairs (A_1, A_2) and (B_1, B_2) both belonging to the set $\mathcal{P}$. We first guess k_0, the first key bit, which corresponds to the first row of our matrix M as it linearly depends on a_{128} and not on the rest of the unseen part. Hence, from k_0 we can deduce a_{128}, b_{128}, etc. for all known slid pairs. With the knowledge of the 128th state bit, a new AND term can be computed that is $a_{113} \wedge a_{128}$ corresponding to k_{43}. Thus, we add the linear term for k_{43} to the matrix M, which now contains 114 rows. Row-wise Gaussian elimination will restore the form of Fig. 5 but with 114 rows and, hence, a $(114 - \kappa)$-bit filter.

This additional bit of filter enables us to check whether the key guess was wrong. If the additional filter pass for all pairs, we proceed. Otherwise, we change our guess. Note that in the last 15 iterations, we can further deduce an additional AND term using a known output bit.

Algorithm 2. Efficient key-recovery after filtering on TinyJAMBU-$(128 + \kappa)$.

Let $\mathcal{P}$ be a list of multiple input/output pairs (S_1, S_2) whose internal (visible and invisible) bit states are denoted as s_i for i from 0 to $255 + \kappa$.
Let M be the filter producing matrix as in Fig. 5.
for i from 0 to $\kappa - 1$ **do**
    $g \leftarrow 0$
    Guess that the relation induced by the $(i + 1)$th row of M sums to g.
    $\forall (S_1, S_2) \in \mathcal{P}$: Deduce s_{128+i} from the guess.
    Add the relation of k_{43+i} in the matrix M.
    if $i \geq \kappa - 15$ **then**
        Add the relation of k_{58+i} in the matrix M.
    end if
    Perform row-wise Gaussian elimination with respect to column $128+i$ to $127+\kappa$.
    Consider the new computable relation (two relations if $i \geq \kappa - 15$).
    if $\forall (S_1, S_2) \in \mathcal{P}$: the relations are not equal **then**
        if $g = 0$ **then**
            $g \leftarrow 1$
            Go back to Step 5
        else
            No consistent key can be fond. **return** $\emptyset$.
        end if
    end if
end for
For some $(S_1, S_2) \in \mathcal{P}$ compute k such that :
$k_i = s_{i+128} \oplus s_i \oplus s_{i+47} \oplus (\neg(s_{i+70} \wedge s_{i+85})) \oplus s_{i+91}$
return k

The probability of success of this algorithm mainly depends on the probability of a wrong guess passing through the additional filter created which depends on the number of input/output pairs we have at hand. Notice that we can further compute additional input/output pairs by chaining the queries as in Sect. 4.2. Gathering around $\log(\kappa)$ input/output pairs will detect a wrong guess with about $1-1/\kappa$ probability. Hence, it will fully recover κ bits of key with good probability ($(1 - 1/\kappa)^\kappa$ tends to $e^{-1} \simeq 36.8\%$ as κ grows) and deduce the full $128 + \kappa$ bits of key. Allowing back-tracking is probably efficient but hard to analyze. Notice that it is possible to know whether the additional filter passed because the guess is true or because it is independent of the guess. Indeed, the AND term we add depend both on a newly guessed key (computed state bit) and on a known state.

Table 2. Experimental reports about Algorithm 2. Success probability with different size of $\mathcal{P}$ on 1000 trials against a theoretical estimate.

$\lvert\mathcal{P}\rvert$	5	6	7	8	9	10	11	12	13	14
success prob. ($\kappa = 64$)	3.6	19.3	46.1	69.7	82.2	90.5	95.0	97.6	98.5	99.5
theoretical prob. ($\kappa = 64$)	4.0	20.8	46.1	68.0	82.5	90.9	95.3	97.6	98.8	99.4
success prob. ($\kappa = 112$)	0.4	4.1	19.9	44.7	69.6	83.8	90.5	95.1	97.7	99.1
theoretical prob. ($\kappa = 112$)	0.2	4.5	21.6	46.7	68.4	82.7	91.0	95.4	97.7	98.8

If the known state was 0, then the AND term does not depend on the guess but if the known state is 1 then the AND term depends linearly on the guess. Comparing both cases together will give us the correct guess; otherwise the filter always verifies independently of the guess.

Experimental Reports. We implemented Algorithm 2 and verified the required number of input/output pairs. We say Algorithm 2 succeeded when it returned the unique secret key. Table 2 summarizes the attack success probability with different sizes of $\mathcal{P}$ and $\kappa \in \{64, 112\}$. The theoretical estimation of the success probability is computed by $(1-2^{-(|\mathcal{P}|-1)})^{(\kappa-15)} \times (1-2^{-2\times(|\mathcal{P}|-1)})^{15}$. It assumes there is a $1/2$ chance to detect a bad guess per filter per additional input/output pairs; we have one filter per step up to $\kappa - 15$ key bits, and two filters for the last 15 key bits. The theoretical estimation of $\log(\kappa)$ pairs required amounts to 6 and 7 for $\kappa = 64$ and $\kappa = 112$, respectively, and has indeed a good probability of success. The theoretical estimations well fit the success probability of our experiments.

4.4 Application on TinyJAMBU-192

The internal permutation of TinyJAMBU-192 is the case with $\kappa = 64$. With the technique in Sect. 4.1, we build a $113 - 64 = 49$-bit basic filter further enhanced by the technique of Sect. 4.2 with chain length $\ell = 2$. This builds a $2 \times 49 = 98$-bit filter and reduces 2^{128} candidate pairs to a sufficiently small size.

The pseudo-algorithm to recover the key of TinyJAMBU-192 is described in Algorithm 3. For simplicity, here we assume that a table T of size 2^{65} is available. In Step 1, we make 2^{65} queries, in which the first 2^{64} queries can be known-plaintexts queries, while the last 2^{64} queries must be adaptively-chosen-plaintext queries. In Step 2, we compute $\mathcal{R}_i$ for two pairs and $\mathcal{R}_o$ for two pairs, which is faster than $4 \times 2^{64} = 2^{66}$ computations of P_{192}. In Step 3, the match of 98 bits will be examined for 2^{128} pairs, hence 2^{30} pairs will remain after the filter. In Step 5, we further make $2 \times 2^{30} = 2^{31}$ adaptively-chosen-plaintext queries. Thanks to the additional 49-bit filter, only the right slid pair will remain after this step. In Step 7, we make additional queries to collect $\log \kappa = 6$ slid pairs, which is required by the key-recovery algorithm. The complexity of the key-recovery algorithm is $64 \times \log 64$, which is negligible. In summary, the bottleneck

Algorithm 3. An adaptively chosen-plaintext slide attack on TinyJAMBU-192.

1: Generate 2^{64} distinct values for A. Make 2^{64} queries of A to obtain B, and make 2^{64} queries of B to obtain C.
2: Compute $\mathcal{R}_i(A,B)$ and $\mathcal{R}_i(B,C)$ for the 98 bits, and compute $\mathcal{R}_o(A,B)$ and $\mathcal{R}_o(B,C)$ for the 98 bits. Store $(A,B,C,\mathcal{R}_i(A,B)\|\mathcal{R}_i(B,C),\mathcal{R}_o(A,B)\|\mathcal{R}_o(B,C))$ in the table.
3: Find collisions of $\mathcal{R}_i(A,B)\|R_i(B,C)$ and $\mathcal{R}_o(A',B')\|R_o(B',C')$ for all 2^{128} pairs.

4: **for** all pairs with $\mathcal{R}_i(A,B)\|\mathcal{R}_i(B,C) = \mathcal{R}_o(A',B')\|\mathcal{R}_o(B',C')$ **do**
5: Make 2 queries of C and C' to obtain D and D'.
6: **if** $\mathcal{R}_i(C,D) = \mathcal{R}_o(C',D')$ **then**
7: Make additional queries to extend the chain length to be $\log \kappa = 6$.
8: Run the key-recovery procedure in Sect 4.3.
9: Return K.
10: **end if**
11: **end for**

of the attack is Steps 1 and 2, which requires 2^{65} adaptively-chosen-plaintext queries, about 2^{66} computational cost, and a memory to store 2^{65} values.

5 Optimization for Attack on TinyJAMBU-256

When $\kappa = 128$, which is the parameter for TinyJAMBU-256, the technique of Sect. 4.1 can no longer construct a filter. Thus, we need additional tricks to attack TinyJAMBU-256. In this section, we optimize the attack on TinyJAMBU-256 by exploiting the structure of TinyJAMBU. First, we show a method to construct a 1-bit filter with only a 2-bit guess. In other words, the complexity is only increased by a factor 2^2. Next, we show an efficient method to recover the secret key given several plaintext-ciphertext pairs on P_{256}. The 15-bit key, i.e., $k_0, \ldots, k_{14}$, is recovered by exploiting the algebraic structure, and then, the other key bits are recovered by using Algorithm 2.

5.1 1-Bit Filter with a 2-Bit Guess

The trivial extension requires an additional 16-bit guess. However, we do not need to guess the whole 16-bit key, and only an additional 2-bit guess is enough to obtain a 1-bit filter. Concretely, guessing the 2 bits of key k_0 and k_{15} is enough. We derive the following equations from the step-update function.

$$s_{128} = s_0 \oplus s_{47} \oplus (\neg(s_{70} \wedge s_{85})) \oplus s_{91} \oplus k_0$$
$$s_{143} = s_{15} \oplus s_{62} \oplus (\neg(s_{85} \wedge s_{100})) \oplus s_{106} \oplus k_{15}$$

By guessing k_0 and k_{15}, we can compute s_{128} and s_{143}. Then, we obtain $k_{21} \oplus k_{58} \oplus k_{186} \oplus k_{233}$ from only known bits. These four key bits are computed as

$$
\begin{aligned}
k_{21} &= s_{21} \oplus s_{68} \oplus (\neg(s_{91} \wedge s_{106})) \oplus s_{112} \oplus s_{149},\\
k_{58} &= s_{58} \oplus s_{105} \oplus (\neg(s_{128} \wedge s_{143})) \oplus s_{149} \oplus s_{186},\\
k_{186} &= s_{186} \oplus s_{233} \oplus (\neg(s_{256} \wedge s_{271})) \oplus s_{277} \oplus s_{314},\\
k_{233} &= s_{233} \oplus s_{280} \oplus (\neg(s_{303} \wedge s_{318})) \oplus s_{324} \oplus s_{361},
\end{aligned}
$$

and the sum is

$$
\begin{aligned}
k_{21} \oplus k_{58} \oplus k_{186} \oplus k_{233} = {} & s_{21} \oplus s_{68} \oplus (\neg(s_{91} \wedge s_{106})) \oplus s_{112} \oplus s_{58} \oplus s_{105} \oplus (\neg(s_{128} \wedge s_{143})) \oplus\\
& s_{314} \oplus (\neg(s_{256} \wedge s_{271})) \oplus s_{277} \oplus s_{361} \oplus s_{280} \oplus (\neg(s_{303} \wedge s_{318})) \oplus s_{324}.
\end{aligned}
$$

Since s_{128} and s_{143} are known by guessing k_0 and k_{15}, we can get this 1-bit filter.

We want to use this 1-bit filter to detect slid pairs. Given a pair of plaintext-ciphertext pairs (A_1, B_1) and (A_2, B_2), we need to define the corresponding functions $G_1(A_1, B_1)$ and $G_2(A_2, B_2)$. Let $(a_0, a_1, \ldots, a_{127})$ and $(a_{256}, a_{257}, \ldots, a_{383})$ denote A_1 and A_2, respectively. Moreover, $(b_0, b_1, \ldots, b_{127})$ and $(b_{256}, b_{257}, \ldots, b_{383})$ denote B_1 and B_2, respectively. Then, two functions are defined as

$$
\begin{aligned}
G_1(A_1, B_1) := {} & a_{21} \oplus a_{68} \oplus (\neg(a_{91} \wedge a_{106})) \oplus a_{112} \oplus a_{58} \oplus a_{105} \oplus (\neg(a_{128} \wedge a_{143})) \oplus\\
& b_{21} \oplus b_{68} \oplus (\neg(b_{91} \wedge b_{106})) \oplus b_{112} \oplus b_{58} \oplus b_{105} \oplus (\neg(b_{128} \wedge b_{143}))\\
G_2(A_2, B_2) := {} & a_{314} \oplus (\neg(a_{256} \wedge a_{271})) \oplus a_{277} \oplus a_{361} \oplus a_{280} \oplus (\neg(a_{303} \wedge a_{318})) \oplus a_{324} \oplus\\
& b_{314} \oplus (\neg(b_{256} \wedge b_{271})) \oplus b_{277} \oplus b_{361} \oplus b_{280} \oplus (\neg(b_{303} \wedge b_{318})) \oplus b_{324}.
\end{aligned}
$$

Note that $G_1(A_1, B_1)$ depends on the guess of k_0 and k_{15}, but $G_2(A_2, B_2)$ is independent of them.

5.2 Key-Recovery from Input/Output Pairs for P_{256}

Algorithm 2 accepts κ until 113. Therefore, Algorithm 2 cannot be applied to P_{256} directly. On the other hand, trivial extension is possible by guessing $15(= 128 - 113)$-bit key. Recall that Algorithm 2 is very efficient and the time complexity is $O(\kappa)$. Even if we additionally guess the 15-bit key, the impact on the time complexity is negligible compared with previous steps. Although the trivial extension is already efficient, we present a more efficient algorithm whose time complexity is still $O(\kappa)$.

In Algorithm 2, the corresponding row vector is not involved in the matrix if either of the NAND inputs is unknown. However, in practice, only one side of NAND inputs is known, so we can obtain an additional relationship. Considering the following NAND $\neg(s_t \wedge s_{t+15})$, the output of the NAND is always 1 independently of s_{t+15} when $s_t = 0$. On the other hand, when $s_t = 1$, the output of the NAND is $s_{t+15} \oplus 1$, i.e., the nonlinear output is linearized. By exploiting this property, we can recover the first 15-bit key efficiently.

The following is a concrete case to recover k_0. When $(s_{113}, s_{256}) = (0, 0)$, we can compute $k_6 \oplus k_{43} \oplus k_{171} \oplus k_{218}$ as

$$
\begin{aligned}
k_6 \oplus k_{43} \oplus k_{171} \oplus k_{218} = {} & s_6 \oplus s_{53} \oplus (\neg(s_{76} \wedge s_{91})) \oplus s_{97} \oplus s_{43} \oplus s_{90} \oplus s_{262}\\
& \oplus s_{299} \oplus s_{265} \oplus (\neg(s_{288} \wedge s_{303})) \oplus s_{309} \oplus s_{346}.
\end{aligned}
$$

As the sum removes 3 uncomputable bits, i.e., s_{134}, s_{171}, and s_{218}. Moreover, when $(s_{113}, s_{256}) = (1, 0)$, we can compute $k_0 \oplus k_6 \oplus k_{43} \oplus k_{171} \oplus k_{218}$ as

$$\begin{aligned} k_0 \oplus k_6 \oplus k_{43} \oplus k_{171} \oplus k_{218} = {} & s_0 \oplus s_{47} \oplus (\neg(s_{70} \wedge s_{85})) \oplus s_{91} \oplus s_6 \oplus s_{53} \\ & \oplus (\neg(s_{76} \wedge s_{91})) \oplus s_{97} \oplus s_{43} \oplus s_{90} \oplus 1 \oplus (\neg(s_{241} \wedge s_{256})) \\ & \oplus s_{262} \oplus s_{299} \oplus s_{265} \oplus (\neg(s_{288} \wedge s_{303})) \oplus s_{309} \oplus s_{346}. \end{aligned}$$

As the sum removes 4 uncomputable bits, i.e., s_{128}, s_{134}, s_{171}, and s_{218}. Finally, the key bit k_0 is derived by summing these two equations. This procedure requires one input-output pairs satisfying each conditions, but the number of restricted bits is only 2. Therefore, we can recover k_0 by observing about 4 input-output pairs. This procedure can be used to recover k_x for $0 \leq x \leq 14$. Then, the restricted bits move to (s_{113+x}, s_{256+x}).

5.3 Complexity of TinyJAMBU-256

The attacker guesses 2-bit key k_0 and k_{15} and generates a 1-bit filter. Since a 1-bit filter is insufficient to detect a unique slid pair, the filter is enhanced with chains of queries. Thus, the attacker enhances the 1-bit filter to a 128-bit filter and detects only a right slid pair for each 2-bit guess. Deriving the key from a slid pair is very efficient by using Algorithm 2 with the technique shown in Sect. 5.2. Thus, the data complexity is $2^{67.5}$. The time complexity is $2^{69.5}$.

6 Conclusions

We have thoroughly analyzed the slide property of the keyed-permutation used as TinyJAMBU's underlying primitive. Our analysis shows that the slide property can be exploited to mount actual slide attacks with near-birthday-bound complexities for all proposed key sizes (128, 192, and 256 bits). The attacks exploit multiple (undesirable) properties of the primitive and work independently from the number of rounds repeated in the permutation.

The attacks do not directly contradict with the security goals to be achieved by TinyJAMBU [16] but invalidate the rationale that the underlying primitive is close to ideal. In particular, the attacks bring into question the (relatively high) 112/168/224-bit encryption/secret-key security goal for TinyJAMBU.

We emphasize that one should not treat TinyJAMBU's primitive as a standard block cipher like Advanced Encryption Standard (AES), as TinyJAMBU's keyed-permutation fails to provide the expected security level (the functionality of a keyed-permutation is the same as that of a block cipher.) The keyed-permutation is a dedicated primitive that should be used exclusively in TinyJAMBU's AEAD mode of operation.

A Discussions and More Observations

A.1 Slide Attack with Deterministic Differential Characteristics

Overall Idea. The chain of queries in Sect. 4.2 efficiently increases the number of filtering bits, but requires adaptively chosen-plaintext. Here, we discuss another approach that was also discussed in [5] which avoids adaptively chosen-plaintext queries and show that it can be applied to recover a 192-bit key. The idea here is to combine differential characteristics with probability 1 with the slide attack. Suppose that there is an input and output difference of P_{192} denoted by α and β, which is satisfied with probability 1. For a slid pair (A_0, B_0) and (A'_0, B'_0) such that $A'_0 = P_{192}(A_0)$ and $B'_0 = P_{192}(B_0)$, we define that $A_1 = A_0 \oplus \alpha$ and $A'_1 = A'_0 \oplus \beta$. Then the pair (A_1, B_1) and (A'_1, B'_1) also satisfies $A'_1 = P_{192}(A_1)$ and $B'_1 = P_{192}(B_1)$ thanks to the probability 1 differential characteristic. Specifically, we obtain 2 slid pairs without using adaptively-chosen-plaintext queries. Moreover, the number of slid pairs can further increase to 2^n if n-many probability 1 differential characteristics are available, by assuming that it is possible to satisfy such n-many probability 1 characteristics simultaneously. This idea for the case with $n = 2$ is illustrated in Fig. 7.

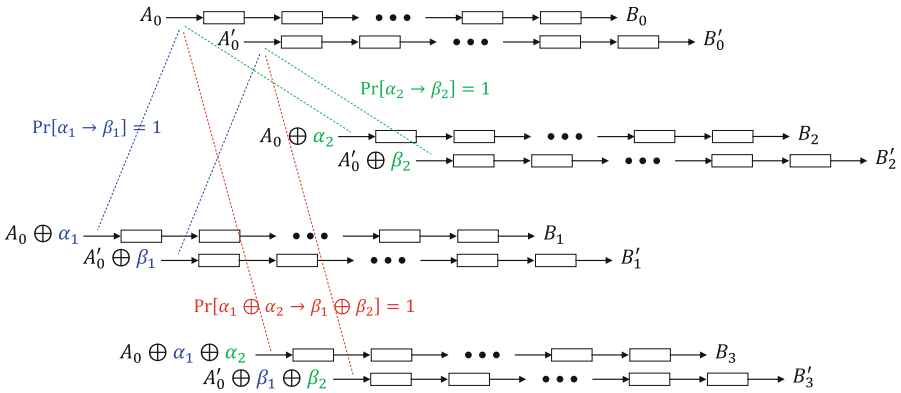

Fig. 7. Attacks on TinyJAMBU-192 with two deterministic differential characteristics.

Note that the previous attack on TinyJAMBU-192 in Sect. 4.4 required adaptively chosen-plaintext queries for not only query chains but also the bit-by-bit key-recovery explained in Sect. 4.3. Currently, we have not found an efficient key-recovery procedure that works in the chosen-plaintext setting. Hence, our approach to recover a 192-bit key is to first identify the valid slid pair and then guess the last 64 key bits. For this reason, we need to filter out all the wrong slid-pair candidates, and it is essential to have $n = 2$ distinct probability 1 characteristics to have a $49 \times 2^2 = 196$-bit filter.

Deterministic Differential Characteristic for P_{192}. In the keyed-permutation of TinyJAMBU, the only non-linear operation is the AND operation between s_{70} and s_{85}. Recall that in each step, the key bit only impacts s_{127}, thus during the first 43 rounds, the input to the AND operation is only dependent on the plaintext. Specifically, given the plaintext value, differential propagation for the first 43 rounds is deterministic. The same can be applied in the backward direction, i.e. given the ciphertext value, differential propagation for the last 70 rounds is deterministic. Moreover, we can set some plaintext and ciphertext bits to 0 to prevent the input difference to AND gates from propagating.

With these observations, we searched for such characteristics for P_{192} by using a refined MILP-based evaluation [14] by adding new constraints to ignore the active AND gates for the first 43 and last 70 rounds from the objective function. As a result, we found many probability 1 differential characteristics.[3] An example is explained in Table 3.

Table 3. An example of probability 1 differential characteristic for TinyJAMBU-192. Differential masks α, β are represented by hexadecimal numbers.

$\alpha : s_{127}, \ldots, s_1, s_0$	0000 0000 0004 0000 0000 0008 0000 0000
$\beta : s_{319}, \ldots, s_{193}, s_{192}$	0000 0008 1000 0000 0080 0000 0004 0000
conditions on plaintext (A_0)	$s_{97} = 0$
conditions on ciphertext (A'_0)	$s_{195} = 0, s_{225} = 0, s_{232} = 0, s_{262} = 0$

AND is active in rounds 12, 125, 140, 160, 177, and these output differences are 0.

We confirmed that the rotated variants of the characteristic in Table 3 are also satisfied with probability 1 for a left rotation by 1, 2, 3, 6, and 7 bits.

Application to TinyJAMBU-192. As mentioned above, using 2 characteristics is sufficient for a 192-bit key. Hence, we use one in Table 3 and its left-rotated version by 1 bit. When we choose 2^{64} distinct values of A_0, we fix $s_{97} = 0$ and $s_{98} = 0$. We also query $A_0 \oplus \alpha$, $A_0 \oplus (\alpha \lll 1)$, and $A_0 \oplus \alpha \oplus (\alpha \lll 1)$ along with A_0. Similarly, when we choose 2^{64} distinct values of A'_0, we fix 8 bits of $s_{195}, s_{225}, s_{232}, s_{262}, s_{196}, s_{226}, s_{233}, s_{263}$ to 0 to satisfy the conditions on the ciphertext, and we also query $A'_0 \oplus \beta$, $A'_0 \oplus (\beta \lll 1)$, and $A'_0 \oplus \beta \oplus (\beta \lll 1)$ along with A'_0. Those would derive a 196-bit filter. Hence, we only have a right slid pair after examining 2^{128} matching candidates. After detecting the slid pair, we exhaustively guess the last 64 key bits.

The complexity is $4 \times 2 \times 2^{64} = 2^{67}$ chosen-plaintext queries. The computational cost is less than $4 \times 2 \times 4 \times 2^{64} = 2^{69}$ computations of P_{192}, which is for computing 4 $\mathcal{R}_i$ or $\mathcal{R}_o$ functions for each query. The memory complexity is to store the queries for A_0 and associated quartets, which is 2^{66}. The memoryless attack is made possible by incurring slightly more computational cost.

[3] Run time was very short. It finished in a few seconds.

A.2 Attacks on Non-multiple Number of Rounds

In our attacks, we assumed that the total number of rounds was a multiple of the key-length, which is the case with $P2$ in all the members of TinyJAMBU. One may wonder that the attack can be prevented by setting the number of rounds to be a non-multiple the key-length. Here, we show that the restriction of the number of rounds to be a multiple of the key-length can easily be lifted for the attacks on P_{128} and P_{192} using the deterministic differential characteristics of Sect. A.1.

Let k be the key of length $klen$ and consider $klen \times m + s$ rounds of encryption for some strictly positive integers m and s. Then, a slid pair (A_0, B_0), (A'_0, B'_0) is such that $A'_0 = P^k_{klen}(A_0)$ and $B'_0 = P^{k \lll s}_{klen}(B_0)$. That is, B'_0 is the encryption of B_0 with $klen$ rounds but with a circular-shifted key. In that setting, one clearly cannot chain queries to enhance a filter because the key schedule does not cycle back to its initial state.

Attacking $klen = 128$ is mostly unchanged from Sect. 3. We simply derive equations on key bits independently for the unshifted and shifted cases that will give us a filter. The only difference is that the 15 unexploitable key bits (bit positions 43 to 57) are shifted in the second case, which can result in at most 30 unexploitable relationships. Nevertheless, we can always build a 98-bit filter and perform a key-recovery with the same complexity as before.

For $klen = 192$, the attack is very similar to Sect. A.1. Indeed, taking the notation of Fig. 7, we can still apply the same filter but only on the outputs $F(B_0, B_1) = F(B'_0, B'_1)$, $F(B_0, B_2) = F(B'_0, B'_2)$, $F(B_0, B_3) = F(B'_0, B'_3)$ and ignoring the relation induced by A_0 and A'_0. The actual shift s has no effect when only comparing relationship on outputs. More generally, in the shifted case, having n independent differential characteristics increase the filter $2^n - 1$ fold (instead of 2^n previously). For the 192-bit key case, a $49 \times 3 = 147$-bit filter is still more than enough to filter all the wrong pairs especially as A_0 and A'_0 can further help us in the guess stage for the remaining key bits.

A.3 Implication on the Security of the AEAD Schemes

Our results do not easily extend to attacks on TinyJAMBU AEAD schemes but bring their security into question. That is, they weaken the rationale to believe 112-bit (resp., 168-bit, or 224-bit) encryption/secret-key security goal being achieved by TinyJAMBU-128 (resp., TinyJAMBU-192, or TinyJAMBU-256); to believe so is essentially equivalent to regarding the security goal itself as an assumption. Neither the security of the primitive nor that of the mode implies security of the scheme; one is assuming that the combination of the two should achieve the security goal even though one is aware of the fact that the primitive is far from being ideal.

In other words, one is assuming that some features of the mode should "enhance" encryption/secret-key security to 112/168/224 bits even though the underlying primitive is vulnerable to birthday-bound (i.e., about 64 bits in any case) key-recovery attacks. The features may include, for example, the fact that

"frame bits" [16][4] are inserted into states and that at most 32 bits of each state value are controllable by adversaries.

In fact, the underlying permutations are already known to be non-ideal. For instance, the designers show in the specifications that $P1$ in the AEAD mode (see Fig. 2) has a differential property of probability 2^{-83}. Nevertheless, we want to state that our attacks are the first to reveal that $P2$ of all the versions of TinyJAMBU is broken by a birthday-bound key-recovery attack, which make us less confident that the security proof of the mode by the designers can be regarded as a convincing reason for the security claim holding.

To be fair, we remark that our results do not significantly affect the privacy security (indistinguishability) shown by the designers or the authentication security goal stated by the designers [16]. This is due to the fact that both of these notions are up to the birthday bound of 64 bits and that our attacks require birthday-bound complexities.

References

1. Bar-On, A., Biham, E., Dunkelman, O., Keller, N.: Efficient slide attacks. J. Cryptol. **31**(3), 641–670 (2018)
2. Bertoni, G., Daemen, J., Peeters, M., Van Assche, G.: Duplexing the sponge: single-pass authenticated encryption and other applications. In: Miri, A., Vaudenay, S. (eds.) SAC 2011. LNCS, vol. 7118, pp. 320–337. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28496-0_19
3. Biham, E., Dunkelman, O., Keller, N.: Improved slide attacks. In: Biryukov, A. (ed.) FSE 2007. LNCS, vol. 4593, pp. 153–166. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-74619-5_10
4. Biryukov, A., Wagner, D.: Slide attacks. In: Knudsen, L. (ed.) FSE 1999. LNCS, vol. 1636, pp. 245–259. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48519-8_18
5. Biryukov, A., Wagner, D.: Advanced slide attacks. In: Preneel, B. (ed.) EUROCRYPT 2000. LNCS, vol. 1807, pp. 589–606. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-45539-6_41
6. Furuya, S.: Slide attacks with a known-plaintext cryptanalysis. In: Kim, K. (ed.) ICISC 2001. LNCS, vol. 2288, pp. 214–225. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45861-1_17
7. Kilian, J., Rogaway, P.: How to protect DES against exhaustive key search (an analysis of DESX). J. Cryptol. **14**(1), 17–35 (2000). https://doi.org/10.1007/s001450010015
8. Naito, Y., Matsui, M., Sugawara, T., Suzuki, D.: SAEB: a lightweight blockcipher-based AEAD mode of operation. IACR Trans. Cryptogr. Hardw. Embed. Syst. **2018**(2), 192–217 (2018)
9. NIST: Submission Requirements and Evaluation Criteria for the Lightweight Cryptography Standardization Process (2018). https://csrc.nist.gov/Projects/lightweight-cryptography

[4] Indeed, the designers argue that the constants in the mode inserted between permutation calls should prevent slide attacks (refer to Fig. 2); it seems that the existence of constants should make it hard to extend our slide attacks to AEAD modes.

10. NIST: Lightweight Cryptography Standardization: Finalists Announced (2021). https://csrc.nist.gov/News/2021/lightweight-crypto-finalists-announced
11. NIST: Status Report on the Second Round of the NIST Lightweight Cryptography Standardization Process (2021). https://csrc.nist.gov/publications/detail/nistir/8369/final
12. van Oorschot, P.C., Wiener, M.J.: Parallel collision search with cryptanalytic applications. J. Cryptol. **12**(1), 1–28 (1999). https://doi.org/10.1007/PL00003816
13. Quisquater, J.-J., Delescaille, J.-P.: How easy is collision search. New results and applications to DES. In: Brassard, G. (ed.) CRYPTO 1989. LNCS, vol. 435, pp. 408–413. Springer, New York (1990). https://doi.org/10.1007/0-387-34805-0_38
14. Saha, D., Sasaki, Y., Shi, D., Sibleyras, F., Sun, S., Zhang, Y.: On the security margin of TinyJAMBU with refined differential and linear cryptanalysis. IACR Trans. Symmetric Cryptol. **2020**(3), 152–174 (2020)
15. Wu, H., Huang, T.: TinyJAMBU: A Family of Lightweight Authenticated Encryption Algorithms. Submitted to NIST, September 2019
16. Wu, H., Huang, T.: TinyJAMBU: A Family of Lightweight Authenticated Encryption Algorithms (Version 2). Submitted to NIST, May 2021