Differential Attacks against Stream Cipher ZUC*

Hongjun Wu, Tao Huang, Phuong Ha Nguyen, Huaxiong Wang, and San Ling

Division of Mathematical Sciences,
School of Physical and Mathematical Sciences
Nanyang Technological University, Singapore
{wuhj,huangtao,ng007ha,hxwang,lingsan}@ntu.edu.sg

Abstract. Stream cipher ZUC is the core component in the 3GPP confidentiality and integrity algorithms 128-EEA3 and 128-EIA3. In this paper, we present the details of our differential attacks against ZUC 1.4. The vulnerability in ZUC 1.4 is due to the non-injective property in the initialization, which results in the difference in the initialization vector being cancelled. In the first attack, difference is injected into the first byte of the initialization vector, and one out of $2^{15.4}$ random keys result in two identical keystreams after testing $2^{13.3}$ IV pairs for each key. The identical keystreams pose a serious threat to the use of ZUC 1.4 in applications since it is similar to reusing a key in one-time pad. Once identical keystreams are detected, the key can be recovered with average complexity $2^{99.4}$. In the second attack, difference is injected into the second byte of the initialization vector, and every key can result in two identical keystreams with about 2^{54} IVs. Once identical keystreams are detected, the key can be recovered with complexity 2^{67}. We have presented a method to fix the flaw by updating the LFSR in an injective way in the initialization. Our suggested method is used in the later versions of ZUC. The latest ZUC 1.6 is secure against our attacks.

1 Introduction

Comparing to block ciphers, dedicated stream ciphers normally require less computation for achieving the same security level. Stream ciphers are widely used in applications. For example, RC4 [10] is used in SSL and WEP, and A5/1 [8] is used in GSM (the Global System for Mobile Communications). But the use of RC4 in WEP is insecure [7], and A5/1 is very weak [4]. ECRYPT (2004–2008) has organised the eSTREAM competition, which stimulated the study on stream ciphers, and a number of new stream ciphers were proposed [1–3, 5, 6, 9, 15].

The 3rd Generation Partnership Project (3GPP) was set up for making globally applicable 3G mobile phone system specifications based on the GSM specifications. Stream cipher ZUC was designed by the Data Assurance and Communication Security Research Center of the Chinese Academy of Sciences.

* This research is supported by the National Research Foundation Singapore under its Competitive Research Programme (CRP Award No. NRF-CRP2-2007-03) and Nanyang Technological University NAP startup grant (M4080529.110).

X. Wang and K. Sako (Eds.): ASIACRYPT 2012, LNCS 7658, pp. 262–277, 2012.

It is the core component of the 3GPP Confidentiality and Integrity Algorithms 128-EEA3 & 128-EIA3 which were proposed for inclusion in the "4G" mobile standard LTE (Long Term Evolution). In July 2010, the ZUC 1.4 [11] was made public for evaluation. We developed two key recovery attacks against ZUC 1.4 [16], and our attacks directly led to the tweak of ZUC 1.4 into ZUC 1.5 [12] in Jan 2011. (Note that it was reported independently in [14] that the non-injective initialization of ZUC 1.4 may result in identical keystreams.) The latest version, ZUC 1.6 [13], was released in June 2011 (ZUC 1.6 and ZUC 1.5 have almost the same specifications).

In this paper, we present the details of our differential attacks against ZUC 1.4. Our attacks against ZUC is similar to the differential attacks against Py, Py6 and Pypy [17], in which different IVs result in identical keystreams. In the first attack against ZUC 1.4, the difference is at the first byte of the IV, and one in $2^{15.4}$ keys results in identical keystreams after testing $2^{13.3}$ IV pairs for each key. Once identical keystreams are detected, the key can be recovered with complexity $2^{99.4}$. In the second attack against ZUC 1.4, the difference is at the second byte of the IV, and identical keystreams can be obtained after testing 2^{54} IVs. The key can be recovered with complexity 2^{67}.

This paper is organized as follows. The notations and the description of ZUC 1.4 are give in Sect. 2. The overview of the attack is is given in Sect. 3. In Section 4 and 5, we present the key recovery attack with difference at the first byte and the second byte of IV, respectively. We suggest the tweak to fix the flaw in Sect. 6. Section 7 concludes the paper.

2 Preliminaries

2.1 The Notations

In this paper, we follow the notations used in the ZUC specifications [11].

$+$	The addition of two integers
$\oplus$	The bit-wise exclusive-or operation of integers
$\boxplus$	The modulo 2^{32} addition
ab	The product of integers a and b
$a\|\|b$	The concatenation of a and b
$a \lll k$	The k-bit cyclic shift of a to the left
$a \ggg k$	The k-bit cyclic shift of a to the right
$a \gg k$	The k-bit right shift of integer a
a_H	The most significant 16 bits of integer a
a_L	The least significant 16 bits of integer a
$(a_1, a_2, \ldots, a_n) \rightarrow (b_1, b_2, \ldots, b_n)$	It assigns the values of a_i to b_i in parallel

0_n	The sequence of n bits 0
1_n	The sequence of n bits 1
$\bar{y}$	The bitwise complement of y

An integer a can be written in different formats. For example,

$$\begin{aligned} a &= 25 && \text{decimal representation} \\ &= 0x19 && \text{hexadecimal representation} \\ &= 00011001_2 && \text{binary representation} \end{aligned}$$

We number the least significant bit with 1 and use $A[i]$ to denote the ith bit of a A. And use $B[i..j]$ to denote the bit i to bit j of B.

2.2 The General Structure of ZUC 1.4

ZUC is a word-oriented stream cipher with 128-bit secret key and a 128-bit initial vector. It consists of three main components: the linear feedback shift register (LFSR), the bit-reorganization (BR) and a nonlinear function F. The general structure of the algorithm is illustrated in Fig. 1.

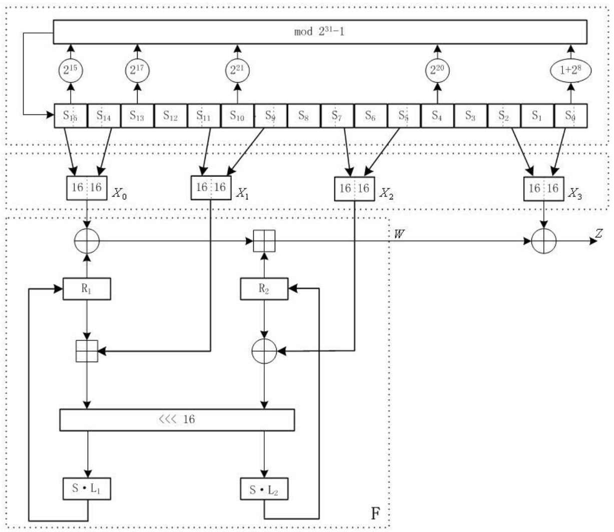

Fig. 1. General structure of ZUC

Linear Feedback Shift Register(LFSR). It consists of sixteen 31-bit registers s_0, s_1, ..., s_{15}, and each register is an integer in the range $\{1, 2, \ldots, 2^{31} - 1\}$. During the keystream generation stage, the LFSR is updated as follows:

LFSRUpdate():

1. $s_{16} = (2^{15}s_{15} + 2^{17}s_{13} + 2^{21}s_{10} + 2^{20}s_4 + (1 + 2^8)s_0)\mathrm{mod}(2^{31} - 1)$;
2. If $s_{16} = 0$ then set $s_{16} = 2^{31} - 1$;
3. $(s_1, s_2, \ldots, s_{15}, s_{16}) \to (s_0, s_1, \ldots, s_{14}, s_{15})$.

Bit-Reorganization Function. It extracts 128 bits from the state of the LFSR and forms four 32-bit words X_0, X_1 X_2 and X_3 as follows:

Bitreorganization():

1. $X_0 = s_{15H}||s_{14L}$;
2. $X_1 = s_{11L}||s_{9H}$;
3. $X_2 = s_{7L}||s_{5H}$;
4. $X_3 = s_{2L}||s_{0H}$;

Nonlinear Function *F*. It contains two 32-bit memory words R_1 and R_2. The description of F is given below. In function F, S is the Sbox layer and L_1 and L_2 are linear transformations as defined in [11]. The output of function F is a 32-bit word W. The keystream word Z is given as $Z = W \oplus X_3$.

$F(X_0, X_1, X_2)$:

1. $W = (X_0 \oplus R_1) \boxplus R_2$;
2. $W_1 = R_1 \boxplus X_1$;
3. $W_2 = R_2 \oplus X_2$;
4. $R_1 = S(L_1(W_{1L}||W_{2H}))$;
5. $R_2 = S(L_2(W_{2L}||W_{1H}))$;

2.3 The Initialization of ZUC 1.4

The initialization of ZUC 1.4 consists of two steps: loading the key and IV into the register, and running the cipher for 32 steps with the keystream word being used to update the state.

Key and IV Loading. Denote the 16 key bytes as k_i $(0 \leq i \leq 15)$, the 16 IV bytes as iv_i $(0 \leq i \leq 15)$. We load the key and IV into the register as: $s_i = (k_i||d_i||iv_i)$. The values of the constants d_i are given in [11]. The two memory words R_1 and R_2 in function F are set as 0.

Running the Cipher for 32 Steps. At the initialization stage, the keystream word Z is used to update the LFSR as follows:

LFSRWithInitialisationMode(u):

1. $v = (2^{15}s_{15} + 2^{17}s_{13} + 2^{21}s_{10} + 2^{20}s_4 + (1 + 2^8)s_0)\mathrm{mod}(2^{31} - 1)$;
2. If $v = 0$ then set $v = 2^{31} - 1$;
3. $s_{16} = v \oplus u$;
4. If $s_{16} = 0$ then set $s_{16} = 2^{31} - 1$;
5. $(s_1, s_2, \ldots, s_{15}, s_{16}) \to (s_0, s_1, \ldots, s_{14}, s_{15})$.

The cipher runs for 32 steps at the initialization stage as follows:

InitializationStage():

for $i = 0$ **to** 31 {

1. Bitreorganization();
2. $Z = F(X_0, X_1, X_2) \oplus X_3$;
3. LFSRWithInitialisationMode($Z >> 1$) .

}

3 Overview of the Attacks

We notice that the LFSR in ZUC is defined over $GF(2^{31}-1)$, with the element 0 being replaced with $2^{31}-1$. To the best of our knowledge, it is the first time that $GF(2^{31}-1)$ is used in the design of stream cipher. In the initialization of ZUC 1.4, we notice that XOR is involved in the update of LFSR ($s_{16} = v \oplus u$). When XOR is applied to the elements in $GF(2^{31}-1)$, we obtain the following undesirable property:

Property 1. Suppose that a and a' are two elements in $GF(2^{31}-1)$, $a \neq a'$, and $\bar{a} = a'$. If $b = a$ or $b = \bar{a}$, then $a \oplus b \bmod (2^{31}-1) = a' \oplus b \bmod (2^{31}-1) = 0$.

The above property shows that the difference between a and a' can get eliminated with an XOR operation! In the rest of this paper, we exploit this property to attack ZUC 1.4 by eliminating the difference in the state.

In our attacks, we try to eliminate the difference in the state without the difference in the state being injected into the nonlinear function F. The reason is that if a difference is injected into F, then Sboxes would be involved, and the difference would remain in F until additional difference being injected into F, thus the probability that the difference in the state being eliminated would get significantly reduced.

We now investigate what are the IV differences that would result in the difference in the state being eliminated with high probability. The IV differences are classified into the following three types:

Type 1. $\Delta iv_i \neq 0$ for at least one value of i ($7 \leq i \leq 15$).

After loading this type of IVs into LFSR, the difference would appear at the least significant byte of at least one of the LFSR elements s_7, s_8, $\cdots$, s_{15}. Note that the least significant byte of s_7 is part of X_2 in the Bit-reorganization function since $X_2 = s_{7L} || s_{5H}$, and X_2 is an input to function F. Due to the shift of LFSR, the difference at the least significant byte of s_7, s_8, $\cdots$, s_{15} would be injected into F. Thus we would not use this type of IV difference in our attacks.

Type 2. $\Delta iv_i = 0$ for $7 \leq i \leq 15$, $\Delta iv_i \neq 0$ for at least one value of i ($2 \leq i \leq 6$). After loading this type of IVs into LFSR, the difference would appear at the least significant byte of at least one of the LFSR elements $s_2, s_3, \cdots, s_6$. Note that the least significant byte of s_2 is part of X_3 in the Bit-reorganization function since $X_3 = s_{2L}||s_{0H}$, X_3 is XORed with the output of F to generate keystream word Z, and Z is used to update the LFSR. Two steps later, the difference in iv_2 would appear in the feedback function to update LFSR. It means that if there is difference in iv_2, the difference in s_2 would be used to update the LFSR twice, and the probability that the difference would be eliminated is very small. Due to the shift of LFSR, the difference at $s_2, s_3, \cdots, s_7$ would be eliminated with very small probability. Thus we did not use this type of IV difference in our attacks.

Type 3. $\Delta iv_i = 0$ for $2 \le i \le 15$, $\Delta iv_0 \ne 0$ or $\Delta iv_1 \ne 0$.
The focus of our attacks is on this type of IV differences. In order to increase the chance of success, we consider the difference at only one byte of the IV. We discuss below how the difference in the state can be eliminated when there is difference in s_0 (the analysis for the difference in s_1 is similar). At the first step in the initialization,

$$s_0 = (k_0||d_0||iv_0)\,, \tag{1}$$
$$v = 2^{15}s_{15} + 2^{17}s_{13} + 2^{21}s_{10} + 2^{20}s_4 + (1+2^8)s_0 \mod (2^{31}-1)\,, \tag{2}$$
$$s_{16} = v \oplus u\,. \tag{3}$$

Suppose that the difference is only at iv_0, and $iv_0 - iv_0' = \Delta iv_0 > 0$. From (1) and (2) we know that

$$\begin{aligned} v - v' &= (1+2^8)(iv_0 - iv_0') \mod (2^{31}-1) \\ &= \Delta iv_0 \parallel \Delta iv_0\,. \end{aligned} \tag{4}$$

If we need to eliminate the difference in s_{16}, from Property 1 and (3), the following condition should be satisfied:

$$v \oplus v' = 1_{31} \tag{5}$$
$$u = v \quad \text{or} \quad u = v' \tag{6}$$

According to (5), v and v' have XOR difference in the left-most 15 bits (i.e.$v[17..31]$ and $v'[17..31]$), while according to (4), the subtraction difference of those bits are 0. The only possible reason is that the 15 bits, $v[17..31]$, are all affected by the carries from the addition of Δiv_0 to v'. After testing all the one-byte differences, we found that v must be in one of the following four forms (the values of v and v' can be swapped):

$$\begin{aligned} & v = 111111111111111_2 \parallel y \parallel 1_2 \parallel y \\ \text{or} \quad & v = 011111111111111_2 \parallel y \parallel 0_2 \parallel y \\ \text{or} \quad & v = 000000000000000_2 \parallel \bar{y} \parallel 0_2 \parallel \bar{y} \\ \text{or} \quad & v = 100000000000000_2 \parallel \bar{y} \parallel 1_2 \parallel \bar{y} \\ & \quad (y \text{ is a 7-bit integer.}) \end{aligned} \tag{7}$$

There are 510 possible values of v ($v = 1_{31}$ and $v = 0_{31}$ are excluded since one of v and $\bar{v}$ cannot be 0). All the $(v,\ v')$ pairs and their differences are given in Table 1 in Appendix A. Notice that we ignored the order of v and v' as they are exchangeable. We have obtained all the possible values of v and u for generating identical keystreams.

We highlight the following property in the table: the difference between v and v' uniquely determines the value of pair (v, v') in the table. As a result, if we know the difference of IVs that results in the collision of the state, we can determine the value of (v, v') immediately.

By eliminating the difference in the state as illustrated above, we developed two attacks against ZUC 1.4. The first attack is to exploit the difference at iv_0, and the second attack is to exploit the difference at iv_1. The details are given in the following two sections.

4 Attack ZUC 1.4 with Difference at iv_0

In this section, we present our first differential attack on the initialization by using IV difference at iv_0 and generating identical keystream. The keys that generate the same keystream are called weak keys in this attack. We will show that a weak key exists with probability $2^{-15.4}$, and a weak key can be detected with about $2^{13.3}$ chosen IVs. Once a weak key is detected, its effective key size is reduced from 128 bits to around 100 bits.

4.1 The Weak Keys for Δiv_0

We will show that when there is difference at iv_0, about one in $2^{15.4}$ keys would result in identical keystream. For a random key, we will check whether there exists a pair of IVs such that (5), (6) and (7) can be satisfied.

We start with analyzing how keys and IVs are involved in the expression of u and v in the first step of initialization. From the specifications of the initialization, we have

$$\begin{aligned} u =& Z \gg 1 = (X_0 \oplus X_3) \gg 1 = ((s_{15H} || s_{14L}) \oplus (s_{2L} || s_{0H})) \gg 1 \\ =& ((k_{15} \parallel iv_2 \parallel k_0 \parallel iv_{14}) \oplus \text{0x6b8f9a89}) \gg 1 \end{aligned} \tag{8}$$

In (2) and (8), there are 5 bytes of key, $\{k_0, k_4, k_{10}, k_{13}, k_{15}\}$, and 7 bytes of IV, $\{iv_0, iv_2, iv_4, iv_{10}, iv_{13}, iv_{14}, iv_{15}\}$ being involved in the computation of u and v. The complexity would be very high if we directly try all possible combinations of the keys and IVs. However, with analysis on the expressions of u and v, we can reduce the search space from 2^{96} to around $2^{26.3}$.

Solve (5), (6), (7) and (8), we obtain the following four groups of solutions:

Group 1.

$$\begin{aligned} u = v &= 1111111111111111_2 \parallel y \parallel 1_2 \parallel y \\ k_{15} &= \text{0x94} \\ iv_2 &= \text{0x70} \\ k_0 &= \text{0x9a} \oplus (y \parallel 1_2) \\ iv_{14} \gg 1 &= \text{0x44} \oplus y \end{aligned} \tag{9}$$

Group 2.

$$\begin{aligned} u = v &= 0111111111111111_2 \parallel y \parallel 0_2 \parallel y \\ k_{15} &= \text{0x14} \\ iv_2 &= \text{0x70} \\ k_0 &= \text{0x9a} \oplus (y \parallel 0_2) \\ iv_{14} \gg 1 &= \text{0x44} \oplus y \end{aligned} \tag{10}$$

Group 3.

$$\begin{aligned} u = v &= 0000000000000000_2 \parallel \bar{y} \parallel 0_2 \parallel \bar{y} \\ k_{15} &= \text{0x6b} \\ iv_2 &= \text{0x8f} \\ k_0 &= \text{0x9a} \oplus (\bar{y} \parallel 0_2) \\ iv_{14} \gg 1 &= \text{0xbb} \oplus \bar{y} \end{aligned} \tag{11}$$

Group 4.

$$\begin{aligned} u = v &= 1000000000000000_2 \parallel \bar{y} \parallel 1_2 \parallel \bar{y} \\ k_{15} &= \text{0xeb} \\ iv_2 &= \text{0x8f} \\ k_0 &= \text{0x9a} \oplus (\bar{y} \parallel 1_2) \\ iv_{14} \gg 1 &= \text{0xbb} \oplus \bar{y} \end{aligned} \tag{12}$$

Furthermore, from (2) we compute v as follows (note that the property $2^k s_i \bmod (2^{31}-1) = s_i \lll k$):

$$\begin{aligned} v = &(1+2^{23})k_0 + 2^7 k_{15} + 2^9(k_{13} + 2^3 k_4 + 2^4 k_{10}) + (1+2^8) iv_0 \\ &+ 2^{15}(iv_{15} + 2^2 iv_{13} + 2^5 iv_4 + 2^6 iv_{10}) + \text{0x451bfe1b} \mod (2^{31}-1) \end{aligned} \tag{13}$$

Let $sum_1 = k_{13} + 2^3 k_4 + 2^4 k_{10}$, $sum_2 = iv_{15} + 2^2 iv_{13} + 2^5 iv_4 + 2^6 iv_{10}$. The value of sum_1 ranges from 0 to 6375, and the value of sum_2 ranges from 0 to 25755. We developed Algorithm 1 to search for weak keys.

Algorithm 1. Find weak keys for Δiv_0

```
for (k15, iv2) in each of the 4 groups of solutions (9), (10), (11), (12) do
    for y = 0 to 127 do
        determine iv14 >> 1 and k0
        for sum1 = 0 to 6375 do
            for iv0 = 0 to 255 do
                keySum ← 2^7 k15 + (2^23 + 1)k0 + 2^9 sum1  mod (2^31 − 1)
                sum2 ← (u − keySum − (1 + 2^8)iv0 − 0x451bfe1b)/2^15  mod (2^31 − 1)
                if sum2 is less than 25756 then
                    v = u; v' = u ⊕ 1_32;
                    if (v − v') mod (2^31 − 1) is a multiple of 1 + 2^8 then
                        Δiv0 = (v − v')  mod (2^31 − 1)/(1 + 2^8);
                        iv0' = iv0 − Δiv0;
                    else
                        Δiv0 = (v' − v)  mod (2^31 − 1)/(1 + 2^8);
                        iv0' = iv0 + Δiv0;
                    end if
                    output u, k0, k15, sum1, iv0, iv0', iv2, iv14 >> 1, sum2
                end if
            end for
        end for
    end for
end for
```

Each output from Algorithm 1 gives the value of (k_{15}, k_0, sum_1, iv_0, iv'_0, iv_2, iv_{14}, sum_2) that results in identical keystreams. Running Algorithm 1, we found $9934 = 2^{13.28}$ different outputs. We note that on average, each sum_1 from the output of the algorithm represents $2^{24}/6376 = 2^{11.36}$ possible choices of (k_4, k_{10}, k_{13}). Thus there are $2^{13.3} \times 2^{11.4} = 2^{24.7}$ weak values of $(k_0, k_4, k_{10}, k_{13}, k_{15})$. Hence, there are $2^{24.7}$ weak keys out of 2^{40} possible values of the 5 key bytes. The probability that a random key is weak for IV difference at iv_0 is $2^{-15.4}$. The complexity of Algorithm 1 is $4 \times 128 \times 6376 \times 256 = 2^{26.3}$.

Identical Keystreams. We give below a weak key and an IV pair with difference at iv_0 that result in identical keystreams.

$key =$ 87,4,95,13,161,32,199,61,20,147,56,84,126,205,165,148

$IV =$ 166,166,112,38,192,214,34,211,170,25,18,71,4,135,68,5

$IV' =$ 116,166,112,38,192,214,34,211,170,25,18,71,4,135,68,5

For both IV and IV', the identical keystreams are: 0xbfe800d5 0360a22b 6c4554c8 67f00672 2ce94f3f f94d12ba 11c382b3 cbaf4b31. . ..

4.2 Detecting Weak Keys for Δiv_0

We have shown above that a random key is weak with probability $2^{-15.4}$. In the attack against ZUC, we will first detect a weak key, then recover it. To detect a weak key, our approach is to use the IV pairs generated from Algorithm 1 to test whether identical keystreams are generated. Note that for a particular value of sum_2, we can always find a combination of $(iv_4, iv_{10}, iv_{13}, iv_{15}\}$ that satisfies $sum_2 = iv_{15} + 2^2 iv_{13} + 2^5 iv_4 + 2^6 iv_{10}$. Thus a pair of IVs $(iv_0, iv_2, iv_4, iv_{10}$, iv_{13}, iv_{14}, $iv_{15})$ and $(iv'_0, iv_2, iv_4, iv_{10}, iv_{13}, iv_{14}, iv_{15})$ can be determined by each output of Algorithm 1. Using this result, we developed Algorithm 2 to detect weak keys for Δiv_0.

Algorithm 2. Detecting weak keys for Δiv_0

1. Choose one of the $2^{13.28}$ outputs of Algorithm 1.
2. Find the pair of IVs determined by this output (if iv_j does not appear in the first initialization step, set it as some fixed constant).
3. Use the IV pair to generate two key steams.
4. If the keystreams are identical, output the IVs and conclude the key is weak.
5. If all outputs of Algorithm 1 have been checked, and there are no identical keystreams, we conclude that the key is not weak.

In Algorithm 2, we need to test at most $2^{13.3}$ pairs of IVs to determine if a key is weak for difference at iv_0.

4.3 Recovering Weak Keys for Δiv_0

After detecting a weak key, we proceed to recover the weak key. Once a key is detected as weak (as given from Algorithm 2), from the IV pair being used to generate identical keystreams, we immediately know the value of k_0, k_{15} and sum_1. Note that $sum_1 = (k_{13} + 2^3 k_4 + 2^4 k_{10})$. In the best situations, the sum is 0 or 25755, then we can uniquely determine k_4, k_{10} and k_{13}. In the worst situation, there are 2^{12} possible choices for k_4, k_{10} and k_{13}, and therefore, we need 2^{12} tests to determine the correct values for k_4, k_{10} and k_{13}. On average, for each value of sum_1, we need to test $2^{11.4}$ combinations of (k_4, k_{10}, k_{13}).

Since there are only five key bytes being recovered in our attack, the remaining 11 key bytes should be recovered with exhaustive search. Hence, the complexity to recover all key bits is $2^{88} \times 2^{11.4} = 2^{99.4}$. From the analysis above, we also know that the best complexity is 2^{88} and the worst complexity is 2^{100}.

5 Attack ZUC 1.4 with Difference at iv_1

In this section, we present the differential attack on ZUC 1.4 for IV difference at iv_1. Different from the attack in Section 4, we need to consider the computation of u and v in the second step of the initialization. For this type of IV difference, for every key, there are some IV pairs that result in identical keystreams since more IV bytes are involved. Once we found such an IV pair, we can recover the key with complexity around 2^{67}.

5.1 Identical Keystreams for Δiv_1

The computation of u and v in the second initialization step involves more key and IV bytes. The v in the second initialization step is computed as:

$$\begin{aligned} v &= (2^{15}s_{16} + 2^{17}s_{14} + 2^{21}s_{11} + 2^{20}s_5 + (1+2^8)s_1) \bmod (2^{31}-1), \\ s_{16} &= ((2^{15}s_{15} + 2^{17}s_{13} + 2^{21}s_{10} + 2^{20}s_4 + (1+2^8)s_0) \bmod (2^{31}-1)) \\ &\quad \oplus (((k_{15} \parallel iv_2 \parallel k_0 \parallel iv_{14}) \oplus 0x6b8f9a89) >> 1) \end{aligned} \tag{14}$$

And u is given as:

$$\begin{aligned} u &= (((X_0 \oplus R_1) + R_2) \oplus X_3) >> 1 \\ X_0 &= (s_{16H}||10101100_2||iv_{15}) \\ X_3 &= (01011110_2||iv_3||k_1||01001101_2) \\ R_1 &= S(L_1(s_{9H}||s_{7L})) = f_1(iv_7, k_9) \\ R_2 &= S(L2(s_{5H}||s_{11L})) = f_2(iv_{11}, k_5) \end{aligned} \tag{15}$$

where f_1 and f_2 are some deterministic non-linear functions.

There are 10 IV bytes involved in the expression of v, i.e. (iv_0, iv_1, iv_2, iv_4, iv_5, iv_{10}, iv_{11}, iv_{13}, iv_{14}, iv_{15}) and 8 IV bytes involved in the expression of u, i.e. (iv_0, iv_3, iv_4, iv_7, iv_{10}, iv_{11}, iv_{13}, iv_{15}). In total, there are 12 IV bytes being involved in the computation of u and v, and every bit of u and v can be affected by IV. We conjecture that for every key, the conditions (5) and (6) can be satisfied, and identical keystreams can be generated. To verify it, we tested 1000 random keys. Our experimental results show that there is always an IV pair for each key that results in identical keystreams.

In the attack, a random key and a random iv pair with difference at iv_1, the probability that v and u satisfy the conditions (5) and (6) is $2^{-31} \times 2^{-31} \times 2 = 2^{-61}$. Choosing 2^8 ivs with difference at iv_1, we have around 2^{15} pairs. The identical keystream pair appears with probability $2^{-61+15} = 2^{-46}$ with 2^8 IVs. We thus need about $2^{46} \times 2^8 = 2^{54}$ IVs to obtain identical keystreams.

Identical Keystreams. We give below a key and an IV pair with difference at iv_1 that result in identical keystreams. The algorithm being used to find the IV pair is given in Appendix B. The algorithm is a bit complicated since a number of optimization tricks are involved. The explanation of the optimization details is omitted here since our focus is to develop a key recovery attack.

key = `123,149,193,87,42,150,117,4,209,101,85,57,46,117,49,243`
IV = `92,80,241,10,0,217,47,224,48,203,0,45,204,0,0,17`
IV' = `92,182,241,10,0,217,47,224,48,203,0,45,204,0,0,17`

The identical keystreams are: 0xf09cc17d 41f12d3f 453ac0c3 cadcef9f f98fb964 ca6e576e b48b813 6c43da22 ....

5.2 Key Recovery for Δiv_1

After identical keystreams are generated from an IV pair with difference at iv_1, we proceed to recover the secret key. From Table 1 in Appendix A, we know the value of (v, v') since we know the difference at iv_1 of the chosen IV pair, and we also know the value of u since $u = v$ or $u = v'$. In the following, we illustrate a key recovery attack after identical keystreams have been detected.

1. In the expression of u in (15), (k_1, k_5, k_9, s_{16H}) is involved. Note that there are only two possible values of the 31-bit u. We try all the possible values of (k_1, k_5, k_9, s_{16H}), then there would be $2^{8\times3+16} \times 2^{-31} \times 2 = 2^{10}$ possible values of (k_1, k_5, k_9, s_{16H}) that generate the two possible values of u. The complexity of this step is 2^{40}.
2. Next we use the expression of s_{16} in (14). For each of the 2^{10} possible values of (k_1, k_5, k_9, s_{16H}), we try all the possible values of (k_0, k_4, k_{10}, k_{13}, k_{15}) and check whether the values of s_{16H} is computed correctly or not. There would be $2^{8\times5} \times 2^{-16} = 2^{24}$ possible values of (k_0, k_4, k_{10}, k_{13}, k_{15}) left. Considering that there are 2^{10} possible values of (k_1, k_5, k_9, s_{16H}), about $2^{10} \times 2^{24} = 2^{34}$ possible values of (k_0, k_1, k_4, k_5, k_9, k_{10}, k_{13}, k_{15}, s_{16H}) remain. The complexity of this step is $2^{8\times5} \times 2^{10} = 2^{50}$.
3. Then we use the expression of v in (14). For each of the 2^{34} possible values of (k_0, k_1, k_4, k_5, k_9, k_{10}, k_{13}, k_{15}, s_{16H}), we try all the possible values of (k_{11}, k_{14}) and check whether the value of v is correct or not. A random value of (k_{11}, k_{14}) would pass the test with probability $2^{8\times2} \times 2^{-31} = 2^{-15}$ Considering that there are 2^{34} possible values of (k_0, k_1, k_4, k_5, k_9, k_{10}, k_{13}, k_{15}, s_{16H}), about $2^{34} \times 2^{-15} = 2^{19}$ possible values of (k_0, k_1, k_4, k_5, k_9, k_{10}, k_{11}, k_{13}, k_{14}, k_{15}) remain. The complexity of this step is $2^{8\times2} \times 2^{34} = 2^{50}$.
4. For each of the 2^{19} possible values of (k_0, k_1, k_4, k_5, k_9, k_{10}, k_{11}, k_{13}, k_{14}, k_{15}), we recover the remaining 6 key bytes (k_2,k_3,k_6,k_7,k_8,k_{12}) by exhaustive search. The complexity of this step is $2^{19} \times 2^{8\times6} = 2^{67}$.

The overall computational complexity to recover a key is $2^{40} + 2^{50} + 2^{50} + 2^{67} \approx 2^{67}$. And we need about 2^{54} IVs in the attack. Note that the complexity in the first, second and third steps can be significantly reduced with optimization since we are dealing with simple functions. For example, meet-in-the-middle attack can be used in the first step, and the sum of a few key bytes can be considered in the second and third steps. However, the complexity of those three steps has little effect on the overall complexity of the attack, so we do not present the details of the optimization here.

6 Improving ZUC 1.4

From the analysis in Sect. 3, the weakness of the initialization comes from the non-injective update of the LFSR. To fix the flaw, we proposed the tweak in the rump session of Asiacrypt 2010. Instead of using the XOR operation, it is better to use addition modulo operation over $GF(2^{31} - 1)$. More specifically, the operation $s_{16} = v \oplus u$ is changed to $s_{16} = v + u \mod (2^{31} - 1)$. With this tweak, the difference in v would always result in the difference in s_{16} if there is no difference in u, and the attack against ZUC 1.4 can no longer be applied. In the later versions ZUC 1.5 and 1.6 (ZUC 1.5 and 1.6 have almost the same specifications), the computation of s_{16} is modified using our suggested method.

7 Conclusion

In this paper, we developed two chosen IV attacks against the initialization of ZUC 1.4. In our attacks, identical keystreams are generated from different IVs, then key recovery attacks are applied. Our attacks are independent of the number of steps in initialization. The lesson from this paper is that when non-injective functions are used in cipher design, we should pay special attention to ensure that the difference cannot be eliminated with high probability.

References

1. Babbage, S., Dodd, M.: The MICKEY Stream Ciphers. In: Robshaw, M., Billet, O. (eds.) New Stream Cipher Designs. LNCS, vol. 4986, pp. 191–209. Springer, Heidelberg (2008)
2. Berbain, C., Billet, O., Canteaut, A., Courtois, N.T., Gilbert, H., Goubin, L., Gouget, A., Granboulan, L., Lauradoux, C., Minier, M., Pornin, T., Sibert, H.: Sosemanuk, a Fast Software-Oriented Stream Cipher. In: Robshaw, M., Billet, O. (eds.) New Stream Cipher Designs. LNCS, vol. 4986, pp. 98–118. Springer, Heidelberg (2008)
3. Bernstein, D.J.: The Salsa20 Family of Stream Ciphers. In: Robshaw, M., Billet, O. (eds.) New Stream Cipher Designs. LNCS, vol. 4986, pp. 84–97. Springer, Heidelberg (2008)
4. Biryukov, A., Shamir, A., Wagner, D.: Real Time Cryptanalysis of A5/1 on a PC. In: Schneier, B. (ed.) FSE 2000. LNCS, vol. 1978, pp. 1–44. Springer, Heidelberg (2001)
5. Boesgaard, M., Vesterager, M., Zenner, E.: The Rabbit Stream Cipher. In: Robshaw, M., Billet, O. (eds.) New Stream Cipher Designs. LNCS, vol. 4986, pp. 69–83. Springer, Heidelberg (2008)
6. De Cannière, C., Preneel, B.: Trivium. In: Robshaw, M., Billet, O. (eds.) New Stream Cipher Designs. LNCS, vol. 4986, pp. 244–266. Springer, Heidelberg (2008)
7. Fluhrer, S., Mantin, I., Shamir, A.: Weaknesses in the Key Scheduling Algorithm of RC4. In: Vaudenay, S., Youssef, A.M. (eds.) SAC 2001. LNCS, vol. 2259, pp. 1–24. Springer, Heidelberg (2001)
8. Golić, J.D.: Cryptanalysis of Alleged A5 Stream Cipher. In: Fumy, W. (ed.) EUROCRYPT 1997. LNCS, vol. 1233, pp. 239–255. Springer, Heidelberg (1997)
9. Hell, M., Johansson, T., Maximov, A., Meier, W.: The Grain Family of Stream Ciphers. In: Robshaw, M., Billet, O. (eds.) New Stream Cipher Designs. LNCS, vol. 4986, pp. 179–190. Springer, Heidelberg (2008)
10. Rivest, R.L.: The RC4 Encryption Algorithm. RSA Data Security, Inc. (March 1992)

11. ETSI/SAGE Specification. Specification of the 3GPP Confidentiality and Integrity Algorithms 128-EEA3 & 128-EIA3. Document 2: ZUC Specification; Version: 1.4 (July 30, 2010)
12. ETSI/SAGE Specification. Specification of the 3GPP Confidentiality and Integrity Algorithms 128-EEA3 & 128-EIA3. Document 2: ZUC Specification; Version: 1.5 (January 4, 2011)
13. ETSI/SAGE Specification. Specification of the 3GPP Confidentiality and Integrity Algorithms 128-EEA3 & 128-EIA3. Document 2: ZUC Specification; Version: 1.6 (June 28, 2011)
14. Sun, B., Tang, X., Li, C.: Preliminary Cryptanalysis Results of ZUC. In: First International Workshop on ZUC Algorithm, vol. 12 (2010)
15. Wu, H.: The Stream Cipher HC-128. In: Robshaw, M., Billet, O. (eds.) New Stream Cipher Designs. LNCS, vol. 4986, pp. 39–47. Springer, Heidelberg (2008)
16. Wu, H., Nguyen, P.H., Wang, H., Ling, S.: Cryptanalysis of the Stream Cipher ZUC in the 3GPP Confidentiality & Integrity Algorithms 128-EEA3 & 128-EIA3. In: Rump Session of Asiacrypt 2010 (2008)
17. Wu, H., Preneel, B.: Differential Cryptanalysis of the Stream Ciphers Py, Py6 and Pypy. In: Naor, M. (ed.) EUROCRYPT 2007. LNCS, vol. 4515, pp. 276–290. Springer, Heidelberg (2007)

A The List of Possible v and v' for Collision

Table 1. The list of possible v, v'

Index	v	v'	Δiv	Index	v	v'	Δiv	Index	v	v'	Δiv
1	0x3fff8000	0x40007fff	0xff	86	0x3fffd555	0x40002aaa	0x55	171	0x7fffaaaa	0x5555	0xaa
2	0x3fff8101	0x40007efe	0xfd	87	0x3fffd656	0x400029a9	0x53	172	0x7fffabab	0x5454	0xa8
3	0x3fff8202	0x40007dfd	0xfb	88	0x3fffd757	0x400028a8	0x51	173	0x7fffacac	0x5353	0xa6
4	0x3fff8303	0x40007cfc	0xf9	89	0x3fffd858	0x400027a7	0x4f	174	0x7fffadad	0x5252	0xa4
5	0x3fff8404	0x40007bfb	0xf7	90	0x3fffd959	0x400026a6	0x4d	175	0x7fffaeae	0x5151	0xa2
6	0x3fff8505	0x40007afa	0xf5	91	0x3fffda5a	0x400025a5	0x4b	176	0x7fffafaf	0x5050	0xa0
7	0x3fff8606	0x400079f9	0xf3	92	0x3fffdb5b	0x400024a4	0x49	177	0x7fffb0b0	0x4f4f	0x9e
8	0x3fff8707	0x400078f8	0xf1	93	0x3fffdc5c	0x400023a3	0x47	178	0x7fffb1b1	0x4e4e	0x9c
9	0x3fff8808	0x400077f7	0xef	94	0x3fffdd5d	0x400022a2	0x45	179	0x7fffb2b2	0x4d4d	0x9a
10	0x3fff8909	0x400076f6	0xed	95	0x3fffde5e	0x400021a1	0x43	180	0x7fffb3b3	0x4c4c	0x98
11	0x3fff8a0a	0x400075f5	0xeb	96	0x3fffdf5f	0x400020a0	0x41	181	0x7fffb4b4	0x4b4b	0x96
12	0x3fff8b0b	0x400074f4	0xe9	97	0x3fffe060	0x40001f9f	0x3f	182	0x7fffb5b5	0x4a4a	0x94
13	0x3fff8c0c	0x400073f3	0xe7	98	0x3fffe161	0x40001e9e	0x3d	183	0x7fffb6b6	0x4949	0x92
14	0x3fff8d0d	0x400072f2	0xe5	99	0x3fffe262	0x40001d9d	0x3b	184	0x7fffb7b7	0x4848	0x90
15	0x3fff8e0e	0x400071f1	0xe3	100	0x3fffe363	0x40001c9c	0x39	185	0x7fffb8b8	0x4747	0x8e
16	0x3fff8f0f	0x400070f0	0xe1	101	0x3fffe464	0x40001b9b	0x37	186	0x7fffb9b9	0x4646	0x8c
17	0x3fff9010	0x40006fef	0xdf	102	0x3fffe565	0x40001a9a	0x35	187	0x7fffbaba	0x4545	0x8a
18	0x3fff9111	0x40006eee	0xdd	103	0x3fffe666	0x40001999	0x33	188	0x7fffbbbb	0x4444	0x88
19	0x3fff9212	0x40006ded	0xdb	104	0x3fffe767	0x40001898	0x31	189	0x7fffbcbc	0x4343	0x86
20	0x3fff9313	0x40006cec	0xd9	105	0x3fffe868	0x40001797	0x2f	190	0x7fffbdbd	0x4242	0x84
21	0x3fff9414	0x40006beb	0xd7	106	0x3fffe969	0x40001696	0x2d	191	0x7fffbebe	0x4141	0x82
22	0x3fff9515	0x40006aea	0xd5	107	0x3fffea6a	0x40001595	0x2b	192	0x7fffbfbf	0x4040	0x80
23	0x3fff9616	0x400069e9	0xd3	108	0x3fffeb6b	0x40001494	0x29	193	0x7fffc0c0	0x3f3f	0x7e
24	0x3fff9717	0x400068e8	0xd1	109	0x3fffec6c	0x40001393	0x27	194	0x7fffc1c1	0x3e3e	0x7c
25	0x3fff9818	0x400067e7	0xcf	110	0x3fffed6d	0x40001292	0x25	195	0x7fffc2c2	0x3d3d	0x7a
26	0x3fff9919	0x400066e6	0xcd	111	0x3fffee6e	0x40001191	0x23	196	0x7fffc3c3	0x3c3c	0x78
27	0x3fff9a1a	0x400065e5	0xcb	112	0x3fffef6f	0x40001090	0x21	197	0x7fffc4c4	0x3b3b	0x76
28	0x3fff9b1b	0x400064e4	0xc9	113	0x3ffff070	0x40000f8f	0x1f	198	0x7fffc5c5	0x3a3a	0x74
29	0x3fff9c1c	0x400063e3	0xc7	114	0x3ffff171	0x40000e8e	0x1d	199	0x7fffc6c6	0x3939	0x72
30	0x3fff9d1d	0x400062e2	0xc5	115	0x3ffff272	0x40000d8d	0x1b	200	0x7fffc7c7	0x3838	0x70
31	0x3fff9e1e	0x400061e1	0xc3	116	0x3ffff373	0x40000c8c	0x19	201	0x7fffc8c8	0x3737	0x6e
32	0x3fff9f1f	0x400060e0	0xc1	117	0x3ffff474	0x40000b8b	0x17	202	0x7fffc9c9	0x3636	0x6c
33	0x3fffa020	0x40005fdf	0xbf	118	0x3ffff575	0x40000a8a	0x15	203	0x7fffcaca	0x3535	0x6a
34	0x3fffa121	0x40005ede	0xbd	119	0x3ffff676	0x40000989	0x13	204	0x7fffcbcb	0x3434	0x68
35	0x3fffa222	0x40005ddd	0xbb	120	0x3ffff777	0x40000888	0x11	205	0x7fffcccc	0x3333	0x66
36	0x3fffa323	0x40005cdc	0xb9	121	0x3ffff878	0x40000787	0xf	206	0x7fffcdcd	0x3232	0x64
37	0x3fffa424	0x40005bdb	0xb7	122	0x3ffff979	0x40000686	0xd	207	0x7fffcece	0x3131	0x62
38	0x3fffa525	0x40005ada	0xb5	123	0x3ffffa7a	0x40000585	0xb	208	0x7fffcfcf	0x3030	0x60
39	0x3fffa626	0x400059d9	0xb3	124	0x3ffffb7b	0x40000484	0x9	209	0x7fffd0d0	0x2f2f	0x5e
40	0x3fffa727	0x400058d8	0xb1	125	0x3ffffc7c	0x40000383	0x7	210	0x7fffd1d1	0x2e2e	0x5c
41	0x3fffa828	0x400057d7	0xaf	126	0x3ffffd7d	0x40000282	0x5	211	0x7fffd2d2	0x2d2d	0x5a
42	0x3fffa929	0x400056d6	0xad	127	0x3ffffe7e	0x40000181	0x3	212	0x7fffd3d3	0x2c2c	0x58
43	0x3fffaa2a	0x400055d5	0xab	128	0x3fffff7f	0x40000080	0x1	213	0x7fffd4d4	0x2b2b	0x56
44	0x3fffab2b	0x400054d4	0xa9	129	0x7fff8080	0x7f7f	0xfe	214	0x7fffd5d5	0x2a2a	0x54
45	0x3fffac2c	0x400053d3	0xa7	130	0x7fff8181	0x7e7e	0xfc	215	0x7fffd6d6	0x2929	0x52
46	0x3fffad2d	0x400052d2	0xa5	131	0x7fff8282	0x7d7d	0xfa	216	0x7fffd7d7	0x2828	0x50
47	0x3fffae2e	0x400051d1	0xa3	132	0x7fff8383	0x7c7c	0xf8	217	0x7fffd8d8	0x2727	0x4e
48	0x3fffaf2f	0x400050d0	0xa1	133	0x7fff8484	0x7b7b	0xf6	218	0x7fffd9d9	0x2626	0x4c
49	0x3fffb030	0x40004fcf	0x9f	134	0x7fff8585	0x7a7a	0xf4	219	0x7fffdada	0x2525	0x4a
50	0x3fffb131	0x40004ece	0x9d	135	0x7fff8686	0x7979	0xf2	220	0x7fffdbdb	0x2424	0x48
51	0x3fffb232	0x40004dcd	0x9b	136	0x7fff8787	0x7878	0xf0	221	0x7fffdcdc	0x2323	0x46
52	0x3fffb333	0x40004ccc	0x99	137	0x7fff8888	0x7777	0xee	222	0x7fffdddd	0x2222	0x44
53	0x3fffb434	0x40004bcb	0x97	138	0x7fff8989	0x7676	0xec	223	0x7fffdede	0x2121	0x42
54	0x3fffb535	0x40004aca	0x95	139	0x7fff8a8a	0x7575	0xea	224	0x7fffdfdf	0x2020	0x40
55	0x3fffb636	0x400049c9	0x93	140	0x7fff8b8b	0x7474	0xe8	225	0x7fffe0e0	0x1f1f	0x3e
56	0x3fffb737	0x400048c8	0x91	141	0x7fff8c8c	0x7373	0xe6	226	0x7fffe1e1	0x1e1e	0x3c
57	0x3fffb838	0x400047c7	0x8f	142	0x7fff8d8d	0x7272	0xe4	227	0x7fffe2e2	0x1d1d	0x3a
58	0x3fffb939	0x400046c6	0x8d	143	0x7fff8e8e	0x7171	0xe2	228	0x7fffe3e3	0x1c1c	0x38
59	0x3fffba3a	0x400045c5	0x8b	144	0x7fff8f8f	0x7070	0xe0	229	0x7fffe4e4	0x1b1b	0x36
60	0x3fffbb3b	0x400044c4	0x89	145	0x7fff9090	0x6f6f	0xde	230	0x7fffe5e5	0x1a1a	0x34
61	0x3fffbc3c	0x400043c3	0x87	146	0x7fff9191	0x6e6e	0xdc	231	0x7fffe6e6	0x1919	0x32
62	0x3fffbd3d	0x400042c2	0x85	147	0x7fff9292	0x6d6d	0xda	232	0x7fffe7e7	0x1818	0x30
63	0x3fffbe3e	0x400041c1	0x83	148	0x7fff9393	0x6c6c	0xd8	233	0x7fffe8e8	0x1717	0x2e
64	0x3fffbf3f	0x400040c0	0x81	149	0x7fff9494	0x6b6b	0xd6	234	0x7fffe9e9	0x1616	0x2c
65	0x3fffc040	0x40003fbf	0x7f	150	0x7fff9595	0x6a6a	0xd4	235	0x7fffeaea	0x1515	0x2a
66	0x3fffc141	0x40003ebe	0x7d	151	0x7fff9696	0x6969	0xd2	236	0x7fffebeb	0x1414	0x28
67	0x3fffc242	0x40003dbd	0x7b	152	0x7fff9797	0x6868	0xd0	237	0x7fffecec	0x1313	0x26
68	0x3fffc343	0x40003cbc	0x79	153	0x7fff9898	0x6767	0xce	238	0x7fffeded	0x1212	0x24
69	0x3fffc444	0x40003bbb	0x77	154	0x7fff9999	0x6666	0xcc	239	0x7fffeeee	0x1111	0x22
70	0x3fffc545	0x40003aba	0x75	155	0x7fff9a9a	0x6565	0xca	240	0x7fffefef	0x1010	0x20
71	0x3fffc646	0x400039b9	0x73	156	0x7fff9b9b	0x6464	0xc8	241	0x7ffff0f0	0xf0f	0x1e
72	0x3fffc747	0x400038b8	0x71	157	0x7fff9c9c	0x6363	0xc6	242	0x7ffff1f1	0xe0e	0x1c
73	0x3fffc848	0x400037b7	0x6f	158	0x7fff9d9d	0x6262	0xc4	243	0x7ffff2f2	0xd0d	0x1a
74	0x3fffc949	0x400036b6	0x6d	159	0x7fff9e9e	0x6161	0xc2	244	0x7ffff3f3	0xc0c	0x18
75	0x3fffca4a	0x400035b5	0x6b	160	0x7fff9f9f	0x6060	0xc0	245	0x7ffff4f4	0xb0b	0x16
76	0x3fffcb4b	0x400034b4	0x69	161	0x7fffa0a0	0x5f5f	0xbe	246	0x7ffff5f5	0xa0a	0x14
77	0x3fffcc4c	0x400033b3	0x67	162	0x7fffa1a1	0x5e5e	0xbc	247	0x7ffff6f6	0x909	0x12
78	0x3fffcd4d	0x400032b2	0x65	163	0x7fffa2a2	0x5d5d	0xba	248	0x7ffff7f7	0x808	0x10
79	0x3fffce4e	0x400031b1	0x63	164	0x7fffa3a3	0x5c5c	0xb8	249	0x7ffff8f8	0x707	0xe
80	0x3fffcf4f	0x400030b0	0x61	165	0x7fffa4a4	0x5b5b	0xb6	250	0x7ffff9f9	0x606	0xc
81	0x3fffd050	0x40002faf	0x5f	166	0x7fffa5a5	0x5a5a	0xb4	251	0x7ffffafa	0x505	0xa
82	0x3fffd151	0x40002eae	0x5d	167	0x7fffa6a6	0x5959	0xb2	252	0x7ffffbfb	0x404	0x8
83	0x3fffd252	0x40002dad	0x5b	168	0x7fffa7a7	0x5858	0xb0	253	0x7ffffcfc	0x303	0x6
84	0x3fffd353	0x40002cac	0x59	169	0x7fffa8a8	0x5757	0xae	254	0x7ffffdfd	0x202	0x4
85	0x3fffd454	0x40002bab	0x57	170	0x7fffa9a9	0x5656	0xac	255	0x7ffffefe	0x101	0x2

B Generating Identical Keystreams for Δiv_1

Here we describe more details of an algorithm that is used to generate identical keystreams for the IV difference at iv_1:

1. Initialize $iv_0, iv_1, \ldots, iv_{15}$ with 0. Set $iv_{13} = 64$.
2. Denote $(iv_4 + 8iv_{13} + 16iv_{10})$ as sum_1 and guess sum_1 with 1 of the 6376 possible values.
3. Guess $iv_2[1,2]$, and compute v, until the condition $v[1..7] - (v >> 8)[1..7] \leq 1$ is satisfied. If not possible, go to (2) .
4. Guess iv_7 and iv_{11}, and compute u, until $u[24..31] =$ `0xff` is satisfied. We store the intermediate state s_{16}. If not possible, go to (3).
5. Guess iv_{15} and re-compute u, until $u[1..7] = u[9..15]$ and $u[8] = 0$ are satisfied. If not possible, go to (4).
6. Now we compare the current s_{16} with stored s_{16} to capture the change. By properly changing iv_2 and iv_{13}(this is the reason iv_{13} is initialized as 64), we can always change the current s_{16} back to the saved value. Hence, $u[24..31]$ will remain.
7. Determine iv_1 as follows:
   - If $v[8] \neq v[16]$, then if $u[1..16] < v[1..16]$ is satisfied, $iv_1 = 256 + u[1..16] - v[1..16]$ and update v, otherwise, go to (5).
   - If $v[8] = v[16]$, then if $u[1..16] >= v[1..16]$ is satisfied, $iv_1 = u[1..16] - v[1..16]$ and update v, otherwise, go to (5).
8. Guess iv_0, iv_5 and iv_{14}, compute v, until $v[16..31] =$ `0xffff`. If not possible, go to (5).
9. If $(u \oplus v)[1] = 1$, let $iv_2 = iv_2 \oplus 2$. Choose iv_3 properly to ensure $u[16..23] =$ `0xff`. Check if we indeed have $v = u$, then output $iv_0, iv_1, \ldots, iv_{15}$. Otherwise, go to (8).

In this algorithm, we restrict the forms of v and u to those starting with `0x7fff` to reduce the search space.