Open access • Book Chapter • DOI:10.1007/978-3-540-77026-8_6

Related-key attacks on the Py-family of ciphers and an approach to repair the weaknesses — Source link

Gautham Sekar, Souradyuti Paul, Bart Preneel

Institutions: Katholieke Universiteit Leuven

Published on: 09 Dec 2007 - International Conference on Progress in Cryptology

Topics: Stream cipher attack, Key schedule, Differential cryptanalysis, Slide attack and Distinguishing attack

Related-key Attacks on the Py-family of Ciphers and an Approach to Repair the Weaknesses*

Gautham Sekar, Souradyuti Paul and Bart Preneel

Katholieke Universiteit Leuven, Dept. ESAT/COSIC,
Kasteelpark Arenberg 10,
B–3001, Leuven-Heverlee, Belgium.
{gautham.sekar, souradyuti.paul, bart.preneel}@esat.kuleuven.be

Abstract

The stream cipher TPypy has been designed by Biham and Seberry in January 2007 as the strongest member of the Py-family ciphers, after weaknesses in the other members Py, Pypy, Py6 were discovered. One main contribution of the paper is the detection of related-key weaknesses in the Py-family of ciphers including the strongest member TPypy. Under related keys, we show a distinguishing attack on TPypy with data complexity $2^{193.7}$ which is lower than the previous best known attack on the cipher by a factor of 2^{88}. It is shown that the above attack also works on the other members TPy, Pypy and Py. A second contribution of the paper is design and analysis of two fast ciphers RCR-64 and RCR-32 which are derived from the TPy and the TPypy respectively. The performances of the RCR-64 and the RCR-32 are 2.7 cycles/byte and 4.45 cycles/byte on Pentium III (note that the speeds of the ciphers Py, Pypy and RC4 are 2.8, 4.58 and 7.3 cycles/byte). Based on our security analysis, we conjecture that no attacks lower than brute force are possible on the RCR ciphers.

1 Introduction

Timeline: the Py-family of Ciphers

- **April 2005, Design.** The ciphers Py and Py6, designed by Biham and Seberry, were submitted to the ECRYPT project for analysis and evaluation in the category of software based stream ciphers [5]. The impressive speed of the cipher Py in software (about 2.5 times faster than the RC4) made it one of the fastest and most attractive contestants.
- **March 2006, Attack (at FSE 2006).** Paul, Preneel and Sekar reported distinguishing attacks with $2^{89.2}$ data and comparable time against the cipher Py [20]. Crowley [8] later reduced the complexity to 2^{72} by employing a Hidden Markov Model.
- **March 2006, Design (at the Rump session of FSE 2006).** A new cipher, namely Pypy, was proposed by the designers to rule out the aforementioned distinguishing attacks on Py [6].
- **May 2006, Attack (presented at Asiacrypt 2006).** Distinguishing attacks were reported against Py6 with $2^{68.6}$ data and comparable time by Paul and Preneel [21].
- **October 2006, Attack (presented at Eurocrypt 2007).** Wu and Preneel showed key recovery attacks against the ciphers Py, Pypy, Py6 with chosen IVs. This attack was subsequently improved by Isobe *et al.* [12].

*The first author is supported by an IWT SoBeNeT project. The second author is funded by the IBBT (Interdisciplinary Institute for BroadBand Technology), a research institute founded by the Flemish Government in 2004. This is a revised version of the paper published in the proceedings of Indocrypt 2007.

- **January 2007, Design.** Three new ciphers TPypy, TPy, TPy6 were proposed by the designers [4]; the ciphers can very well be viewed as the strengthened versions of the previous ciphers Py, Pypy and Py6 where the above attacks should not apply. So far there exist no published attacks on TPypy, TPy and TPy6.

- **February 2007, Attack.** Sekar, Paul and Preneel published distinguishing attacks on Py, Pypy, TPy and TPypy with data complexities 2^{281} each [25].

- **June 2007, Attack (to be presented at ISC 2007).** Sekar, Paul and Preneel showed new weaknesses in the stream ciphers TPy and Py [23]. Exploiting these weaknesses distinguishing attacks on the ciphers are constructed where the best distinguisher requires $2^{268.6}$ data and comparable time.

- **July 2007, Attack and Design (presented at WEWoRC 2007).** Sekar, Paul and Preneel mounted distinguishing attacks on TPy6 and Py6 with $2^{224.6}$ data and comparable time each [24]. Moreover, they have modified TPy6 to design two new ciphers TPy6–A and TPy6–B which were claimed to be free from all attacks excluding brute force ones.

Contribution of the paper. The list that orders the Py-family of ciphers in terms of increasing security is: Py6→Py→ Pypy → TPy6 → TPy → TPypy (the strongest). The ciphers are normally used with 32-byte keys and 16-byte initial values (or IV). However, the key size may vary from 1 to 256 bytes and the IV from 1 to 64 bytes. The ciphers were claimed by the designers to be free from related-key and distinguishing attacks [4, 5, 6].
(i) *Related-key Weaknesses.* One major contribution of the paper is the discovery of related-key attacks due to weaknesses in the key scheduling algorithms of the Py-family of ciphers. The main idea behind a related-key attack is that, the attacker, who chooses a relation f between a pair of keys key_1 and key_2 (e.g., $key_1 = f(key_2)$) rather than the actual values of the keys, is able to extract secret information from a cryptosystem using the relation f [3, 14]. Related-key weakness is a cause for concern in a protocol where key-integrity is not guaranteed or when the keys are generated manually rather than from a pseudorandom number generator [13]. Related-key weaknesses are not new in the literature. The usefulness of such type of attacks was first outlined by Knudsen in [15, 16]; since then a good deal of research has been spent on related-key weaknesses on block ciphers [3, 13, 14, 17]. The related-key weaknesses of a block cipher can be translated into attacking hash functions based on that particular block cipher and vice versa [10, 11, 19, 22, 26, 27, 29, 31]. Theoretical treatments of related-key attacks were done in [2] and [18].

On the other hand, discovery of related-key weaknesses of stream ciphers is not very common in the literature, mainly due to the heavy operations executed in one-time key-scheduling algorithms compared to the operations performed in iterative block ciphers. However, there is an example where related-key weaknesses of the stream cipher RC4 are used to break the WEP protocol with practical complexity [9]. Furthermore, there is a growing tendency by the designers nowadays to build hash functions from stream ciphers [7] instead of building them from block ciphers. In such attempts, related-key weaknesses of stream ciphers need to be addressed carefully.

In the paper, we show that, when used with the identical IVs of 16 bytes each, if two long keys key_1 and key_2 of 256 bytes each, are related in the following manner,

1. $key_1[16] \oplus key_2[16] = 1$,

2. $key_1[17] \neq key_2[17]$ and

3. $key_1[i] = key_2[i] \ \forall i \notin \{16, 17\}$

then the above relation, exploiting the weaknesses of the key setup algorithms of Py-family of ciphers (i.e., TPypy, TPy, Pypy, Py), propagates through the IV setup algorithms and finally induces biases in the outputs at the 1st and the 3rd rounds. Such related key pairs are used to build a distinguisher for each of the aforementioned ciphers with $2^{193.7}$ output words and comparable time (note that, in total, there are 2^{2048} such pairs, while our distinguisher needs any $2^{193.7}$ randomly chosen pairs of keys). This result constitutes the best attack on the strongest member of the Py-family of ciphers TPypy; they are also shown to be effective on the other members TPy, Pypy and Py (see Table 1). These related-key attacks work with any IV-size ranging from 16 to 64 bytes. However, the attack complexities increase with shorter keys. Note that the usage of long keys in the Py-family of ciphers makes it very attractive to be used as fast hash functions (e.g., by replacing of the key with the message). In such cases, these related-key weaknesses can turn out to be serious impediments.

Table 1: Attacks on the Py-family of stream ciphers ('X' denotes that the attack does not work)

Attack	**Py6**	**Py**	**Pypy**	**TPy6**	**TPy**	**TPypy**
Crowley [8]	X	2^{72}	X	X	2^{72}	X
Isobe *et al.* [12]	X	2^{24}	2^{24}	X	X	X
Paul *et al.* [20]	X	$2^{89.2}$	X	X	$2^{89.2}$	X
Paul-Preneel [21]	$2^{68.6}$	X	X	$2^{68.6}$	X	X
Sekar *et al.* [23]	X	$2^{268.6}$	X	X	$2^{268.6}$	X
Sekar *et al.* [24]	$2^{224.6}$	X	X	$2^{224.6}$	X	X
Sekar *et al.*[25]	X	2^{281}	2^{281}	X	2^{281}	2^{281}
Wu-Preneel [32]	X	2^{24}	2^{24}	X	X	X
Related key (this paper)	X	$2^{193.7}$	$2^{193.7}$	X	$2^{193.7}$	$2^{193.7}$

(ii) *The Ciphers RCR-32 and RCR-64.* Finally, we make simple modifications to the ciphers TPypy and TPy to build two new ciphers RCR-32 and RCR-64 respectively. In the modified designs, the key scheduling algorithms of RCR-32 and RCR-64 are identical with those of the TPypy and the TPy. The changes are made *only* to the round functions where *variable rotations* are replaced with *constant rotations*. Our extensive analyses show that the modifications not only free the Py-family ciphers from *all* the existing attacks, it also improves on the performance of the ciphers without exposing them to new weaknesses (see Sect. 5 for an elaborate security analysis). As a result, the cipher RCR-64 goes on to become one of the *the fastest* stream ciphers published in the literature (approximately 2.7 cycles per byte on Pentium III). The names are chosen to reflect the functionalities involved in the ciphers. For example, RCR-64 denotes *Rolling, Constant Rotation and 64 bits output/round.*

2 Description of the Stream Ciphers TPypy, TPy, Pypy and Py

Each of the Py-family of ciphers is composed of three parts: (1) a key setup algorithm, (2) an IV setup algorithm and (3) a round function or pseudorandom bit generation algorithm (PRBG). The first two parts are used for the initial one-time mixing of the secret key and the IV. These parts generate a pseudorandom internal state composed of (1) a permutation P of 256 elements, (2) a 32-bit array Y of 260 elements and (3) a 32-bit variable s. The key/IV setup uses two intermediate variables: (1) a fixed permutation of 256 elements denoted by *internal_permutation* and (2) a variable EIV whose size is equal to that of the IV. The round function, which is executed iteratively, is used to update the internal state (i.e., P, Y and s) and to generate pseudorandom output bits. The key setup algorithms of the TPypy, the TPy, the Pypy and the Py are identical. Notation for different parts of the four ciphers is provided in Table 2.

Table 2: Description of the ciphers TPypy, TPy, Pypy and Py

	TPypy	TPy	Pypy	Py
Key Setup	KS	KS	KS	KS
IV Setup	IVS_1	IVS_1	IVS_2	IVS_2
Round Function	RF_1	RF_2	RF_1	RF_2

Due to space constraints, the KS, the IVS_1, the IVS_2, the RF_1 and the RF_2, as mentioned in Table 2, are described in Appendix A. The details of the algorithms can also be found in [4, 5, 6].

3 Notation and Convention

The notation and the convention followed in the paper are described below.

- The pseudorandom bit generation algorithm of a stream cipher is denoted by PRBG.
- The outputs generated when key_1 and key_2 are used are denoted by O and Z respectively.
- $O^a_{(b)}$ (or $Z^a_{(b)}$) denotes the bth bit ($b = 0$ is the least significant bit or lsb) of the second output word generated at round a when key_1 (or key_2) is used. We do not use the first output word anywhere in our analysis.
- P^a_1, Y^{a+1}_1 and s^a_1 are the inputs to the PRBG at round a when key_1 is used. It is easy to see that when this convention is followed the O^a takes a simple form: $O^a = (s \oplus Y^a[-1]) + Y^a[P^a[208]]$. The same applies to key_2.
- $Y^a_1[b]$, $P^a_1[b]$ denote the bth elements of array Y^a_1 and P^a_1 respectively, when key_1 is used.
- $Y^a_1[b]_i$, $P^a_1[b]_i$ denote the ith bit of $Y^a_1[b]$, $P^a_1[b]$ respectively.
- The operators '+' and '−' denote *addition modulo* 2^{32} and *subtraction modulo* 2^{32} respectively, except when used with expressions which relate two elements of array P. In this case they denote *addition and subtraction over* $\mathbb{Z}$.
- The symbol '$\oplus$' denotes bitwise *exclusive-or*, $\cap$ denotes set intersection and $\cup$ denotes set union.

4 Related-key Weaknesses in the Py-family of Ciphers

We first choose two keys, key_1 and key_2 (each key is 256 bytes long), such that,
C1. $key_1[16] \oplus key_2[16] = 1$ (without loss of generality, assume lsb of $key_1[16]$ is 1),
C2. $key_1[17] \neq key_2[17]$ and **C3.** $key_1[i] = key_2[i]\ \forall i \notin \{16, 17\}$.

Now we observe that the above relation between the keys can be traced through various parts of the Py-family of ciphers.

4.1 Propagation of the Weaknesses through the Key Setup Algorithm

For key_1 and key_2, the values of the variable s through Algorithm A are tabulated in Table 3. The Algorithm A is a part of the key setup algorithm KS (described in Algorithm 2 in Appendix A).

```
Algorithm A
 for(j=0; j<keysizeb; j++)
  {
    s = s + key[j];
    s0 = internal_permutation[s&0xFF];
    s = ROTL32(s, 8) ^ (u32)s0;
  }
```

Table 3: The variable s after rounds 15, 16 and 17 of Algorithm A

End of round	s (using key_1)	s (using key_2)
15	$s^A_{1,15}$	$s^A_{2,15} = s^A_{1,15}$
16	$s^A_{1,16}$	$s^A_{2,16} = s^A_{1,16} - \delta_1$ (say)
17	$s^A_{1,17}$	$s^A_{2,17} = s^A_{1,17}$ if $key_2[17] = key_1[17] + \delta_1$

If x is a 32-bit variable, let $B(x)$ denote the least significant byte of x. In Table 3,

$$\begin{aligned} \delta_1 &= s^A_{1,16} - s^A_{2,16} && (1)\\ &= ROTL32((s^A_{1,15} + key_1[16]), 8) \oplus ip[B(s^A_{1,15} + key_1[16])] && (2)\\ &- ROTL32((s^A_{2,15} + key_2[16]), 8) \oplus ip[B(s^A_{2,15} + key_2[16])], && (3) \end{aligned}$$

where ip denotes *internal_permutation*.

Now, if $key_2[17] = key_1[17] + \delta_1$ (call this the event D_1), it is observed from Algorithm A that the following equation is satisfied:

$$s^A_{1,17} = s^A_{2,17}.$$

For event D_1 to occur, δ_1 should be an 8-bit integer. Running simulation, it is determined that

$$Pr[|\delta_1| = 8] \approx \frac{1}{2}.$$

Hence,

$$Pr[D_1] \approx 2^{-9}. \quad (4)$$

If $s^A_{1,17} = s^A_{2,17}$, then in the subsequent rounds of Algorithm A, the s^A_1 and s^A_2 remain the same, that is, $s^A_{1,k} = s^A_{2,k}$, where $k = 18, 19, ..., 255$.

Given that the D_1 occurs, that is, $s^A_1 = s^A_2$ at the end of Algorithm A, or $s^A_{1,255} = s^A_{2,255}$, we now trace the values of s through Algorithm B which forms another part of the key setup. Table 4 compares the values of s after rounds 15, 16 and 17 of Algorithm B when key_1 and key_2 are used.
In Table 4,

$$\begin{aligned} \delta_2 &= s^B_{1,16} - s^B_{2,16}\\ &= ROTL32((s^B_{1,15} + key_1[16]), 8) \oplus ip[B(s^B_{1,15} + key_1[16])]\\ &- ROTL32((s^B_{2,15} + key_2[16]), 8) \oplus ip[B(s^B_{2,15} + key_2[16])]. && (5) \end{aligned}$$

```
Algorithm B
 for(j=0; j<keysizeb; j++)
  {
    s = s + key[j];
    s0 = internal_permutation[s&0xFF];
    s ^= ROTL32(s, 8) + (u32)s0;
  }
```

Table 4: s after rounds 15, 16 and 17 of Algorithm B given event D_1 occurs

End of round	s (using key_1)	s (using key_2)
15	$s^B_{1,15}$	$s^B_{2,15} = s^B_{1,15}$
16	$s^B_{1,16}$	$s^B_{2,16} = s^B_{1,16} - \delta_2$ (say)
17	$s^B_{1,17}$	$s^B_{2,17} = s^B_{1,17}$ if $key_2[17] = key_1[17] + \delta_2$

Now, given event D_1 occurs, i.e., $s^A_1 = s^A_2$ at the end of Algorithm A, if $\delta_2 = \delta_1$ (call this the event D_2), we will have $key_2[17] = key_1[17] + \delta_2$ and hence from Algorithm B, the following equation is satisfied:

$$s^B_{1,17} \quad = \quad s^B_{2,17}.$$

For event D_2 to occur, δ_2 should be an 8-bit integer. Running simulation, it is determined that

$$Pr[|\delta_2| = 8] \approx \frac{1}{2^{2.4}}.$$

Hence,

$$Pr[D_2|D_1] \approx 2^{-10.4} \Rightarrow Pr[D_2 \cap D_1] \approx Pr[D_1] \cdot 2^{-10.4} \approx 2^{-19.4}. \quad (6)$$

If $s^B_{1,17} = s^B_{2,17}$, then in the subsequent rounds of Algorithm B, the s^B_1 and s^B_2 remain the same, that is, $s^B_{1,k} = s^B_{2,k}$, where $k = 18, 19, ..., 255$.

Given that the $D_2 \cap D_1$ occurs, that is, $s^B_1 = s^B_2$ at the end of Algorithm B , or $s^B_{1,255} = s^B_{2,255}$, the values of s and Y are traced through Algorithm C which forms the final part of the key setup. Table 6 in Appendix C compares the values of s and Y after rounds 15, 16 and 17 of Algorithm C when key_1 and key_2 are used. Since Algorithm C and Table 6 have striking similarities with Algorithm A and Table 3, they are described in Appendix C and we provide only the results of our analysis. Now, given that the event $D_2 \cap D_1$ occurs, i.e., $s^B_1 = s^B_2$ at the end of Algorithm B, if $\delta_3 = \delta_1$ (call this the event D_3), we will have $key_2[17] = key_1[17] + \delta_3$ and hence from Algorithm C, the following equation is satisfied:

$$s^C_{1,17} \quad = \quad s^C_{2,17}.$$

For event D_3 to occur, δ_2 should be an 8-bit integer. Running simulation, it is determined that

$$Pr[|\delta_3| = 8] \approx \frac{1}{2}.$$

Hence,

$$Pr[D_3|D_2 \cap D_1] \approx 2^{-9} \Rightarrow Pr[D_3 \cap D_2 \cap D_1] \approx Pr[D_2 \cap D_1] \cdot 2^{-9} \approx 2^{-28.4}. \quad (7)$$

If $s^C_{1,17} = s^C_{2,17}$, then in the subsequent rounds of Algorithm C, the s^C_1 and s^C_2 remain the same, that is, $s^C_{1,k} = s^C_{2,k}$, where $k = 18, 19, ..., 255$ and $Y_1[j] = Y_2[j]$, where $j \neq 13$.

4.2 Propagation of the Weaknesses through the IV Setup

```
Algorithm D
 for(i=0; i<ivsizeb; i++)
  {
   s = s + iv[i] + Y(YMININD+i);
   u8 s0 = P(s&0xFF);
   EIV(i) = s0;
   s = ROTL32(s, 8) ^ (u32)s0;
 }
```

Given that the $D_3 \cap D_2 \cap D_1$ occurs, i.e., $s_1^C = s_2^C$ at the end of Algorithm C, or $s_{1,255}^C = s_{2,255}^C$, and $Y_1[i] = Y_2[i]$ $(i \neq 13)$, we now trace the variables s, Y, P and EIV through the first part of the IV setup. We now consider Algorithm D which is a part of the IV setup. It is to be noted that s, Y (obtained after the key setup) and the iv are the basic elements used in the IV setup to define the P and the EIV and to update the s and the Y. We now model our attack in such a way that the same IV is used with both the keys. Prior to the execution of Algorithm D, the only elements of array Y which are used in the first part of the IV setup are $Y[0]$, $Y[1]$, $Y[YMININD]$ and $Y[YMAXIND]$. Since $Y[13]$ is not used, it follows that P_1 (that is, P when key_1 is used) and P_2 (that is, P when key_2 is used) are identical.

In Algorithm D as well, $Y[13]$ is not used to update the s or define the EIV when the IV is of the recommended size of 16 bytes. For longer IVs, we can induce the first difference in the keys (that is, where the least significant bits alone differ) according to the size of the IV. An example is provided in Appendix D. It is to be noted that, if the IV-size is N bytes, the first difference in the keys should be induced nowhere: neither (1) in the first $N-1$ bytes (i.e., key bytes 0 to $N-1$), nor (2) in the last $N-3$ bytes (i.e., key bytes $260-N$ to 256). Otherwise, it is immaterial as to where the first difference is set

```
Algorithm E
 for(i=0; i<ivsizeb; i++)
  {
    s = s + iv[i] + Y(YMAXIND-i);
    /*s = s + EIV((i+ivsizeb-1)mod ivsizeb) + Y(YMAXIND-i); for IVS1.*/
    u8 s0 = P(s&0xFF);
    EIV(i) += s0;
    s = ROTL32(s, 8) ^ (u32)s0;
  }
```

(i.e., anywhere from byte N to $259-N$) – in all the cases, bias induced will be approximately identical (this is established from a large number of experiments).

We now consider Algorithm E. Again, $Y[13]$ is not used to update the s or the EIV (for both IVS_1 and IVS_2). Hence, at the end of Algorithm E, we have $s_1 = s_2$, $EIV_1 = EIV_2$, $P_1 = P_2$ and $Y_1[i] = Y_2[i]$ (where $i \neq 13$). With this result, we now proceed to the second part of the IV setup.

In the second part of the IV setup (that is, for IVS_2), when $i = 16$ ($i = 17$ for IVS_1), the s generated using key_1 and key_2 are different due to the difference in $Y[13]$. This causes the EIVs to be different in the following round and hence $P_1 \neq P_2$. In the subsequent rounds, the mixing becomes more random with the result that at the end of 260 rounds, we have $Y_1[j] = Y_2[j]$ where $j \in \{-3, ..., 12\}$. This result holds only if $x0 \neq 13$ when $i = 0, ..., 15$. The probability that this occurs is $(\frac{255}{256})^{j+4} \approx 1$ when $j \in \{-3, ..., 12\}$. With this result, we now analyze the keystream generation algorithm.

```
IV setup part-2
  for(i=0; i<260; i++)
  {
    u32 x0 = EIV(0) = EIV(0) ^ (s&0xFF);
    rotate(EIV);
    swap(P(0), P(x0));
    rotate(P);
    Y(YMININD)=s=(s ^ Y(YMININD))+Y(x0);
    /*s=ROTL32(s,8)+Y(YMAXIND);
    Y(YMININD)+=s^Y(x0); for IVS1.*/
    rotate(Y);
  }
```

4.3 Propagation of the Weaknesses through the Round Function

Here, we consider only the round function RF_1 of Algorithm 5 (see Appendix A). The formulas for the lsb of the outputs generated at rounds 1 and 3 when key_1 (the output words are denoted by O) and key_2 (the output words are denoted by Z) are used are given below.

$$O^1_{(0)} = s^1_{1(0)} \oplus Y^1_1[-1]_0 \oplus Y^1_1[P^1_1[208]]_0, \quad (8)$$
$$O^3_{(0)} = s^3_{1(0)} \oplus Y^3_1[-1]_0 \oplus Y^3_1[P^3_1[208]]_0, \quad (9)$$
$$Z^1_{(0)} = s^1_{2(0)} \oplus Y^1_2[-1]_0 \oplus Y^1_2[P^1_2[208]]_0, \quad (10)$$
$$Z^3_{(0)} = s^3_{2(0)} \oplus Y^3_2[-1]_0 \oplus Y^3_2[P^3_2[208]]_0. \quad (11)$$

Let C_1, C_2, C_3 and C_4 denote $Y^1_1[P^1_1[208]]_0$, $Y^3_1[P^3_1[208]]_0$, $Y^1_2[P^1_2[208]]_0$ and $Y^3_2[P^3_2[208]]_0$ respectively. Each row in Table 5 gives the conditions on the elements of P_1 and P_2 which when simultaneously satisfied gives $C_1 \oplus C_2 \oplus C_3 \oplus C_4 = 0$. The corresponding probabilities are also given. From Table 5, it follows

Table 5: When G_j $(1 \le j \le 4)$ occurs, $C_1 \oplus C_2 \oplus C_3 \oplus C_4 = 0$

Event	Conditions	Probability	Result
G_1	$P^1_1[208] = P^3_1[208]+2$, $P^1_2[208] = P^3_2[208]+2$	2^{-16}	$C_1 = C_2$,$C_3 = C_4$
G_2	$P^1_1[208] = P^1_2[208]$, $P^1_1[208], P^1_2[208] \le 12$, $P^3_1[208] = P^3_2[208]$, $P^3_1[208], P^3_2[208] \le 12$	$2^{-24.6}$	$C_1 = C_3$,$C_2 = C_4$
G_3	$P^1_1[208] = P^3_2[208] + 2$, $2 \le P^1_1[208] \le 12$, $P^3_2[208] \le 10$, $P^1_2[208] = P^3_1[208] + 2$, $2 \le P^1_2[208] \le 12$, $P^3_1[208] \le 10$	$2^{-25.4}$	$C_1 = C_4$,$C_2 = C_3$
G_4	$G_2 \cap G_1$	Negligible ($<< 2^{-25}$)	$C_1 = C_2 = C_3 = C_4$

that events G_2, G_3 and G_4 can be ignored when compared to G_1. We now state the following theorem.

Theorem 1 $s^1_1 = s^3_1$ *when the following conditions are simultaneously satisfied.*

1. $P^2_1[116] \equiv -18 \bmod 32$ *(event E_1),*
2. $P^3_1[116] \equiv -18 \bmod 32$ *(event E_2),*
3. $P^2_1[72] = P^3_1[239] + 1$ *(event E_3),*

4. $P_1^2[239] = P_1^3[72] + 1$ *(event E_4).*

Proof. The formulas for s_1^2 and s_1^3 are given below (see Algorithm 5):

$$\begin{aligned} s_1^2 &= ROTL32(s_1^1 + Y_1^2[P_1^2[72]] - Y_1^2[P_1^2[239]], P_1^2[116] + 18 \bmod 32), \quad (12)\\ s_1^3 &= ROTL32(s_1^2 + Y_1^3[P_1^3[72]] - Y_1^3[P_1^3[239]], P_1^3[116] + 18 \bmod 32). \quad (13)\end{aligned}$$

Condition 1 (i.e., $P_1^2[116] \equiv -18 \bmod 32$) reduces (12) to

$$s_1^2 = s_1^1 + Y_1^2[P_1^2[72]] - Y_1^2[P_1^2[239]].$$

Therefore, (13) becomes

$$s_1^3 = ROTL32(s_1^1 + \sum_{i=2}^{3}(Y_1^i[P_1^i[72]] - Y_1^i[P_1^i[239]]), P_1^3[116] + 18 \bmod 32). \quad (14)$$

Now, condition 3 (i.e., $P_1^2[72] = P_1^3[239] + 1$) and condition 4 ($P_1^2[239] = P_1^3[72] + 1$) together imply $\sum_{i=2}^{3}(Y_1^i[P_1^i[72]] - Y_1^i[P_1^i[239]]) = 0$ and hence reduce (14) to

$$s_1^3 = ROTL32(s_1^1, P_1^3[116] + 18 \bmod 32). \quad (15)$$

Now, when event E_2 (that is, $P_1^3[116] \equiv -18 \bmod 32$) occurs, (15) becomes

$$s_1^3 = ROTL32(s_1^1, 0) = s_1^1. \quad (16)$$

This completes the proof. □

Now, $s_1^1 = s_1^3 \Rightarrow s_{1(0)}^1 = s_{1(0)}^3$ and $Pr[E_1] \approx Pr[E_2] \approx 2^{-5}$ and $Pr[E_3] \approx Pr[E_4] \approx 2^{-8}$. The four events E_1, E_2, E_3 and E_4 are assumed to be independent to facilitate calculation of bias. The actual value without independence assumption is in fact more, making the attack marginally stronger. Hence, $Pr[E_1 \cap E_2 \cap E_3 \cap E_4] = 2^{-26}$. Similarly, we have $s_2^1 = s_2^3$ when the following conditions are simultaneously satisfied.

1. $P_2^2[116] \equiv -18 \bmod 32$ (event E_5), **2.** $P_2^3[116] \equiv -18 \bmod 32$ (event E_6),
3. $P_2^2[72] = P_2^3[239] + 1$ (event E_7), **4.** $P_2^2[239] = P_2^3[72] + 1$ (event E_8).

Again, $s_2^1 = s_2^3 \Rightarrow s_{2(0)}^1 = s_{2(0)}^3$ and

$$Pr[\cap_{i=1}^{8} E_i] = \frac{1}{2^{52}}. \quad (17)$$

From the analysis in Sect. 4.1 and 4.2, when $D_3 \cap D_2 \cap D_1$ occurs, $Y_1^1[j] = Y_2^1[j]$ where $j \in \{-3, ..., 12\}$. $Y_1^1[i] = Y_2^1[i] \Rightarrow Y_1^1[-1]_0 = Y_2^1[-1]_0$ and $Y_1^3[-1]_0 = Y_1^1[1]_0 = Y_2^1[1]_0 = Y_2^3[-1]_0$. Therefore, from equations (8), (9), (10) and (11), we observe that

$$O_{(0)}^1 \oplus O_{(0)}^3 \oplus Z_{(0)}^1 \oplus Z_{(0)}^3 = 0 \quad (18)$$

holds when the following events simultaneously occur.

1. $D_3 \cap D_2 \cap D_1$, **2.** $\cap_{i=1}^{8} E_i$ and **3.** G_1.

In the following section, we calculate the probability that (18) is satisfied.

4.4 The Distinguisher

Let L denote the event $(\cap_{i=1}^{8} E_i) \cap (D_3 \cap D_2 \cap D_1) \cap (G_1)$. From (7), (17) and Table 5, we get: $Pr[L] = 2^{-52} \cdot 2^{-28.4} \cdot 2^{-16} = 2^{-96.4}$. Assuming randomness of the outputs when event L does not occur (concluded from a large number of experiments), we have:

$$Pr[O^1_{(0)} \oplus O^3_{(0)} \oplus Z^1_{(0)} \oplus Z^3_{(0)} = 0] \quad = \quad \frac{1}{2}(1 + \frac{1}{2^{96.4}}). \tag{19}$$

To compute the number of samples required to establish an optimal distinguisher with advantage greater than 0.5, we use the following equation:

$$n = 0.4624 \cdot \frac{1}{p^2} \tag{20}$$

from [1, 20]. Here, $p = 2^{-97.4}$. Therefore, the number of samples is $2^{193.7}$.

4.5 Attacks with Shorter Keys

The related-key attacks described in the previous sections can be applied with shorter keys also. However, the data complexity of the distinguisher increases exponentially as key size decreases. For example, when the key size is 128 bytes, the distinguisher works with $2^{229.7}$ data and comparable time. For 64-byte key size, the data complexity of the distinguisher is $2^{247.7}$.

5 New Stream Ciphers: RCR-32 and RCR-64

As mentioned in Sect. 1, in the last couple of years, the Py-family of ciphers have come under several cryptanalytic attacks. In spite of the weaknesses, the ciphers retain some attractive features such as modification of the internal states with clever use of rolling arrays and fast mixing of several arithmetic operations. This motivates us to explore the possibility of designing new ciphers that retain all the good properties of the Py-family and yet are secure against all the existing and new attacks.

In this section, we propose two new ciphers, RCR-32 (*R*olling, *C*onstant *R*otation, *32*-bit output per round) and RCR-64 derived from TPypy and Tpy, which are shown to be secure against all the existing attacks on the TPypy and TPy. The speeds of execution of the RCR-64 and the RCR-32 in software are 2.7 cycles and 4.45 cycles per byte which are better than the performances of the TPy (2.8 cycles/byte) and the TPypy (4.58 cycles/byte) respectively.

The key/IV setup algorithms of the RCR-64 and the RCR-32 are identical with those of the TPy and the TPypy. The PRBGs of the RCR-64 and the RCR-32 are also very similar to those of the TPy and the TPypy. The only changes in the PRBGs are that: the *variable rotation* of the quantity s is replaced by a *constant rotation* (c) of 19. Single round of RCR-32 and RCR-64 are shown in Algorithm 1.

5.1 Security Analysis

In this section we justify how the new ciphers RCR-32 and RCR-64 should be able to resist several common attacks against array-based stream ciphers. In the following analysis the symbols $x_{r(i)}$ and A^c denote the ith bit of the variable x at round r and the *bitwise* complement of A. This notation is made slightly different from the one used throughout the paper to accommodate the complement operation.

(i) Resistance to Distinguishing Attacks: The RCR-32 and the RCR-64 are the modified versions of the TPy and the TPypy. The following distinguishing attacks are applicable to the TPy and TPypy. We now show why those attacks do not apply to the RCR ciphers.

1. *Paul-Preneel-Sekar attack [20]:* This attack applies to the TPy. Condition 1 under Theorem 1 in [20], that is, $P_2[116] \equiv -18 \bmod 32$, is impossible when $c = 19$ (in which case we have $P_2[116] \equiv$

Algorithm 1 Round functions of RCR-32 and RCR-64

Require: $Y[-3, ..., 256]$, $P[0, ..., 255]$, a 32-bit variable s

Ensure: 64-bit random output (for RCR-64) or 32-bit random output (for RCR-32)

```
    /*Update and rotate P*/
 1: swap (P[0], P[Y[185]&255]);
 2: rotate (P);
    /* Update s*/
 3: s+ = Y[P[72]] − Y[P[239]];
 4: s = ROTL32(s, 19); /*Tweak - the variable s undergoes a constant, non-zero rotation (c = 19).*/
    /* Output 4 or 8 bytes (the least significant byte first)*/
 5: output ((ROTL32(s, 25) ⊕ Y[256]) + Y[P[26]]);/* This step is skipped for RCR-32.*/
 6: output ((        s        ⊕Y[−1]) + Y[P[208]]);
    /* Update and rotate Y*/
 7: Y[−3] = (ROTL32(s, 14) ⊕ Y[−3]) + Y[P[153]];
 8: rotate(Y);
```

1 mod 32). Note that when $c = 0$, $P_2[116] \equiv -18 \bmod 32$ is satisfied. Therefore, $c = 0$ is not a safe choice.

2. *Sekar-Paul-Preneel attack [25]:* This attack applies to both TPy and TPypy. Again, condition 1 under Theorem 1, that is, $P_2[116] \equiv -18 \bmod 32$, is violated when $c = 19$ (in which case we have $P_2[116] \equiv 1 \bmod 32$). Note that condition 1 is common to all the 144 sets of conditions (see [25]) and hence its violation nullifies the attack.

3. *Sekar et al. attack [23]:* This attack applies to the TPy. Condition 1 under Theorem 1, that is, $P_1[116] \equiv -18 \bmod 32$, is not satisfied when $c = 19$ (in which case we have $P_1[116] \equiv 1 \bmod 32$). This leads to another important observation: none of the large number of weaknesses detected in TPy in[23], apply to the RCR-32 or RCR-64. Here again, when $c = 0$, $P_1[116] \equiv -18 \bmod 32$ is satisfied. Therefore, $c = 0$ is not a safe choice.

In Appendix B, we elaborate more on the usefulness of selection of *constant rotation* to eliminate any distinguishing attacks on RCR ciphers. Here, it may appear that a constant rotation results in cyclic repetition of the variable s every 32 rounds. However, in each round, a 32-bit random is added to s (see line 3 of Algorithm 1) and hence such a cycle (or any short cycle) can only occur with negligible probability.

(ii) Resistance to Related-key attacks of this paper: The related-key attacks presented in Sect. 4 are similar to the attacks by Paul *et al.* described in [20]. Here, the event E_5 (i.e., $P_2^2[116] \equiv -18 \bmod 32$), described in Sect. 4.3, does not occur if $c = 19$ (in which case we have $P_2^2[116] \equiv 1 \bmod 32$). Note that when $c = 0$, $P_2^2[116] \equiv -18 \bmod 32$ is satisfied; therefore, the choice of c to be zero is not very safe. Apparently, it may seem that the security of the RCR-64 and the RCR-32 are threatened by the *unchanged* key setup algorithms. However, the weaknesses in the key setup cannot be translated into any meaningful attack on any of our designs. This is because of the heavy mixing that takes place in the second part of the IV setup. As a result, we expect that the variable s, generated at the end of the IV setup, is uniformly distributed at random. Therefore, the outputs generated in the keystream generation algorithm are not expected to be correlated unless we have the s to be rotated by a variable term (note that this variable term is set to different values at different rounds to construct the attacks). In the round functions of the ciphers RCR-64 and RCR-32, the s is rotated by a constant term and hence the ciphers are expected to be free from any correlations between the outputs.

(iii) Resistance to Differential attacks: Wu and Preneel found weaknesses in the IV setups of the Py and the Pypy [32]. Exploiting these weaknesses, some key-dependent information has been recovered.

The ciphers TPy and TPypy were specifically designed to rule out these weaknesses. Since the IV setup algorithms of the RCR-64 and the RCR-32 are identical with those of TPy and TPypy, these attacks are no longer applicable in new ciphers.

(iv) Resistance to Algebraic attacks and Guess-and-Determine Attacks: RCR-32 and RCR-64 are array-based stream ciphers. The sizes of the internal states of RCR-32 and RCR-64 are 10,400 bits each, which is very large. Hence, it appears infeasible to mount algebraic attacks that are otherwise common against LFSR-based stream ciphers which have low footprints. From our experiments, we expect that the RCR-32 and RCR-64 are also secure against guess-and-determine attacks.

6 Future Work and Conclusion

In this paper, for the first time, we detect weaknesses in the key scheduling algorithms of several members of the Py-family. Precisely, we build distinguishing attacks with data complexities 2^{193} each. Furthermore, we modify the ciphers TPypy and TPy to generate two fast ciphers, namely RCR-32 and RCR-64, in an attempt to rule out all the attacks against the Py-family of ciphers. We conjecture that attacks lower than brute force are not possible on RCR ciphers.

Our present work leaves room for interesting future work. The usage of long keys and IVs (e.g., possibility of 256-byte keys and 64-byte IVs) in RCR ciphers makes them good candidates to be used as hash functions. One can also try to combine a MAC and an encryption algorithm in a single primitive using RCR ciphers. It seems worthwhile to address these issues in future.

References

[1] T. Baignères, P. Junod and S. Vaudenay, "How Far Can We Go Beyond Linear Cryptanalysis?," *Asiacrypt 2004* (P. Lee, ed.), vol. 3329 of *LNCS*, pp. 432–450, Springer-Verlag, 2004. 10

[2] M. Bellare, T. Kohno, "A Theoretical Treatment of Related-Key Attacks: RKA-PRPs, RKA-PRFs, and Applications," EUROCRYPT 2003 (E. Biham, ed.), vol 2656 of LNCS, pp 2491-506, Springer-Verlag, 2003. 2

[3] E. Biham, "New Types of Cryptanalytic Attacks Using Related Keys," J. Cryptology, vol 7(4), pp. 229-246, 1994. 2

[4] E. Biham, J. Seberry, "Tweaking the IV Setup of the Py Family of Ciphers – The Ciphers Tpy, TPypy, and TPy6," Published on the author's webpage at `http://www.cs.technion.ac.il/ biham/`, January 25, 2007. 2, 4

[5] E. Biham, J. Seberry, "Py (Roo): A Fast and Secure Stream Cipher using Rolling Arrays," *ecrypt submission*, 2005. 1, 2, 4

[6] E. Biham, J. Seberry, "Pypy (Roopy): Another Version of Py," *ecrypt submission*, 2006. 1, 2, 4

[7] D. Chang, K. Gupta, M. Nandi, "RC4-Hash : A New Hash Function based on RC4 (Extended Abstract)," Indocrypt 2006 (R. Barua, T. Lange, eds.), vol 4329 of LNCS, Springer-Verlag, 2006. 2

[8] P. Crowley, "Improved Cryptanalysis of Py," *Workshop Record of SASC 2006 - Stream Ciphers Revisited*, ECRYPT Network of Excellence in Cryptology, February 2006, Leuven (Belgium), pp. 52-60. 1, 3

[9] S. Fluhrer, I. Mantin, A. Shamir, "Weaknesses in the Key Scheduling Algorithm of RC4," Selected Areas in Cryptography 2001 (S. Vaudenay, A. Youssef, eds.), vol 2259 of LNCS, pp. 1-24, Springer-Verlag, 2001. 2

[10] H. Handschuh, L. Knudsen, M. Robshaw, "Analysis of SHA-1 in Encryption Mode," CT-RSA 2001 (D. Naccache, ed), vol 2020 of LNCS, pp. 70-83, Springer-Verlag, 2001. 2

[11] H. Handschuh, D. Naccache, "SHACAL," First Nessie Workshop, Leuven, 2000. 2

[12] T. Isobe, T. Ohigashi, H. Kuwakado M. Morii, "How to Break Py and Pypy by a Chosen-IV Attack," eSTREAM, ECRYPT Stream Cipher Project, Report 2006/060. 1, 3

[13] J. Kelsey, B. Schneier, D. Wagner, "Related-key cryptanalysis of 3-WAY, Biham-DES, CAST, DES-X, NewDES, RC2, and TEA," ICICS 1997 (Y. Han, T. Okamoto, S. Qing, eds.), vol 1334 of LNCS, pp 233-246, Springer-Verlag, 1997. 2

[14] J. Kelsey, B. Schneier, D. Wagner, "Key-Schedule Cryptoanalysis of IDEA, G-DES, GOST, SAFER, and Triple-DES," CRYPTO 1996 (N. Koblitz, ed.), vol. 1109 of LNCS, pp. 237-251, Springer-Verlag, 1996. 2

[15] L. R. Knudsen, "Cryptanalysis of LOKI," Advances in Cryptology ASIACRYPT '91 (H. Imai, R. Rivest, T. Matsumoto), vol 739 of LNCS, Springer-Verlag, 1993, pp. 22-35. 2

[16] L.R. Knudsen, "Cryptanalysis of LOKI91," Advances in Cryptology AUSCRYPT '92, Springer-Verlag, 1993, pp. 196-208. 2

[17] L. Knudsen, "A key-schedule weakness in SAFER K-64," In D. Coppersmith, editor, Advances in Cryptology CRYPTO 95, volume 963 of Lecture Notes in Computer Science, pages 274286. Springer-Verlag, Berlin Germany, 1995. 2

[18] S. Lucks, "Ciphers Secure against Related-Key Attacks," FSE 2004 (W. Meier, B. Roy, eds.), vol 3017 of LNCS, pp 359-370, Springer-Verlag, 2004. 2

[19] O. Dunkelman, E. Biham, N. Kellar, "A Simple Related-Key Attack on the Full SHACAL-1," CT-RSA 2007 (Masayuki Abe, ed.), vol 4377 of LNCS, Springer-Verlag, 2007. 2

[20] S. Paul, B. Preneel, G. Sekar, "Distinguishing Attacks on the Stream Cipher Py," *Fast Software Encryption 2006* (M. Robshaw, ed.), vol. 4047 of *LNCS*, pp. 405-421, Springer-Verlag, 2006. 1, 3, 10, 11, 14

[21] S. Paul, B. Preneel "On the (In)security of Stream Ciphers Based on Arrays and Modular Addition," *Asiacrypt 2006* (X. Lai and K. Chen, eds.), vol. 4047 of *LNCS*, pp. 69-83, Springer-Verlag, 2006. 1, 3

[22] Research and Development in Advanced Communication Technologies in Europe, RIPE Integrity Primitives: Final Report of RACE Integrity Primitives Evaluation (R1040), RACE, Jun 1992. 2

[23] G. Sekar, S. Paul, B. Preneel, "New Weaknesses in the Keystream Generation Algorithms of the Stream Ciphers TPy and Py," available at `http://eprint.iacr.org/2007/230.pdf`. 2, 3, 11

[24] G. Sekar, S. Paul and B. Preneel, "Attacks on the Stream Ciphers TPy6 and Py6 and Design of New Ciphers TPy6-A and TPy6-B," available at `http://eprint.iacr.org/2007/436.pdf`. 2, 3

[25] G. Sekar, S. Paul, B. Preneel, "Weaknesses in the Pseudorandom Bit Generation Algorithms of the Stream Ciphers TPypy and TPy," available at `http://eprint.iacr.org/2007/075.pdf`. 2, 3, 11

[26] R. Winternitz, "Producing One-Way Hash Functions from DES," Advances in Cryptology: Proceedings of Crypto 83, Plenum Press, 1984, pp. 203-207. 2

[27] X. Wang, A. Yao, Frances Yao, "Cryptanalysis on SHA-1," Cryptographic Hash Workshop, NIST, Gaithersburg, 2005. 2

[28] X. Wang, X. Lai, D. Feng, H. Chen, X. Yu, "Cryptanalysis of the Hash Functions MD4 and RIPEMD," Advances in Cryptology, proceedings of EUROCRYPT 2005, Lecture Notes in Computer Science 3494, pp. 118, 2005.

[29] Xiaoyun Wang, Yiqun Lisa Yin, Hongbo Yu, "Finding Collisions in the Full SHA-1," Advances in Cryptology, proceedings of CRYPTO 2005, Lecture Notes in Computer Science 3621, pp. 1736, 2005. 2

[30] X. Wang, H. Yu, "How to Break MD5 and Other Hash Functions," Advances in Cryptology, proceedings of EUROCRYPT 2005, Lecture Notes in Computer Science 3494, pp. 1935, 2005.

[31] X. Wang, H. Yu, Y. L. Yin, "Efficient Collision Search Attacks on SHA-0," Advances in Cryptology, proceedings of CRYPTO 2005, Lecture Notes in Computer Science 3621, pp. 116, 2005. 2

[32] H. Wu and B. Preneel, "Differential Cryptanalysis of the Stream Ciphers Py, Py6 and Pypy," *Eurocrypt 2007* (Moni Naor, ed), vol. 4515 of LNCS, pp. 276-290, Springer-Verlag, 2007. 3, 11

A Various Parts of Py-family of Ciphers

The algorithms are shown in the next page.

B Effect of Any Non-zero Constant Rotation in RCR Ciphers

The distinguishing attacks presented in [20] are based on the fact that, when certain conditions on the elements of array P are satisfied then $s_{r(i)} = s_{r+2(j)}$, where r denotes the round and i, j $(0 \leq i,\ j \leq 31)$ denote the bit positions.

We now examine the effect of constant rotation (say c) in step 4 of the PRBGs of TPy and TPypy (see Algorithm 5).

$$\begin{aligned} s_{r(i)} &= ROTL32(s_{r-1} + Y_r[P_r[72]] - Y_r[P_r[239]], c)_i && (21) \\ &= (s_{r-1} + Y_r[P_r[72]] - Y_r[P_r[239]])_{i-c \bmod 32}. && (22) \end{aligned}$$

Let k denote $i - c \bmod 32$. Therefore,

$$s_{r(i)} = s_{r-1(k)} \oplus Y_r[P_r[72]]_k \oplus Y_r^c[P_r[239]]_k \oplus e_{r(k)},$$

where e denotes the carry term generated in (22) and $e_{r(0)} = 1$.

Similarly, if l denotes $j - c \bmod 32$, we have,

$$s_{r+2(j)} = s_{r+1(l)} \oplus Y_{r+2}[P_{r+2}[72]]_l \oplus Y_{r+2}^c[P_{r+2}[239]]_l \oplus e_{r+2(l)}. \quad (23)$$

Again, we have

$$s_{r+1(l)} = s_{r(m)} \oplus Y_{r+1}[P_{r+1}[72]]_m \oplus Y_{r+1}^c[P_{r+1}[239]]_m \oplus e_{r+1(m)}, \quad (24)$$

where m denotes $l - c \bmod 32$, and

$$s_{r(m)} = s_{r-1(n)} \oplus Y_r[P_r[72]]_n \oplus Y_r^c[P_r[239]]_n \oplus e_{r(n)}, \quad (25)$$

where n denotes $m - c \bmod 32$. Substituting (24) and (25) in (23), we get that the expression for $s_{r(i)} \oplus s_{r+2(j)}$ contains the term $s_{r-1(k)} \oplus s_{r-1(n)}$. It now follows that if $k \neq n$, it is very likely that the terms $s_{r(i)}$ and $s_{r+2(j)}$ are not correlated. Besides, we have a number of Y-terms at different bit-positions and the terms do not cancel out if $i \neq j$.

Now, $n = j - 3c \bmod 32$ and $k = i - c \bmod 32$. Hence, when $i = j$, we have $c \neq 0$ in order that $k \neq n$ be satisfied. Thus, with $c = 19$, we expect that there will be no correlations in the output stream in order that a distinguisher be built with data complexity less than that of exhaustive search. The constant 19 is not influenced by any factors and any non-zero constant is expected to work.

Algorithm 2 Key setup: KS

Require: A key, an IV and an initial permutation

Ensure: An array $Y[-3, \ldots, 256]$ and a 32-bit variable s

```
keysizeb = size of key in bytes;
ivsizeb = size of IV in bytes;
YMININD=-3;
YMAXIND=256;

s = internal_permutation[keysizeb-1];
s = (s<<8) | internal_permutation[(s ^(ivsizeb-1))&0xFF];
s = (s<<8) | internal_permutation[(s ^ key[0])&0xFF];
s = (s<<8) | internal_permutation[(s ^ key[keysizeb-1])&0xFF];

for(j=0; j<keysizeb; j++)
 {
   s = s + key[j];
   s0 = internal_permutation[s&0xFF];
   s = ROTL32(s, 8) ^ (u32)s0;
 }

/* Again */
for(j=0; j<keysizeb; j++)
 {
   s = s + key[j];
   s0 = internal_permutation[s&0xFF];
   s ^= ROTL32(s, 8) + (u32)s0;
 }

/* Initialize the array Y */
for(i=YMININD, j=0; i<=YMAXIND; i++)
 {
   s = s + key[j];
   s0 = internal_permutation[s&0xFF];
   Y(i) = s = ROTL32(s, 8) ^ (u32)s0;
   j = j+1 mod keysizeb;
 }
```

Algorithm 3 Part I of the IV setup algorithms of IVS_1 and IVS_2 - initialization of P and EIV

Require: The Y, the s from the key setup algorithm and the IV

Ensure: Rolling arrays $P[0,\ldots,255]$, $EIV[0,\ldots,ivsizeb-1]$, the variable s

```
/* Create an initial permutation */

 u8 v= iv[0] ^ ((Y(0)>>16)&0xFF);
 u8 d=(iv[1 mod ivsizeb] ^ ((Y(1)>>16)&0xFF))|1;

 for(i=0; i<256; i++)
  {
    P(i)=internal_permutation[v];
    v+=d;
  }
/* Now P is a permutation */

/* Initialize s */
 s = ((u32)v<<24) ^ ((u32)d<<16) ^ ((u32)P(254)<<8) ^ ((u32)P(255));
 s ^= Y(YMININD)+Y(YMAXIND);

 for(i=0; i<ivsizeb; i++)
  {
   s = s + iv[i] + Y(YMININD+i);
   u8 s0 = P(s&0xFF);
   EIV(i) = s0;
   s = ROTL32(s, 8) ^ (u32)s0;
 }

/* Again, but with the last words of Y, and update EIV */

 for(i=0; i<ivsizeb; i++)
  {
    s = s + iv[i] + Y(YMAXIND-i);
    /*s = s + EIV((i+ivsizeb-1)mod ivsizeb) + Y(YMAXIND-i); for IVS1.*/
    u8 s0 = P(s&0xFF);
    EIV(i) += s0;
    s = ROTL32(s, 8) ^ (u32)s0;
  }
```

Algorithm 4 Part II of the IV setup algorithms IVS_1 and IVS_2- updating the rolling arrays and the variable s

Require: Outputs of the Part I of IV setup

Ensure: The rolling arrays $Y[-3,\ldots,256]$, $P[0,\ldots,255]$ and the variable s

```
for(i=0; i<260; i++)
 {
   u32 x0 = EIV(0) = EIV(0) ^ (s&0xFF);
   rotate(EIV);
   swap(P(0), P(x0));
   rotate(P);
   Y(YMININD)=s=(s ^ Y(YMININD))+Y(x0);
   /*s=ROTL32(s,8)+Y(YMAXIND);
   Y(YMININD)+=s^Y(x0); for IVS1.*/
   rotate(Y);
 }

s=s+Y(26)+Y(153)+Y(208);

if(s==0)

s=(keysizeb*8)+((ivsizeb*8)<<16)+0x87654321;
```

Algorithm 5 Round functions: RF_1 and RF_2

Require: $Y[-3,...,256]$, $P[0,...,255]$, a 32-bit variable s

Ensure: 32-bit random output (for RF_1) or 64-bit random output (for RF_2)

/*Update and rotate P*/

1: swap ($P[0]$, $P[Y[185]\&255]$);

2: rotate (P);

/* Update s*/

3: $s+=Y[P[72]]-Y[P[239]]$;

4: $s=ROTL32(s,((P[116]+18)\&31))$;

/* Output 4 or 8 bytes (least significant byte first)*/

5: output $((ROTL32(s,25)\oplus Y[256])+Y[P[26]])$;/* This step is skipped for RF_1.*/

6: output $((\quad s \quad \oplus Y[-1])+Y[P[208]])$;

/* Update and rotate Y*/

7: $Y[-3]=(ROTL32(s,14)\oplus Y[-3])+Y[P[153]]$;

8: rotate(Y);

C Description of Algorithm C and Table 6

Here, we describe Algorithm C which constitutes the third *for*-loop of the key setup algorithm KS.

```
Algorithm C
  for(i=YMININD, j=0; i<=YMAXIND; i++)
  {
    s = s + key[j];
    s0 = internal_permutation[s&0xFF];
    Y(i) = s = ROTL32(s, 8) ^ (u32)s0;
    j = j+1 mod keysizeb;
  }
```

Table 6: s and Y after rounds 15, 16 and 17 of Algorithm C given event $D_2 \cap D_1$ occurs.

End of round	s (using key_1)	s (using key_2)	Y (using key_1)	Y (using key_2)
15	$s^C_{1,15}$	$s^C_{2,15} = s^C_{1,15}$	$Y_1[12]$	$Y_2[12] = Y_1[12]$
16	$s^C_{1,16}$	$s^C_{2,16} = s^C_{1,16} - \delta_3$ (say)	$Y_1[13]$	$Y_2[13] \neq Y_1[13]$
17	$s^C_{1,17}$	$s^C_{2,17} = s^C_{1,17}$ if $key_2[17] = key_1[17] + \delta_3$	$Y_1[14]$	$Y_2[14] = Y_1[14]$ if $key_2[17] = key_1[17] + \delta_3$

In Table 6,

$$\begin{aligned} \delta_3 &= s^C_{1,16} - s^C_{2,16} \\ &= ROTL32((s^C_{1,15} + key_1[16]), 8) \oplus ip[B(s^C_{1,15} + key_1[16])] \\ &- ROTL32((s^C_{2,15} + key_2[16]), 8) \oplus ip[B(s^C_{2,15} + key_2[16])]. \end{aligned} \quad (26)$$

D Related Keys When Size of the IV is Varied

As mentioned in Sect. 4.2, for longer IVs, one can induce the first difference in the keys (that is, where the least significant bits alone differ) accordingly as the size of the IV used. For example, when the size of the IV is 32 bytes, we take two keys, key_1 and key_2 (each key is 256 bytes long), such that,

1. $key_1[32] \oplus key_2[32] = 1$,
2. the lsb of $key_1[32]$ is 1, and
3. $key_1[33] \neq key_2[33]$.
4. $key_1[i] = key_2[i]\ \forall i \notin \{32, 33\}$.

More generally, if the IV is of size N bytes, the first difference in the keys should *not* be induced anywhere: neither (1) in the first $N-1$ bytes (i.e., key bytes 0 to $N-1$), nor (2) in the last $N-3$ bytes (i.e., key bytes $260-N$ to 256). Otherwise, it is immaterial as to where the first difference is set, that is, anywhere from byte N to byte $259-N$.