Differential Cryptanalysis of the Stream Ciphers Py, Py6 and Pypy ⋆

Hongjun Wu and Bart Preneel

Katholieke Universiteit Leuven, ESAT/SCD-COSIC
Kasteelpark Arenberg 10, B-3001 Leuven-Heverlee, Belgium
{wu.hongjun,bart.preneel}@esat.kuleuven.be

Abstract. Py and Pypy are efficient array-based stream ciphers designed by Biham and Seberry. Both were submitted to the eSTREAM competition. This paper shows that Py and Pypy are practically insecure. If one key is used with about 2^{16} IVs with special differences, with high probability two identical keystreams will appear. This can be exploited in a key recovery attack. For example, for a 16-byte key and a 16-byte IV, 2^{23} chosen IVs can reduce the effective key size to 3 bytes. For a 32-byte key and a 32-byte IV, the effective key size is reduced to 3 bytes with 2^{24} chosen IVs. Py6, a variant of Py, is more vulnerable to these attacks.

Keywords. Differential Cryptanalysis, Stream Cipher, Py, Py6, Pypy.

1 Introduction

RC4 has inspired the design of a number of fast stream ciphers, such as ISAAC [8], Py [2], Pypy [3] and MV3 [10]. RC4 was designed by Rivest in 1987. Being the most widely used software stream cipher, RC4 is extremely simple and efficient. At the time of the invention of RC4, its array based design was completely different from the previous stream ciphers mainly based on linear feedback shift registers.

There are two main motives to improve RC4. One motive is that RC4 is byte oriented, so we need to design stream ciphers that can run more efficiently on today's 32-bit microprocessors. Another motive is to strengthen RC4 against various attacks [7, 11, 16, 5, 6, 12, 15, 17, 13, 14]. Two of these attacks affect the security of RC4 in practice: the broadcast attack which exploits the weakness that the first few keystream bytes are heavily biased [12], and the key recovery attack using related IVs [6] which results in the practical attack on RC4 in WEP [13]. These two serious weaknesses are caused by the imperfection in the initialization of RC4.

Recently Biham and Seberry proposed the stream cipher Py [2] which is related to the design of RC4. Py is one of the fastest stream ciphers on 32-bit

⋆ This work was supported in part by the Concerted Research Action (GOA) Ambiorics 2005/11 of the Flemish Government and in part by the European Commission through the IST Programme under Contract IST-2002-507932 ECRYPT.

processors (about 2.5 times faster than RC4). A distinguishing attack against Py was found by Paul, Preneel and Sekar [18]. In that attack, the keystream can be distinguished from random with about 2^{88} bytes. Later, the attack was improved by Crowley [4], and the data required in the attack is reduced to 2^{72}. In order to resist the distinguishing attack on Py, the designers of Py decided to discard half of the outputs, i.e., the first output of the two outputs at each step is discarded. The new version is called Pypy [3]. Py and Pypy are selected as focus ciphers in the Phase 2 of the ECRYPT eSTREAM project.

The initializations of Py and Pypy are identical. In this paper, we show that there are serious flaws in the initialization of Py and Pypy, thus these two ciphers are vulnerable to differential cryptanalysis [1]. Two keystreams can be identical if a key is used with about 2^{16} IVs with special differences. It is a practical threat since the set of IVs required in the attack may appear with high probability in applications. Then we show that part of the key of Py and Pypy can be recovered with chosen IVs. For a 16-byte key and a 16-byte IV, 2^{23} chosen IVs can reduce the effective key size to 3 bytes.

Py6 [2] is a variant of Py with reduced internal state size. We show that Py6 is more vulnerable to the attacks against Py and Pypy.

This paper is organized as follows. In Sect. 2, we illustrate the Key and IV setups of Py and Pypy. Section 3 describes the attack of generating identical keystreams. The key recovery attack is given in Sect. 4. In Sect. 5, we outline the attacks against Py6. Section 6 concludes this paper.

2 The Specifications of Py and Pypy

Py and Pypy are two synchronous stream ciphers supporting key and IV sizes up to 256 bytes and 64 bytes, respectively. The initializations of Py and Pypy are identical. The initialization consists of two stages: key setup and IV setup.

In the following descriptions, P is an array with 256 8-bit elements. Y is an array with 260 32-bit elements, s is a 32-bit integer. $YMININD = -3$, $YMAXIND = 256$. The table 'internal_permutation' is a constant permutation table with 256 elements. '$\wedge$' and '&' in the pseudo codes denote binary XOR and AND operations, respectively. 'u8' and 'u32' mean 'unsigned 8-bit integer' and 'unsigned 32-bit integer', respectively. 'ROTL32(a,n)' means that the 32-bit a is left rotated over n bits.

2.1 The key setup

The key setups of Py and Pypy are identical. In the key setup, the key is used to initialize the array Y. The description is given below.

```
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;
}
/* Algorithm C is the following 'for' loop */
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;
}
```

2.2 The IV setup

The IV setups of Py and Pypy are identical. In the IV setup, the IV is used to affect every bit of the internal state. *EIV* is a temporary byte array with the same size as the IV. The IV setup is given below.

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

/* Algorithm A is the following 'for' loop */
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 */
/* Algorithm B is the following ‘for’ loop */
for(i=0; i<ivsizeb; i++)
{
     s = s + iv[i] + Y(YMAXIND-i);
     u8 s0 = P(s&0xFF);
     EIV(i) += s0;
     s = ROTL32(s, 8) ^ (u32)s0;
}

/*updating the rolling array and 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);
     rotate(Y);
}
s=s+Y(26)+Y(153)+Y(208);
if(s==0)
     s=(keysizeb*8)+((ivsizeb*8)<<16)+0x87654321;
```

2.3 The keystream generation

After the key and IV setup, the keystream is generated. One step of the keystream generation of Py is given below. Note that the first output at each step is discarded in Pypy.

```
/* swap and rotate P */
swap(P(0), P(Y(185)&0xFF));
rotate(P);

/* Update s */
s+=Y(P(72)) - Y(P(239));
s=ROTL32(s, ((P(116) + 18)&31));

/* Output 8 bytes (least significant byte first) */
output ((ROTL32(s, 25) ^ Y(256)) + Y(P(26)));
```

```
output (( s ^ Y(-1)) + Y(P(208)));

/* Update and rotate Y */
Y(-3)=(ROTL32(s, 14) ^ Y(-3)) + Y(P(153));
rotate(Y);
```

3 Identical Keystreams

We notice that the IV appears only in the IV setup algorithm described in Sect. 2.2. At the beginning of the IV setup, only 15 bits of the IV ($iv[0]$ and $iv[1]$) are applied to initialize the array P and s (the least significant bit of $iv[1]$ is not used). For an IV pair, if those 15 bits are identical, then the resulting P are the same. Then we notice that the IV is applied to update s and EIV as follows.

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

We call the first 'for' loop Algorithm A, and the second 'for' loop Algorithm B. In the following, we give two types of IV pairs that result in identical keystreams.

3.1 IVs differing in two bytes

We illustrate the attack with an example. Suppose that two IVs, iv_1 and iv_2, differing in only two consecutive bytes with $iv_1[i] \oplus iv_2[i] = 1$, the least significant bit of $iv_1[i]$ is 1, $iv_1[i+1] \neq iv_2[i+1]$ $(1 \leq i \leq ivsizeb - 1)$, and $iv_1[j] = iv_2[j]$ for $0 \leq j < i$ and $i + 1 < j \leq ivsizeb - 1$. We trace how the difference in IV affects s and EIV in Algorithm A. At the ith step in Algorithm A,

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

At the end of the ith step, $EIV_1[i] \neq EIV_2[i]$. Let $\beta_1 = EIV_1[i]$, and $\beta_2 = EIV_2[i]$. We obtain that $s_1 - s_2 = 256 + \delta_1$, where $\delta_1 = (\beta_1 \oplus x) - (\beta_2 \oplus x)$, and $x = \text{ROTL32}(s, 8)$. Then we look at the next step.

```
s = s + iv[i+1] + Y(YMININD+i+1);
u8 s0 = P(s&0xFF);
EIV(i+1) = s0;
s = ROTL32(s, 8) ^ (u32)s0;
```

Because $iv_1[i+1] \neq iv_2[i+1]$, if $iv_2[i+1] - iv_1[i+1] = \delta_1$, then s_1 and s_2 become identical with high probability. Let $s_1 = s_2$ with probability p_1. Based on the simulation, we obtain that $p_1 = 2^{-10.6}$. If $s_1 = s_2$, then $EIV_1[i+1] = EIV_2[i+1]$, and in the following steps $i+2, i+3, \cdots, i+ivsizeb-1$ in Algorithm A, s_1 and s_2 remain the same, and $EIV_1[j] = EIV_2[j]$ for $j \neq i$.

After Algorithm A, the $iv[i]$ and $iv[i+1]$ are used again to update s and EIV in Algorithm B. At the ith step in Algorithm B,

```
s = s + iv[i] + Y(YMAXIND-i);
u8 s0 = P(s&0xFF);
EIV(i) += s0;
s = ROTL32(s, 8) ^ (u32)s0;
```

At the end of this step, $EIV_1[i] = EIV_2[i]$ with probability $\frac{1}{255}$. Let $\gamma_1 = s0_1$, and $\gamma_2 = s0_2$. If $EIV_1[i] = EIV_2[i]$, we know that $\gamma_2 - \gamma_1 = \beta_1 - \beta_2$. At the end of this step, $s_1 - s_2 = 256 + \delta_2$, where $\delta_2 = (\gamma_1 \oplus y) - (\gamma_2 \oplus y)$, and y is ROTL32(s,8). Note that δ_1 and δ_2 are correlated since $\gamma_2 - \gamma_1 = \beta_1 - \beta_2$. Then we look at the next step.

```
s = s + iv[i+1] + Y(YMAXIND-i-1);
u8 s0 = P(s&0xFF);
EIV(i+1) += s0;
s = ROTL32(s, 8) ^ (u32)s0;
```

At the end of this step, if $iv_2[i+1] - iv_1[i+1] = \delta_2$, then s_1 and s_2 become identical with high probability. Note that $iv_2[i+1] - iv_1[i+1] = \delta_1$, and δ_1 and δ_2 are correlated, so $iv_2[i+1] - iv_1[i+1] = \delta_2$ with probability larger than 2^{-8}. Let $s_1 = s_2$ with probability p'_1. Based on a simulation, we obtain that $p'_1 = 2^{-5.6}$. Once the two s values are identical, $EIV_1[i+1] = EIV_2[i+1]$, and in the following steps $i+2, i+3, \cdots, i+ivsize-1$ in Algorithm B, s_1 and s_2 remain the same, and $EIV_1[i+2] = EIV_2[i+2]$, $EIV_1[i+3] = EIV_2[i+3]$, $\cdots$, $EIV_1[i+ivsize-1] = EIV_2[i+ivsize-1]$.

Thus after introducing the IV to update s and EIV, $s_1 = s_2$ and $EIV_1 = EIV_2$ with probability $p_1 \times \frac{1}{255} \times p'_1 \approx 2^{-24.2}$.

Note that once an IV has been introduced in Algorithm A and B, the IV is not used in the rest of the IV setup. Thus once $s_1 = s_2$ and $EIV_1 = EIV_2$ at the end of Algorithm B, we know that those two keystreams will be the same.

Experiment 1. We use 2^{14} random 128-bit keys in the attack. For each key, we randomly generate 2^{16} pairs of 128-bit IV that differ in only two bytes: $iv_1[6] \oplus iv_2[6] = 1$, $iv_1[7] \neq iv_2[7]$. We found that 111 pairs of those 2^{30} keystream pairs are identical. For example, for the key (08 da f2 35 a3 d5 94 e2 85 cc 68 ba 7e 10 8a b4), and the IV pair (6e e7 09 b1 35 85 2f 07 1a fe 3f 50 a8 84 30 11) and (6e e7 09 b1 35 85 2e 80 1a fe 3f 50 a8 84 30 11), the two keystreams are identical, and the first 16 keystream bytes of Pypy are (6f eb ca 18 54 3f 59 96 b6 17 8a 54 6e bd 45 1f).

From the experiment, we deduce that for an IV pair with the required difference, the two keystreams are identical with probability about $\frac{111}{2^{30}} = 2^{-23.2}$, about twice the theoretical value.

The IV difference at two bytes. In the above analysis, the difference is chosen as $iv_1[i] \oplus iv_2[i] = 1$, $iv_1[i+1] \neq iv_2[i+1]$ $(i \geq 1)$. We can generalize this type of IV difference so that $iv_1[i]$ and $iv_2[i]$ can take other differences. **As long as** $(iv_1[i] - iv_2[i])$ **mod** $256 = 1$ **or** 255, $iv_1[i+1] \neq iv_2[i+1]$ $(i \geq 2)$**, there is a non-zero probability that the two keystreams can be identical.**

For example, if $iv_1[i] \oplus iv_2[i] = 3$, the two least significant bits of $iv_1[i]$ are 01 or 10, and $iv_1[i+1] \neq iv_2[i+1]$ $(i \geq 2)$, then two identical keystreams appear with probability $2^{-23.2}$. On average, if $iv_1[i] - iv_2[i] = 1$, and $iv_1[i+1] \neq iv_2[i+1]$ $(i \geq 2)$, then two identical keystreams appear with probability $2^{-26.4}$.

3.2 IVs differing in three bytes

In the above attack, we deal with the ith and $(i+1)$th bytes of the IV, and use the difference at $iv[i+1]$ to eliminate the difference introduced by $iv[i]$ in s. In the following, we introduce another type of difference to deal with the situation when the difference at $iv[i+1]$ cannot eliminate the difference introduced by $iv[i]$ in s. The solution is to introduce a difference in $iv[i+4]$.

We illustrate the attack with an example. Suppose that two IVs, iv_1 and iv_2, differ in only three bytes $iv_1[i] \oplus iv_2[i] = 0x80$, the most significant bit of $iv_1[i]$ is 1, $iv_1[i+1] \neq iv_2[i+1]$, $iv_1[i+4] \oplus iv_2[i+4] = 0x80$, and the most significant bit of $iv_1[i+4]$ is 0, where $i \geq 2$. We trace how the difference affects s and EIV. At the ith step in Algorithm A,

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

At the end of this step, $EIV_1[i] \neq EIV_2[i]$, and $s_1 - s_2 = 0x8000 + \delta_1$, where δ_1 is the difference of two different 8-bit numbers. Then we look at the next step.

```
s = s + iv[i+1] + Y(YMININD+i+1);
u8 s0 = P(s&0xFF);
```

```
EIV(i+1) = s0;
s = ROTL32(s, 8) ^ (u32)s0;
```

Because $iv_1[i+1] \neq iv_2[i+1]$, $s_1 - s_2 = 0x8000$ with probability $p_2 = 2^{-8}$. If $s_1 - s_2 = 0x8000$, then $EIV_1[i+1] \oplus EIV_2[i+1] = 0$.

Since $v_1[i+2] = v_2[i+2]$, at the end of the $(i+2)$th step of Algorithm A, $EIV_1[i+2] = EIV_2[i+2]$, and $s_1 - s_2 = 0x800000$ with probability close to 1.

Since $v_1[i+3] = v_2[i+3]$, at the end of the $(i+3)$th step of Algorithm A, $EIV_1[i+3] = EIV_2[i+3]$, and $s_1 - s_2 = 0x80000000$ with probability close to 1. Now consider the $(i+4)$th step.

```
s = s + iv[i+4] + Y(YMININD+i+4);
u8 s0 = P(s&0xFF);
EIV(i+4) = s0;
s = ROTL32(s, 8) ^ (u32)s0;
```

At the end of this step, the probability that $EIV_1[i+4] = EIV_2[i+4]$, and $s_1 = s_2$ is 1. So for the above 5 steps, $s_1 = s_2$ with probability p_2. Once $s_1 = s_2$, in the following steps $i+5, i+6, \cdots, i+ivsize-1$ in Algorithm A, the s_1 and s_2 remain the same, and $EIV_1[i+5] = EIV_2[i+5]$, $EIV_1[i+6] = EIV_2[i+6]$, $\cdots$, $EIV_1[i+ivsize-1] = EIV_2[i+ivsize-1]$.

Then $iv[i]$ and $iv[i+1]$ are used again to update s and EIV. With a similar analysis, we can show that at the end of the updating, $EIV_1 = EIV_2$, $s_1 = s_2$ with probability about $(p_2)^2 \times \frac{1}{255} \approx 2^{-24}$. (As shown in the experiment in the next subsection, this probability is about $2^{-22.9}$.)

The IV difference at three bytes. In the above analysis, the difference is chosen at only three bytes, $iv_1[i] \oplus iv_2[i] = 0x80$, the most significant bit of $iv_1[i]$ is 1, $iv_1[i+1] \neq iv_2[i+1]$, $iv_1[i+4] \oplus iv_2[i+4] = 0x80$, and the most significant bit of $iv_1[i+4]$ is 0 ($i \geq 2$). For this type of IV difference, we can generalize it so that $iv_1[i]$ and $iv_2[i]$ can choose other differences instead of 0x80. In fact, **once we set the difference as $iv_1[i] - iv_2[i] = iv_2[i+4] - iv_1[i+4]$, $iv_1[i+1] \neq iv_2[i+1]$ ($i \geq 2$), then the two keystreams are identical with probability close to 2^{-23}**. For two IVs different only at three bytes, if $iv_1[1] \oplus iv_2[1] = 1$, $iv_1[2] \neq iv_2[2]$, and $iv_1[1] - iv_2[1] = iv_2[5] - iv_1[5]$, then this IV pair is also weak.

3.3 Improving the attack

The number of IVs required to generate identical keystreams can be reduced in practice. The idea is to generate more IV pairs from a group of IVs. For the IV pair with a two-byte difference $iv_1[i] \oplus iv_2[i] = 1$, $iv_1[i+1] \neq iv_2[i+1]$, if $iv[2]$ takes all the 256 values, then we can obtain $255 \times 255 = 2^{15.99}$ IV pairs with the required differences from 512 IVs. Thus with 512 chosen IVs, the probability that there is one pair of identical keystreams becomes $2^{15.99} \times 2^{-23.2} \approx 2^{-7.2}$. With about $2^{7.2} \times 512 = 2^{16.2}$ IVs, identical keystreams can be obtained.

Experiment 2. We use 2^{16} random 128-bit keys in the improved attack. For each key, we generate 512 128-bit IVs with the values of the least significant bit of $iv[4]$ and the eight bits of $iv[5]$ choosing all the 512 possible values, while all the other 119 IV bits remain unchanged for each key (but those 119 IV bits are random from key to key). Then we obtain $255 \times 255 = 2^{15.99}$ IV pairs with the required difference. Among these $2^{16} \times 2^{15.99} \approx 2^{32}$ IV pairs, 447 IV pairs result in identical keystreams.

The above experiment shows that with $2^{16} \times 512 = 2^{25}$ selected IVs, 447 IVs result in identical keystreams. It shows that two identical keystreams appear for every $\frac{2^{25}}{447} = 2^{16.2}$ IVs.

For the IV pair with three-byte difference, a similar improvement can also be applied.

Experiment 3. We use 2^{16} random 128-bit keys in the improved attack. For each key, we generate 512 128-bit IVs with the values of the most significant bit of $iv[4]$ and the eight bits of $iv[5]$ choosing all the 512 possible values, and the most significant bit of $iv[8]$ is different from the most significant bit of $iv[4]$, while all the other 118 IV remain unchanged for each key (but those 118 IV bits are randomly generated for each key). Then we obtain $255 \times 255 = 2^{15.99}$ IV pairs with the required difference. Among these $2^{16} \times 2^{15.99} \approx 2^{32}$ IV pairs, 570 IV pairs result in identical keystreams.

The above experiment shows that with $2^{16} \times 512 = 2^{25}$ selected IVs, 570 IVs result in identical keystreams. It means that two identical keystreams appear for every $\frac{2^{25}}{570} = 2^{15.9}$ IVs.

Remarks. The attacks show that the Py and Pypy are practically insecure. In the application, if the IVs are generated from a counter, or if the IV is short (such as 3 or 4 bytes), then the special IVs (with the differences as illustrated above) appear with high probability, and identical keystreams can be obtained with high probability.

4 Key Recovery Attack on Py and Pypy

In this section, we develop a key recovery attack against Py and Pypy by exploiting the collision in the internal state. The key recovery attack consists of two stages: recovering part of the array Y in the IV setup and recovering the key information from Y in the key setup.

4.1 Recovering part of the array Y

We use the following IV differences to illustrate the attack (the other IV differences can also be used). Let two IVs iv_1 and iv_2 differ only in two bytes,

$iv_1[i] \oplus iv_2[i] = 1$, $iv_1[i+1] \neq iv_2[i+1]$ $(i \geq 1)$, and the least significant bit of $iv_1[i]$ be 1. This type of IV pair results in identical keystreams with probability $2^{-23.2}$.

We first recover part of Y from Algorithm A in the IV setup (more information of Y will be recovered from Algorithm B).

Note that the permutation P in Algorithm A is unknown. According to the IV setup algorithm, there is 15 bits of secret information in P, i.e., there are at most 2^{15} possible permutations. During the recovery of Y, we assume that P is known (the effect of the 15-bit secret information in P will be analyzed in Sect. 4.2). For iv_m, denote the s at the end of the jth step of Algorithm A as s_j^m, and denote the least and most significant bytes of s_j^m as $s_{j,0}^m$ and $s_{j,3}^m$, respectively. Denote the least and most significant bytes of $Y(j)$ with $Y_{j,0}$ and $Y_{j,3}$, respectively. Note that in Algorithm A, Y remains the same for all the IVs. Denote ξ as a binary random variable with value 0 with probability 0.5. Denote with $B(x)$ a function that gives the least significant byte of x. If the keystreams for iv_1 and iv_2 identical, then from the analysis given in Sect. 3.1, we know that $s_{i+1}^1 = s_{i+1}^2$, i.e.,

$$s_i^1 + iv_1[i+1] = s_i^2 + iv_2[i+1]\,. \tag{1}$$

From Algorithm A, we know

$$\begin{aligned} s_i = {} & \mathrm{ROTL32}(s_{i-1} + iv[i] + Y(-3+i), 8) \\ & \oplus P(B(s_{i-1} + iv[i] + Y(-3+i))) \end{aligned} \tag{2}$$

Thus we obtain

$$s_{i,0} = P(B(s_{i-1,0} + iv[i] + Y(-3+i))) \oplus B(s_{i-1,3} + Y(-3+i) + \xi_i)\,, \tag{3}$$

$$(s_i^1 - s_{i,0}^1) - (s_i^2 - s_{i,0}^2) = (iv_1[i] - iv_2[i]) \ll 8 = 256\,, \tag{4}$$

where ξ_i is caused by the carry bits at the 24th least significant bit position when $iv[i]$ and $Y(-3+i)$ are introduced, and (4) holds with probability $1-2^{-15}$. From (1), (3) and (4), we obtain

$$\begin{aligned} & (P(B(s_{i-1,0}^1 + iv_1[i] + Y_{-3+i,0})) \oplus B(s_{i-1,3}^1 + Y_{-3+i,3} + \xi_{i,1})) + 256 + iv_1[i+1] \\ = {} & (P(B(s_{i-1,0}^2 + iv_2[i] + Y_{-3+i,0})) \oplus B(s_{i-1,3}^2 + Y_{-3+i,3} + \xi_{i,2})) + iv_2[i+1]\,, \end{aligned} \tag{5}$$

where $\xi_{i,1} = \xi_{i,2}$ with probability $1 - 2^{-15}$ since the $iv[i]$ has a negligible effect on the value of ξ_1 and ξ_2. In the following, we use ξ_i to represent $\xi_{i,1}$ and $\xi_{i,2}$.

Denote iv_θ as a fixed IV with the first i bytes being identical to all the IVs with differences only at $iv[i]$ and $iv[i+1]$. Thus $s_{i-1,0}^\theta = s_{i-1,0}^1 = s_{i-1,0}^2$, and $s_{i-1,3}^\theta = s_{i-1,3}^1 = s_{i-1,3}^2$. (5) becomes

$$\begin{aligned} & (P(B(s_{i-1,0}^\theta + iv_1[i] + Y_{-3+i,0})) \oplus B(s_{i-1,3}^\theta + Y_{-3+i,3} + \xi_i)) + 256 + iv_1[i+1] \\ = {} & (P(B(s_{i-1,0}^\theta + iv_2[i] + Y_{-3+i,0})) \oplus B(s_{i-1,3}^\theta + Y_{-3+i,3} + \xi_i)) + iv_2[i+1]\,. \end{aligned} \tag{6}$$

Using another IV pair different at $iv[i]$ and $iv[i+1]$, and the first i bytes being the same as iv_θ, another equation (6) can be obtained if there is collision in their internal states. Suppose that several equations (6) are available. We consider that the value of ξ_i is independent of $iv[i]$ in the following attack since ξ_i is affected by $iv[i]$ with small probability 2^{-15}. We can recover the values of $B(s^\theta_{i-1,0}+Y_{-3+i,0})$ and $B(s^\theta_{i-1,3}+Y_{-3+i,3}+\xi_i)$. From the experiment, we find that if there are two equations (6), on average the correct values can be recovered together with 5.22 wrong values. If there are three, four, five, six, seven equations (6), in average the correct values can be recovered together with 1.29, 0.54, 0.25, 0.12, 0.06 wrong values, respectively. It shows that the values of $B(s^\theta_{i-1,0}+Y_{-3+i,0})$ and $B(s^\theta_{i-1,3}+Y_{-3+i,3}+\xi_i)$ can be determined with only a few equations (6).

After recovering several consecutive $B(s^\theta_{i-1,0}+Y_{-3+i,0})$ and $B(s^\theta_{i-1,3}+Y_{-3+i,3}+\xi_i)$ $(i \geq 1)$, we proceed to recover part of the information of the array Y. From the values of $B(s^\theta_{i-1,0}+Y_{-3+i,0})$, $B(s^\theta_{i-1,3}+Y_{-3+i,3}+\xi_i)$ and (3), we determine the value of $s^\theta_{i,0}$. From the values of $B(s^\theta_{i,0}+Y_{-3+i+1,0})$ and $s^\theta_{i,0}$, we know the value of $Y_{-3+i+1,0}$.

Generating the equations (6). The above attack can only be successful if we can find several equations (6) with the same $s^\theta_{i-1,0}$ and $s^\theta_{i-1,3}$. In the following, we illustrate how to obtain these equations for $2 \leq i \leq ivsizeb-3$. At the beginning of the attack, we set a fixed iv_θ. For all the IVs different at only $iv[i]$ and $iv[i+1]$, we require that their first i bytes are identical to that of iv_θ. Let the least significant bit of $iv[i]$ and the 8 bits of $iv[i+1]$ choose all the 512 values, and the other 119 bits remain unchanged, then we obtain a $255 \times 255 \approx 2^{16}$ desired IV pairs. We call these 512 IVs a desired IV group. According to Experiment 2, this type of IV pair results in identical keystreams with probability $2^{-23.2}$, we thus obtain $\frac{2^{-23.2}}{2^{16}} = 2^{-7.2}$ identical keystream pairs from one desired IV group. It means that we can obtain $2^{-7.2}$ equations (1) from one desired IV group. We modify the values of the 7 most significant bits of $iv_1[i]$ and $iv_2[i]$, and 3 bits of $iv_1[i+2]$ and $iv_2[i+2]$, then we obtain $2^7 \times 2^3 = 2^{10}$ desired IV groups. From these desired IV groups, we obtain $2^{10} \times 2^{-7.2} = 7$ equations (1). There are $2^7 \times 2^3 \times 2^9 = 2^{19}$ IVs being used in the attack. To find all the $s_{i,0}$ for $2 \leq i \leq ivsizeb-3$, we need $(ivsizeb-4) \times 2^{19}$ IVs in the attack.

We are able to recover $s^\theta_{i,0}$ for $2 \leq i \leq ivsizeb-3$, which implies that we can recover the values of $Y_{-3+i,0}$ for $3 \leq i \leq ivsizeb-3$. Then we proceed to recover more information of Y by considering Algorithm B. Applying an attack similar to the above attack and reusing the IVs, we can recover the values of $Y_{256-i,0}$ for $3 \leq i \leq ivsizeb-3$.

Thus with $(ivsizeb-4) \times 2^{19}$ IVs, we are able to recover $2 \times (ivsizeb-6)$ bytes of Y: $Y_{-3+i,0}$ and $Y_{256-i,0}$ for $3 \leq i \leq ivsizeb-3$.

4.2 Recovering the key

In the above analysis, we recovered the values of $Y_{-3+i,0}$ and $Y_{256-i,0}$ for $3 \le i \le ivsizeb-3$ by exploiting the difference elimination in s. Next, we will recover the 15-bit secret information in P by exploiting the difference elimination in EIV. Denote s_i^{θ} in Algorithm A and B as $s_i^{A,\theta}$ and $s_i^{B,\theta}$, respectively. Denote $EIV_1[i]$ at the end of Algorithm A and B as ${EIV_1}^A[i]$ and ${EIV_1}^B[i]$, respectively. For two IVs differing in only $iv[i]$ and $iv[i+1]$ and generating identical keystreams, ${EIV_1}^A[i]$, ${EIV_2}^A[i]$, ${EIV_1}^B[i]$ and ${EIV_2}^B[i]$ are computed as:

$$
\begin{aligned}
{EIV_1}^A[i] &= P(B(s_{i-1,0}^{A,\theta} + iv_1[i] + Y_{-3+i,0})) && (7)\\
{EIV_2}^A[i] &= P(B(s_{i-1,0}^{A,\theta} + iv_2[i] + Y_{-3+i,0})) && (8)\\
{EIV_1}^B[i] &= {EIV_1}^A[i] + P(B(s_{i-1,0}^{B,\theta} + iv_1[i] + Y_{256-i,0})) && (9)\\
{EIV_2}^B[i] &= {EIV_2}^A[i] + P(B(s_{i-1,0}^{B,\theta} + iv_2[i] + Y_{256-i,0})) && (10)
\end{aligned}
$$

Since the two keystreams are identical, it is required that

$$
{EIV_1}^B[i] = {EIV_2}^B[i]\,. \tag{11}
$$

Note that the values of $B(s_{i-1,0}^{A,\theta} + Y_{-3+i,0})$ and $B(s_{i-1,0}^{B,\theta} + Y_{256-i,0})$ are determined when we recover part of Y from Algorithm A and Algorithm B, respectively. Eight bits of information on P is revealed from (7),(8),(9),(10) and (11). In Sect. 4.1, there are about 7 pairs of IVs resulting in identical keystreams for each value of i. Thus P can be recovered completely.

We proceed to recover the key information. We consider the last part of the key schedule:

```
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;
}
```

We call the above algorithm Algorithm C. From Algorithm C, we obtain the following relation:

$$
\begin{aligned}
&B(Y_{-3+i,0} + key[i+1 \bmod keysizeb] + \xi'_i)\\
&\oplus P'(B(Y_{-3+i+3,0} + key[i+4 \bmod keysizeb])) = Y_{-3+i+4,0}\,,
\end{aligned} \tag{12}
$$

where P' indicates the 'internal_permutation', ξ'_i indicates the carry bit noise introduced by $key[i+2]$ and $key[i+3]$; it is computed as $\xi'_i \approx (key[i+2] + Y_{-3+i+1,0}) \gg 8$. The value of the binary ξ'_i is 0 with probability about 0.5.

Once the values of $Y_{-3+i,0}$ $(3 \le i \le ivsizeb-3)$ are known, we find a relation (12) linking $key[i+1 \bmod keysizeb]$ and $key[i+4 \bmod keysizeb]$ for

$3 \leq i \leq ivsizeb - 7$. Each relation leaks at least 7 bits of $key[i+1 \bmod keysizeb]$ and $key[i+4 \bmod keysizeb]$. The values of $Y_{256-i,0}$ $(3 \leq i \leq ivsizeb - 3)$ are also known, thus we can find a relation (12) linking $key[i+1 \bmod keysizeb]$ and $key[i+4 \bmod keysizeb]$ for $262 - ivsizeb \leq i \leq 252$. Thus there are $2 \times (ivsizeb - 9)$ relations (12) linking the key bytes.

For the 16-byte key and 16-byte IV, 14 relations (12) can be obtained: 7 relations linking $key[i]$ and $key[i+3]$ for $4 \leq i \leq 10$, and another 7 relations (12) linking $key[i]$ and $key[i+3 \bmod 16]$ for $7 \leq i \leq 13$. There are 13 key bytes in these 14 relations (12). Note that the randomness of ξ'_i does not affect the overall attack (once we guess the values of $key[4]$, $key[5]$ and $key[6]$, then we obtain the other key bytes $key[j]$ $(7 \leq j \leq 15)$, $key[0]$, and all the ξ'_j $(3 \leq j \leq 9$ and $247 \leq j \leq 249)$. Thus these 14 relations are sufficient to recover the 13 key bytes. The effective key size is reduced to 3 bytes and these three bytes can be found easily with brute force search.

For the 32-byte key and 32-byte IV, 46 relations (12) can be obtained: 23 relations linking $key[i]$ and $key[i+3]$ for $4 \leq i \leq 26$, and another 23 relations (12) linking $key[i]$ and $key[i+3 \bmod 32]$ for $7 \leq i \leq 29$. There are 29 key bytes in these 46 relations. The effective key size is again reduced to 3 bytes.

5 The Security of Py6

Py6 is a variant of Py with reduced internal state size. The array P is a permutation with only 64 elements, and the array Y has 68 entries. Py6 was proposed to achieve fast initialization, but it is weaker than Py. Paul and Preneel has developed distinguishing attack against Py6 with data complexity $2^{68.6}$ [19]. In the following, we show that identical keystreams are genereated from Py6 with high probability. There is no detailed description of the key and IV setups of Py6. Thus we use the source code of Py6 submitted to eSTREAM as reference. In our experiment, the following IV differences are used: $iv_1[i] - iv_2[i] = 32$, $iv_1[i+1] \neq iv_2[i+1]$, $iv_1[i+1] \gg 6 = iv_2[i+1] \gg 6$, and $iv_2[i+5] - iv_1[i+5] = 8$ $(i \geq 2)$. After testing 2^{30} pairs with the original Py6 source code, we found that identical keystreams appear with probability $2^{-11.45}$. This probability is much larger than the probability 2^{-23} for Py and Pypy. It shows that Py6 is much weaker than Py and Pypy.

6 Conclusion

In this paper, we developed practical differential attacks against Py, Py6 and Pypy: the identical keystreams appear with high probability, and the key information can be recovered when the IV size is more than 9 bytes. To resist the attacks given in this paper, we suggest that the IV setup be performed in an invertible way.

Several ciphers in the eSTREAM competition have been broken due to the flaws in their IV setups: DECIM [20], WG [21], LEX [21], Py, Pypy and VEST [9]. We should pay great attention to the design of the stream cipher IV setup.

Acknowledgements

The authors would like to thank the anonymous reviewers of Eurocrypt 2007 for their helpful comments.

References

1. E. Biham, A. Shamir, "Differential Cryptanalysis of DES-like Cryptosystems." *Advances in Cryptology – Crypto'90*, LNCS 537, A. J. Menezes and S. A. Vanstone (Eds.), pp. 2–21, Springer-Verlag, 1991.
2. E. Biham, J. Seberry, "Py (Roo): A Fast and Secure Stream Cipher Using Rolling Arrays." The ECRYPT eSTREAM project Phase 2 focus ciphers. Available at http://www.ecrypt.eu.org/stream/ciphers/py/py.ps .
3. E. Biham, J. Seberry, "Pypy (Roopy): Another Version of Py." The ECRYPT eSTREAM project Phase 2 focus ciphers. Available at http://www.ecrypt.eu.org/stream/p2ciphers/py/pypy_p2.ps
4. P. Crowley, "Improved Cryptanalysis of Py." Available at http://www.ecrypt.eu.org/stream/papersdir/2006/010.pdf .
5. S. R. Fluhrer, D. A. McGrew, "Statistical Analysis of the Alleged RC4 Keystream Generator," *Fast Software Encryption – FSE 2000*, LNCS 1978, B. Schneier (Ed.), pp. 19–30, Springer-Verlag, 2000.
6. S. R. Fluhrer, I. Mantin, A. Shamir, "Weaknesses in the Key Scheduling Algorithm of RC4," *Selected Areas in Cryptography – SAC 2001*, LNCS 2259, S. Vaudenay and A.M. Youssef (Eds.), pp. 1–24, Springer-Verlag, 2001.
7. J. Golić, "Linear statistical weakness of alleged RC4 keystream generator," *Advances in Cryptology – Eurocrypt'97*, LNCS 1233, W. Fumy (Ed.), pp. 226–238, Springer-Verlag, 1997.
8. R. J. Jenkins Jr., "ISAAC," *Fast Software Encryption – FSE 1996*, LNCS 1039, D. Gollmann (Ed.), pp. 41–49, Springer-Verlag, 1996.
9. A. Joux, J. Reinhard, "Overtaking VEST." *Fast Software Encryption – FSE 2007*, LNCS, A. Biryukov (Ed.), Springer-Verlag, to appear.
10. N. Keller, S. D. Miller, I. Mironov, and R. Venkatesan, "MV3: A new word based stream cipher using rapid mixing and revolving buffers," *Topics in Cryptology – CT-RSA 2007, The Cryptographers' Track at the RSA Conference 2007*, LNCS 4377, M. Abe (Ed.), pp. 1–19, Springer-Verlag, 2006.
11. L. R. Knudsen, W. Meier, B. Preneel, V. Rijmen and S. Verdoolaege, "Analysis Methods for (Alleged) RC4," *Advances in Cryptology – ASIACRYPT'98*, LNCS 1514, K. Ohta and D. Pei (Eds.), pp. 327–341, Springer-Verlag, 1998.
12. I. Mantin, A. Shamir, "A Practical Attack on Broadcast RC4," *Fast Software Encryption – FSE 2001*, LNCS 2355, M. Matsui (Ed.), pp. 152–164, Springer-Verlag, 2001.
13. I. Mantin, "A Practical Attack on the Fixed RC4 in the WEP Mode." *Advances in Cryptology – ASIACRYPT 2005*, LNCS 3788, B. Roy (Ed.), pp. 395–411, Springer-Verlag, 2005.
14. I. Mantin, "Predicting and Distinguishing Attacks on RC4 Keystream Generator." *Advances in Cryptography – EUROCRYPT 2005*, LNCS 3494, R. Cramer (Ed.), pp. 491–506, Springer-Verlag, 2005.
15. I. Mironov, "(Not so) random shuffles of RC4," *Advances in Cryptology – CRYPTO'02*, LNCS 2442, M. Yung (Ed.), pp. 304–319, Springer-Verlag, 2002.

16. S. Mister and S. E. Tavares, “Cryptanalysis of RC4-like Ciphers,” *Selected Areas in Cryptography – SAC’98*, LNCS 1556, S. Tavares, H. Meijer (Eds.), pp. 131–143, Springer-Verlag, 1998.
17. S. Paul, B. Preneel, “A NewWeakness in the RC4 Keystream Generator and an Approach to Improve the Security of the Cipher,” *Fast Software Encryption – FSE 2004*, LNCS 3017, B. Roy (Ed.), pp. 245–259, Springer-Verlag, 2004.
18. S. Paul, B. Preneel, S. Sekar, “Distinguishing Attack on the Stream Cipher Py.” *Fast Software Encryption – FSE 2006*, LNCS 4047, M. J. Robshaw (Ed.), pp. 405–421, Spring-Verlag, 2006.
19. S. Paul, B. Preneel, “On the (In)security of Stream Ciphers Based on Arrays and Modular Addition.” *Advances in Cryptology – ASIACRYPT 2006*, LNCS 4284, K. Chen, and X. Lai (Eds.), pp. 69–83, Spring-Verlag, 2006.
20. H. Wu, B. Preneel, “Cryptanalysis of the Stream Cipher DECIM.” *Fast Software Encryption – FSE 2006*, LNCS 4047, M. J. Robshaw (ed.), pp. 30–40, Springer-Verlag, 2006.
21. H. Wu, B. Preneel, “Resynchronization Attacks on WG and LEX.” *Fast Software Encryption – FSE 2006*, LNCS 4047, M. J. Robshaw (ed.), pp. 422–432, Springer-Verlag, 2006.