Py (Roo, רו): A Fast and Secure Stream Cipher using Rolling Arrays

Eli Biham*   Jennifer Seberry†

April 29, 2005

Abstract

Py (pronounced Roo, a shorthand for Kangaroo) is a new stream cipher designed especially for the Ecrypt stream cipher contest. It is based on a new kind of primitive, which we call *Rolling Arrays*. It also uses various other ideas from many types of ciphers, including variable rotations and permutations. In some sense, this design is a kind of a new type of rotor machine, which is specially designed with operations that are very efficient in software. The allowed stream size is 2^{64} bytes in each stream (or 2^{40} in the smaller version Py6). The security claims of the cipher are that no key recovery attacks can be performed with complexity smaller than that of exhaustive search, and distinguishing attacks are also impractical with a similar complexity. The speed of the cipher is impressively fast, as it is more than 2.5 times faster than RC4 on a Pentium III (with less than 2.9 cycles/byte when implemented with the API of NESSIE and tested with the NESSIE software).

1 Introduction

Stream ciphers are widely used to encrypt sensitive data at fast speeds. Many stream ciphers that have been used in the past are not considered secure enough under current conditions. Moreover, the NESSIE project [15] did not select any stream cipher for its portfolio, as it was felt that none of the submissions was

*Computer Science department, Technion, Haifa 32000, Israel. `biham@cs.technion.ac.il`, `http://www.cs.technion.ac.il/~biham/`.

†Centre for Computer Security Research, SITACS, University of Wollongong, 2522, NSW, Australia. `j.seberry@uow.edu.au`, `http://www.uow.edu.au/~jennie/`.

strong enough. This feeling also applies to RC4, which is the de-facto standard for software applications (some of the properties and attacks on RC4 are given in [8,9,10,11,13,14,18,7,12]). These problems in stream ciphers directed many applications (such as cellular phones) to make a shift from stream ciphers to block ciphers. The feeling was that the current technology in stream ciphers in inferior to the technology of block ciphers in terms of security. In order to solve this problem, and create a portfolio of secure stream ciphers, the Ecrypt project had recently made a call for designs of new stream ciphers (see [6]). This paper is a response to this call.

In this paper we propose a new stream cipher, called Py (written in the Cyrillic alphabet, thus pronounced Roo). This cipher is very fast in software, and is claimed to be very secure. The main building block of Py is called *rolling arrays*. These arrays are rotated and updated over time in a way that allows both the data to be updated very quickly and a very efficient implementation. The design of Py is built upon a long history of experience, by many people, of the design of stream ciphers as well as of block ciphers. In particular, rolling arrays are related to rotor machines (e.g., Enigma), and Feedback Shift Registers (FSR's). Other parts of the design use other operations studied in the literature, including permutations (as in RC4), and variable rotations (as in RC5 [17]).

Py is a stream cipher designed for very fast and secure encryption of extremely long streams. It is intended for use with keys of up to 256 bits (32 bytes), and initial values (IVs) of up to 128 bits (16 bytes) (but it also allows longer keys of up to 256 bytes, and IV sizes up to 64 bytes; keys and IVs should be in multiples of a byte, and at least one byte in length). The streams generated for a given pair of key and IV is restricted to lengths of up to 2^{64} bytes, which is long enough for any practical purpose, but does not require the extra precautions that would be required for still longer streams. In particular, protection against distinguishing attacks on variants with longer streams may require additional operations to verify that the distribution of the values of the 32-bit output words is extremely close to uniform, and that these words are extremely independent.

The speed of Py is very impressive. According to the NESSIE performance document [16], RC4 spends about 7.3 cycles/byte for stream generation on a Pentium III, and many competitors are even slower. Py spends only about 2.85 cycles/byte on stream generation in its efficient implementation (when generating long streams, the most efficient case is when the code generates about 32K bytes at a time). This speed is about 2.5 times faster than RC4. This ratio is consistent also when running on other kinds of processors (in the range of 2.4–3). The Speed of RC4 and Py on several processors is given in Table 1 (the speed is measured by the NESSIE test suite under the NESSIE API for synchronous stream ciphers). On Pentium III, the key setup time is 2727 cycles, and the

	Pentium III	Alpha	Sun Sparc	Mac
Py (cycles/byte)	2.85	4.3	5.7	4.1
Py6 (cycles/byte)	2.82	4.6	6.0	4.3
RC4 (cycles/byte)	7.3	12.6	13.9	10.2

Table 1. Speed of Py and Py6 on Several Processors, Compared to RC4 (more information on the speed is given in Table 2).

IV setup time is 4092 cycles. The internal loop of Py under the NESSIE API has 32 machine instructions (when compiled by gcc), including the loop control operations, and it generates eight bytes. Note that under some non-standard API's, Py may be even slightly faster, due to the ability to remove some of the restrictions set by the API (such as extra registers needed to keep the location of the output, and copying of the arrays). Though it is very fast, Py is claimed to be very secure, and has none of the bad properties that have been already found for RC4.

A second variant of Py, called Py6, intended for shorter streams, is also presented. This variant has smaller rolling arrays, thus its key setup and IV setup are much faster than of Py, and take 796 and 1464 cycles, respectively (see Table 1). The stream generation speed is essentially similar to that of Py. Therefore, when encrypting stream of 100–150 bytes or more, the total key setup, IV setup and stream generation time of Py6 will already be faster than the corresponding times of other standard ciphers, such as AES and RC4. After about 500 bytes the time ratio is about 2 compared to AES and 1.6 compared to RC4.

The paper is organized as follows: In Section 2 we describe the idea of rolling arrays, the way they can be efficiently implemented, and basic facts on their usage in Py. In Section 3 we describe the key stream generation step of Py. In Section 4 we describe the key schedule, and in Section 5 we describe the IV schedule. In Section 6 we discuss the security of Py. In Section 7 a variant for shorter streams, called Py6, is presented. Finally, Section 8 summarizes the paper. In the appendices we give some more information for the Ecrypt contest, and answer some frequently asked questions.

2 Rolling Arrays

A *rolling array* is a vector whose units (which may be either bytes of words, or other memory units) are cyclically rotated every rotation step by one unit, i.e., it is similar to a rotor (e.g., of Enigma) that shifts it's entries by one location

Rolling Array with Swap
given k
swap($S[0]$, $S[k]$);
rotate(S)

Rolling Array with Addition
given v
$S[0]$+=v
rotate(S)

Figure 1. Examples of Rolling Arrays (the arrays are denoted by $S[0, \ldots, N-1]$), and rotate(S) is {tmp=$S[0]$, $S[0]$=$S[1]$, $S[1]$=$S[2]$, ..., $S[N-1]$=tmp}.

every step. Rolling arrays are formed with an additional fundamental operation, which is performed as an integral part of the rotation of the array. Two examples of such operations are a swap operation (as in RC4), and an addition operation. In the first case, where a swap operation is performed, for some entry k, the swap exchanges entry 0 and entry k, and then the rotation is performed. In the second case, where an addition is performed, a value is computed by the calling algorithm, and when the array is rotated, this value is added to the content of entry 0. The entry 0 becomes the last entry. These two examples are outlined in Figure 1.

A useful property of rolling arrays is that if you access the same entry in two consecutive steps, the contents of this entry are expected to be different (unlike in static arrays where they are expected to be the same). This property is very useful for increasing the speed of mixing the internal state of the cipher.

At first glance, a rolling array would be thought to be very inefficient in software, as all the entries of the array must be copied every step. However, careful observation reveals that if we define the array with extra entries at its end, and use a pointer to the beginning of the array, we can perform the above operations very efficiently, in about the same number of operations that are required to perform the fundamental operation without the rotation. The trick is to assign the new value of the entry $S[0]$ to $S[N]$ instead, where N is the size of the array ($S[N]$ is the first extra entry after the last entry), and then incrementing the pointer to the beginning of the array so that the new $S[0]$ to $S[255]$ is the old $S[1]$ to $S[256]$. This way, the array behaves like a caterpillar that crawls from one location to the next every step. Figure 2 shows this efficient implementation. Note that in contrast to other possible implementations that would keep the entries in the same location, but keep a running index of the beginning of the array, while computing the indices modulo the size of the array, we access the array without any modular operation on the indices, saving a large number of operations (and saving pipeline stalls due to the dependencies between the instructions).

In the cipher that we propose in this paper we use two rolling arrays, which

Rolling Array with Swap
given k
$S[N]$=$S[k]$; $S[k]$=$S[0]$
S++

Rolling Array with Addition
given v
$S[N]$=$S[0]$+v
S++

Figure 2. An Efficient Implementation of the Examples of Figure 1 (now the initial contents of the arrays are indexed by $S[0,\ldots,N-1]$, but the actual size of the array is larger, with entries $S[N]$, $S[N+1]$, $S[N+2]$, etc.).

affect each other. One is a permutation P of all 256 byte values, which is updated by one swap every step. The other is an array Y of 260 words of 32 bits, indexed in the range -3, ..., 256 (the first entry is indexed by -3 for simplicity of the indirect accesses, but of course all entries of the array are rotated, and the oldest entry, indexed -3, is updated). The update operation accesses other entries of Y, some of them are accessed directly (e.g., $Y[256]$), and others are accessed indirectly through a value in P (e.g., $Y[P[26]]$). The state is updated by swapping two entries of P (entry 0 and an entry that is selected using a value of Y), setting a new value for $Y[-3]$ in an invertible way (in order to ensure that the state space is not collapsed to a smaller subspace), and rotating the arrays.

The indirect accesses through P add a large amount of complexity to the mixing process, in a way that is highly non-linear and hard for a cryptanalyst to follow. A nice and useful property of indirect accesses through P is that when we access several entries in such a way (e.g., $Y[P[1]]$ and $Y[P[6]]$) in the same step, we are assured that the accessed entries in Y are different, as the content of P is a permutation. As the direct accesses to Y access entries outside the range 0, ..., 255, all the entries accessed in Y in each step are different.

3 The Cipher Py

The cipher Py maintains two rolling arrays P and Y and one word variable s. P is an array of 256 bytes that contains a permutation of all the values 0, ..., 255, and Y is an array of 260 32-bit words, indexed -3, ..., 256. In each step of the cipher the two arrays are rotated, and two output words (a total of eight bytes) are computed. The word s is updated by mixing two words of Y into it, where the two words are indirectly selected by indices from P, and then a variable rotation is performed, which rotates s by a number of bits which is taken from another entry of P.

The update of the rolling array (i.e., entry $Y[-3]$), and the computation

```
/* 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);
```

Figure 3. A (Non-Optimized) Step of Py (The state consists of a 32-bit variable s and two rolling arrays $P[0, \ldots, 255]$ and $Y[-3, \ldots, 256]$).

of the two output words are very similar: take a rotated value of s, XOR it with a value of Y (with a direct access to a fixed entry of Y), and add a value from Y which is accessed indirectly through a fixed entry of P. When updating $Y[-3]$, the direct access to Y accesses $Y[-3]$, and the result is then returned to it (or to $Y[257]$ in the efficient implementation) to ensure that the step is invertible. The set of ten fixed indices used to access Y and P ($\{-3, -1, 26, 72, 116, 153, 185, 208, 239, 256\}$) was selected as a difference free set [4] (i.e., all differences between pairs of indices are different) both modulo 260 and modulo 256.

A step of Py is outlined in Figure 3.

4 The Key Schedule

The key schedule of Py initializes the array Y (and only the array Y) from the key. In order to have a fast non-linear mixing, it uses an 8x8-bit S box, which appears in Figure 4.[1]

The key setup starts by initializing a 32-bit word s to depend on the key size, IV size, and the first and last bytes of the key, by setting one of its bytes

[1] The generation of this S box can be seen in the initialization function in the source code.

```
unsigned char internal_permutation[256]={
  B3, 99, 0D, 24, 41, 7F, E6, 21, 77, 2F, FB, A9, C0, E0, 37, 11,
  A4, 94, 38, 95, 97, 22, CD, 8F, D8, 43, B2, D7, 16, 20, C2, 55,
  C5, 8C, 96, 1A, FE, 4F, 9C, 02, 7C, BC, 04, E1, 46, 6A, 70, FF,
  EB, C7, 1C, 05, 19, 83, 13, 28, B9, C3, 12, 89, 98, 6F, 4D, B8,
  45, AA, F6, 9B, 36, 26, FD, 56, D4, 6E, 2B, 7D, AC, DB, CE, F0,
  92, D3, 5C, 09, E5, 71, 65, 3E, 9E, 60, 7E, 6B, BD, 23, CF, 2E,
  4B, F5, 62, 3D, 00, 9D, F1, 5F, 50, 61, DF, B4, A3, 17, ED, 88,
  30, F9, F3, 48, 06, EE, 39, 42, 64, 79, 73, 5E, 93, A0, 32, 8A,
  8D, B0, 0F, BA, 4C, BB, FC, B1, 85, 86, F2, A7, 15, 87, D1, 63,
  F4, B5, F7, 40, EC, DC, 58, 81, 31, 5A, C6, C4, 4A, 57, B7, EA,
  35, E7, 0E, A1, 51, 34, 0B, 9F, C9, E8, A6, DD, DA, 1F, F8, C8,
  07, 4E, A8, 0A, E4, 03, 75, 78, AB, E3, 2C, 0C, D0, 2A, 8E, 10,
  E9, CC, 54, AE, 3B, 6D, D6, 3A, 90, 47, 8B, 25, 84, 66, 01, A5,
  76, 3C, 49, 82, 1B, D5, D9, 6C, 7B, E2, 18, 91, 2D, 67, 7A, 68,
  80, 14, AF, 27, CA, 72, 9A, 5D, AD, EF, 5B, 69, DE, 3F, A2, 59,
  08, 74, CB, BF, 52, 53, 33, C1, 44, 1E, D2, 29, 1D, FA, BE, B6
};
```

Figure 4. The Internal Permutation of the Key Setup and IV Setup (in hex).

to be the value of the internal permutation applied on the key size (minus one, to ensure it is in the range 0–255). The next byte applies the permutation in the same way on the IV, but this time it is XORed with the previous computed byte before the application of the permutation. Then similarly it is applied to the first key byte (without minus one), and then to the last key byte. Now, after s is a mixture of the key size and IV size (and two key bytes), it is mixed with the key twice: for each key byte, the key byte is added to s, and the internal permutation is applied on the least significant byte of s. In the first pass, the result is XORed to s rotated by eight bits, to form the new s. In the second pass it is added to s rotated by eight bits, and the sum is XORed into s. All these operations generate a value of s that highly depends on the key and size parameters in a relatively small number of operations. This s is then used with the key to initialize Y, in the following way: for each entry in Y, take the next byte of the key, cyclically, starting from the first byte, and add it to s. Apply the internal permutation on the least significant byte of the result (thus, it highly depends on the key byte), and XOR the result of the permutation to s rotated by eight bits. This result is the next entry of Y as well as the new value of s. The key setup can be seen in Figure 5.

Note that the entries of the initial array Y are not fully independent, but this property will be canceled by the IV setup. Also note that in the pseudo code we mark the rolling arrays with parentheses (e.g., $Y(\cdot)$), while regular arrays

```
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;
  }

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;
  }
```

Figure 5. The Key Setup.

are marked with brackets (e.g., key[·]).

5 The IV Schedule

The IV schedule of Py is divided to two parts: the first initializes the P array (and an EIV array). The second part updates the P and Y array with the rolling array technique (with operations that are similar to — but are not the same as — a step of Py), and initializes s. The usage of a rolling array does not modify the original content of the Y array, so that it can be used again in further IV setups.

The first part starts by generating a permutation P, which serves as a base for later changes. It takes two bytes of the IV (unless the IV size is only one byte, in which case this byte is taken twice), and mixes them with some bytes that depend on the key (i.e., from array Y) in order to generate coefficients v and d (d must be odd) for an affine permutation $P'(i) = (di + v) \bmod 256$. The output of this affine permutation is then subjected to the internal permutation of Py, to become the initial value of the permutation P, before it is updated by the rolling array technique along with Y.

Then, another rolling array EIV is initialized, of the same size as the initial value, along with a word s. This array is used only in the IV setup, and is ignored later. The value of s is initialized by the concatenation of the four bytes v, d, $P(254)$ and $P(255)$, and then XORed by the sum of the first and last words of Y, in order to make it key dependent. Then, two very similar passes of update by the IV are performed. The idea is similar to the idea in the key setup: a byte of the IV is added to s, but in the IV setup a word from Y is also mixed to increase the dependence on the key. The least significant byte is subjected to the internal permutation, and the output becomes the value of the corresponding entry in the new EIV array. Finally, the output is also added to a rotated version of s to form the new s. In the second pass, the only two differences are that different words of Y are used (the last ones instead of the first ones), and that the output of the internal permutation is added to the already existing content of EIV, instead of being assigned into it.

The second part of the IV setup use the technique of rolling arrays to update P and Y, while still handling the array EIV. When performed in an array of $2 \cdot 260 = 520$ entries, the first 260 entries of Y remain dependent on the key only, and the remaining 260 form the result of the IV setup. This calculation runs the rolling array 260 steps: In each step EIV is updated first, then P is updated, and finally Y is updated. EIV is updated by XORing its oldest byte with the least significant byte of s, in order to generate the new byte. The result, which we call now $x0$, is then used as the index of the byte in P which is exchanged with its oldest byte (i.e., that becomes the new byte). Finally, the new word of Y is computed with a similar function as used three times in the main step of Py, where s is not rotated, the word of Y is the oldest word, and the indirect access through P is replaced by an indirect access using $x0$. The new Y is also set to be the new s.

After the Y and P arrays are set, it remains to set the value of s of the initial state. This value is the sum of the last s and of $Y(26) + Y(153) + Y(208)$ (the three indices are those that are used for the indirect accesses through P in the

```
/* 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);
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);
    u8 s0 = P(s&0xFF);
    EIV(i) += s0;
    s = ROTL32(s, 8) ^ (u32)s0;
  }
```

Figure 6. The IV Setup — Part 1 — Initialization of P and EIV (Initializes two rolling arrays: $P(0,\ldots,255)$, $EIV(0,\ldots,\text{ivsizeb}-1)$, and the value of the word s. The rolling array Y made in the key setup is also used here).

main step of Py). We observed that if the content of Y is fully zero and s is also zero, their content will remain zero for ever by the main step of Py. This case is very unexpected, and we actually belive it cannot happen. However, to ensure that it will never happen, we decided to ensure that s will never be zero at the initial state. We do this by setting it to keysize+ROTL32(ivsize,16)+87654321_x in case it is zero (the keysize and ivsize values are in units of bits).

The full details of the IV setup can be seen in Figures 6 and 7.

```
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;
```

Figure 7. The IV Setup — Part 2 — Updating the Rolling Arrays and s.

6 Security

6.1 Design Goals

The goals of this design is to be extremely fast, as well as very secure. It was designed to satisfy the following security properties:

1. For keys of up to 256 bits, and key streams of up to 2^{64} bytes, no attack can find the key, or expand a key stream, with a complexity less than of exhaustive search.
2. There are no distinguishing attacks that succeed given less than 2^{64} bytes of key stream, with a complexity less than of exhaustive search.

6.2 Tests

The cipher was tested in several ways. We first checked the statistical properties of the output stream to see whether it is possible to distinguish it from a random stream using some standard statistical tests. None of these identified any weakness. We actually claim that the output stream is highly uncorrelated, so that statistical tests should not succeed even when more extensive tests are made (including tests that are specially designed for Py).

6.3 Analysis

We then checked whether there may be a source of correlations between the outputs through the ways they are computed. It is easy to see that the new values in the array Y are unpredictable. The latest such updated value is used in computation of the first output word, together with the other terms of the formula of this output word, it is highly unlikely that this word is correlated with future first words. Similarly, it is highly unlikely that a correlation exists between second words and past second words (this time the fixed word location in Y is a very old one). The case of correlation between a first word to the second word in the same round shows that four other words, along with a different amount of rotation of s, differentiate between them, giving a large enough difference in the values. On the other hand, if we wait 257 steps, and try to correlate the first word at some time to the second word 257 steps afterwards, where the word $Y[256]$ that is mixed in the first word becomes the $Y[-1]$ word that is mixed in the other, in the hope that the rest of the words would also be similar, we find that the words that were used in the computation of the first word are not in the array any more, and the word s would certainly be unrelated to the content 257 steps earlier, so in such a case a correlation between the output words is very unlikely as well.

Another direction would be to guess the values of the two indirect accesses to Y in both outputs (one in each), subtract them from the outputs, and then derive the XOR of both direct accesses to Y in these formulae (and eliminate the values of s). However, since the values guessed are of indirect accesses to Y, the attacker would not then be able to use these values in other parts of an attack without having information on the content of P, so the attacker would only find 32 bits of the XOR of two words by guessing 64 bits. It would be easier to guess the 32 bits directly. In both cases the guesses cannot be verified, so no information can be derived concerning the internal state.

The design also ensures that the following cases would not help the attacker:

1. If during the computation step of Py, in any of the output words, both Y accesses (the direct access and the indirect access) did point to the same entry of Y, the security may have been reduced due to possible cancellations (probably in such a case there would be a distinguishing attack somewhere in the range of 2^{30}–2^{50} complexity). However, the direct and indirect accesses always select distinct entries (-3, -1 and 256 for the direct accesses, and entries in the range 0–255 for the indirect accesses).
2. If during the computation of s, both Y accesses pointed to the same entry, then s would not be updated (as zero would be added). However, the way

the indirect accesses are designed ensures that the two indices of Y are distinct.

3. If during the computation of s, both Y accesses pointed to the same two entries that they pointed to in the previous step, but in the reverse order, and if also the rotation did not rotate, then the same value of s would be used twice with a difference of two steps. The probability for that to happen is about $2^{-8} \cdot 2^{-8} \cdot 2^{-5} = 2^{-21}$. To make this property useful for the attacker, the indirect accesses in one of the output words should point to the same word of Y in those both steps, but even in that case, the direct accesses to Y ensure that the attacker does not see similar outputs that he can identify.

Finally, we remark that the average period of a cycle of the output of Py is expected to be in the order of $O(2^{|P,Y,s|})$, where $|P, Y, s| = 10400$ represents the size of the state in bits (taking into consideration that P is a permutation, we can compute the period to be about $O(2^{|Y,s|} \cdot 256!) = O(2^{8352} \cdot 256!)$).

6.4 Weak and Other Bad Keys

The designers made all efforts to ensure that no weak nor other kinds of bad keys or IVs exist, that no related-key attacks can be performed, and that no statistical information leaks through the key and IV schedules.

6.5 Other Kinds of Attacks

The designers believe that the standard known kinds of attacks against stream ciphers will not work against this design. In particular, correlation attacks and all kinds of attacks on (and bad properties of) RC4 do not work against Py.

7 Py6

We also present a second variant of Py, to which we call Py6. In this variant the size of the values of the permutation P is reduced to a smaller number of six bits, while reducing the sizes of P and Y to 64 and 68 entries, respectively. The word size of Y is left unchanged. The speed of this variant is essentially the same as the speed on Py (actually our measurements showed that it is even very slightly faster on Pentium III), however, the smaller size of the internal state allows a much faster key setup and IV setup, that are very attractive for encryption of short streams. This variant has smaller rolling arrays, thus its key setup and IV setup are much faster than of Py, and take 796 and 1464 cycles, respectively. The total number of cycles required by the key and IV setups is thus smaller than the key setup of RC4, and the stream generation is about 2.5 times faster than RC4.

In this variant the difference free set had to be replaced by a difference free set modulo 64 and modulo 68. However, there cannot be 10 numbers in such a difference free set, thus in this variant the difference free set contains only the six values that are used as indices to the array P. Also, in the generation of the permutation P the internal permutation cannot be used, thus it is removed from one location, and some rotations by eight bits are replaced by rotations by six bits in order to ensure full mixture of the data.

As the indices in this variant are shorter, we restrict the length of the generated streams to 2^{40}. Though stream that are shorter than 70–100 bytes would probably be faster to encrypt using the AES (or other block ciphers, due to the longer key setup and IV setup times), this variant will encrypt stream of 120–150 bytes faster than the AES, and streams of 500 bytes over twice faster than the AES, and over 40% faster than RC4 (including the key setup and IV setup times).

8 Summary

In this paper we presented the stream ciphers Py and Py6, and claimed that they are very secure as well as very fast. The cryptographic community is invited to study their security, and we will be very glad to hear about any security (or insecurity) results. The designers keep their rights on the design and it's name Py (Roo). However, no royalty will be necessary for use of Py, nor for using the submitted code. We hope that these new ciphers will be found useful, and that many people and applications will use it for good cause and for the benefit of human kind.

The speed of the stream generation, key setup and IV setup of Py and Py6 on several processors is given in Table 2.

Finally, we would also wish to note that the design of Py allows for many additional variants. In one of the additional attractive variants the array Y may have 64-bit entries. This variant may be very attractive on 64-bit processors, but to ensure security a small change in the step function may be needed (in particular, additional direct or indirect accesses to Y may be necessary to ensure

	Pentium III	Alpha	Sun Sparc	Mac
Py	2.85/2727/4091	4.31/2618/4053	5.78/2622/5492	4.1/2891/5003
Py6	2.82/796/1464	4.55/890/1253	5.99/864/1632	4.29/1001/1982
RC4	7.3/2715/-	12.6/3270/-	13.9/4234/-	10.15/2557/-

Table 2. Speeds of Py, Py6, and RC4 on Several Kinds of Processors (NESSIE API, entries in format stream generation in cycles/byte, key setup in cycles/key, and IV setup in cycles/IV).

the same security level).

Acknowledgments

The authors thank the URC Small Grant awarded to Dr Tianbing Xia which helped support the first author's visit to Australia. The authors would also like to thank Brendan McKay, Ian Wanless, and an anonymous friend for their input on combinatorial questions.

A Information for Ecrypt

1. Py is a synchronous stream cipher.
2. Generates key stream in units of eight bytes.
3. The key size is in units of bytes, from one byte to 256 bytes.
4. The IV size is in units of bytes, from one byte to 64 bytes.
5. The security claims are for keys with up to 256 bits (32 bytes) and IVs up to 128 bits (16 bytes).
6. Py is designed mainly to meet profile 1.
7. No hidden weaknesses were inserted by the designers, nor are the designers are aware of any weakness.
8. The internal state size is 10400 bits (1300 bytes).
9. For fast implementations the internal state size may need 2084 bytes (16656 bits). Together with keeping the result of the key setup in a rolling array, a total of 4164 bytes may be useful. Some implementation would keep even larger memory for the state.

10. The keys of Py6 are limited by 64 bytes. The internal state of Py6 is 2720 bits (340 bytes). Fast implementations will usually use 548 bytes for the state, and together with the result of the key setup the state will usually take 1092 bytes.
11. The length of the key streams generated by Py is limited by 2^{64} bytes. For Py6 the streams are limited by 2^{40} bytes.
12. The designers keep their rights on the design and it's name Py (Roo). However, no royalty will be necessary for use of Py, nor for using the submitted code.
13. No patents are set by the authors to protect Py, and the authors are not aware of any other patents that cover Py.

B Questions and Answers

1. Why Py, and why isn't it spelled Roo?

   Our tradition has been that we call new cryptographic functions after animals, e.g., Tiger [1], Bear and Lion [2], and Serpent [3]. In this case, this stream cipher was designed in Australia to jump around and be very fast. Also the byte array can be viewed in some implementations as residing inside a pouch of the word array, so the name Kangaroo is just obvious. But we thought that a shorter name would be preferred, so we stuck to the suffix Roo.

   Then, we wished to use this opportunity to make sure people learn to read Hebrew, so we wished to spell Roo as "רו" (read it from right to left), but it appeared that this may be too difficult for some people to typeset. So we decided to use the Cyrillic alphabet instead, which is much easier for most people to use and learn. Still, for most people the Cyrillic alphabet itself looks like a good cipher that cannot be read, just like Py.

2. How is that pronounced?

   If you know Hebrew, just pronounce it as "רו". If you know Russian or Greek, then you know how to do it anyway. Otherwise, you should pronounce it like the suffix of Hebrew or the suffix of Kangaroo. As long as you make it sound different from "π" we will not complain.

3. Why did you choose this name?

   Because we noticed that some people have various difficulties pronouncing the names "אלי" (i.e., Eli) and "Seberry". (There are three messages in this answer.)

4. Can't you give it another name?

   Yes. We considered the name "Rijndael", but it was already used [5]. A bear who wishes to remain anonymous proposed the name "POOH".

References

[1] Ross Anderson and Eli Biham, "Tiger: a fast new hash function", *Proceedings of Fast Software Encryption*, Lecture Notes in Computer Science, Vol. 1039, pp. 89–97, Springer-Verlag, Berlin, 1996.

[2] Ross Anderson and Eli Biham, "Two practical and provably secure block ciphers: BEAR and LION", *Proceedings of Fast Software Encryption*, Cambridge, Lecture Notes in Computer Science, pp. 113–120, Springer-Verlag, Berlin, 1996.

[3] Ross Anderson, Eli Biham and Lars R. Knudsen, *Serpent: A Proposal for the Advanced Encryption Standard*, NIST AES Proposal, 1998.

[4] L. D. Baumert, *Cyclic Difference Sets*, Lecture Notes in Mathematics, Vol. 182, Springer-Verlag, Berlin-Heidelberg-New York, 1971.

[5] Joan Daemen and Vincent Rijmen *The design of Rijndael: AES — the Advanced Encryption Standard*, Springer-Verlag, Berlin, 2002.

[6] Ecrypt, `http://www.ecrypt.eu.org/`.

[7] Hal Finney, *An RC4 Cycle that Can't Happen*, Usenet newsgroup sci.crypt, September 1994.

[8] S. R. Fluhrer, I. Mantin, and A. Shamir, "Weaknesses in the key scheduling algorithm of RC4." in *Proceedings of Selected Areas in Cryptography – SAC'01* (S. Vaudenay and A. M. Youssef, eds.), Lecture Notes in Computer Science, Vol. 2259, pp. 1–24, Springer-Verlag, Berlin, 2001.

[9] S. R. Fluhrer and D. A. McGrew, "Statistical analysis of the alleged RC4 stream cipher." in *Proceedings of Fast Software Encryption – FSE'00* (B. Schneier, ed.), Lecture Notes in Computer Science, Vol. 1978, pp. 19–30, Springer-Verlag, Berlin, 2000.

[10] J.D. Golic, "Linear statistical weakness of alleged RC4 keystream generator." in *Proceedings of Eurocrypt'97* (W. Fumy, ed.), Lecture Notes in Computer Science, Vol. 1233, pp. 226–238, Springer-Verlag, Berlin, 1997.

[11] L.R. Knudsen, W. Meier, B. Preneel, V. Rijmen, and S. Verdoolaege, "Analysis methods for (alleged) RC4." in *Proceedings of Asiacrypt'98* (K. Ohta and D. Pei, eds.), Lecture Notes in Computer Science, Vol. 1514, pp. 327–341, Springer-Verlag, Berlin, 1998.

[12] I. Mantin, *Analysis of the Stream Cipher RC4*, M.Sc. thesis, The Weizmann Institute of Science, 2001. Available at `http://www.wisdom.weizmann.ac.il/~itsik/RC4/rc4.html`.

[13] I. Mantin and A. Shamir, "A practical attack on broadcast RC4." in *Proceedings of Fast Software Encryption – FSE'01* (M. Matsui, ed.), Lecture Notes in Computer Science, Vol. 2355, pp. 152–164, Springer-Verlag, Berlin, 2001.

[14] I. Mironov, "(Not so) random shuffles of RC4." in *Proceedings of Crypto'02* (M. Yung, ed.), Lecture Notes in Computer Science, Vol. 2442, pp. 304–319, Springer-Verlag, Berlin, 2002.

[15] NESSIE: New European Schemes for Signature, Integrity and Encryption, `http://www.nessie.eu.org/nessie/`.

[16] NESSIE partners, *Performance of Optimized Implementations of the NESSIE Primitives*, Technical report, NES/DOC/TEC/WP6/D21/2, 2003. Available in `http://www.nessie.eu.org/nessie/`.

[17] Ronald L. Rivest, "The RC5 encryption algorithm", *Proceedings of Fast Software Encryption*, Leuven, Lecture Notes in Computer Science, pp. 86–96, Springer-Verlag, Berlin, 1994.

[18] Souradyuti Paul and Bart Preneel, "A new weakness in the RC4 keystream generator and an approach to improve the security of the cipher.", *Proceedings of Fast Software Encryption*, 11th International Workshop, FSE 2004, Delhi, India, Lecture Notes in Computer Science, Vol. 3017, pp. 245–259, Springer-Verlag, Berlin, 2004.