ABC: A New Fast Flexible Stream Cipher

Specification

Vladimir Anashin[*], Andrey Bogdanov[†], Ilya Kizhvatov
Faculty of Information Security
Institute for Information Sciences and Security Technologies
Russian State University for the Humanities
{anashin, bogdanov, kizhvatov}@rsuh.ru
Sandeep Kumar [‡]
Communication Security Group (COSY)
Department of Electrical Engineering and Information Sciences
Ruhr-University Bochum
kumar@crypto.rub.de

Introduction

ABC is a synchronous stream cipher optimized for software applications. Its design offers large flexibility concerning key material usage and memory consumption. Here we present a version of ABC with a 128-bit key and a 128-bit IV, a 32-bit non-linear filter, flexible key expansion, flexible and fast IV setup procedures, and variable memory requirements.

[*] Corresponding author
[†] Partially supported by the Institute for Experimental Mathematics, University of Duisburg-Essen, Germany
[‡] Implementation for 8-bit processor

A new approach to the stream cipher design has been used which results in a cipher based upon key- and clock-dependent state transition and non-linear filter functions. Our techniques guarantee the period of $2^{32} \cdot (2^{63} - 1)$ words (the longest of possible), uniform distribution, and high linear complexity of the key stream of ABC.

The ABC stream cipher offers a security level of 2^{128}. No hidden weaknesses have been incorporated in the design of ABC. We have found no weaknesses of the design with respect to standard cryptanalytic attacks.

ABC can be efficiently implemented in software. Our C implementation provides the throughput of about 7 Gbps. The flexibility property results in the possibility of its efficient application on almost every platform by choosing proper implementation parameters.

1 Variables and basic operations

In the description of cryptographic primitives and in the specification of ABC we rest upon some variables which change at each step of computations:

x is a 32-bit integer value and can be represented in different ways:

$$x = (x_{31}, \dots, x_0) = \sum_{i=0}^{31} x_i 2^i \in \mathbb{Z}/2^{32}\mathbb{Z},\ \ x_i \in \{0,1\},\ \ i = 0, \dots, 31;$$

$$x = (\hat{x}_{t-1}, \dots, \hat{x}_0),\ \ \hat{x}_i \in \mathbb{Z}/2^w\mathbb{Z},\ \ i = 0, \dots, t-1,\ w \in \mathbb{Z},\ w \mid 32,\ t = 32/w \in \mathbb{Z};$$

$$x \in \mathbb{V}_{32} = \mathrm{GF}(2)^{32};$$

y is a 32-bit integer value, one way of representing it being used only:

$$y = (y_{31}, \dots, y_0) = \sum_{i=0}^{31} y_i 2^i \in \mathbb{Z}/2^{32}\mathbb{Z},\ \ y_i \in \{0,1\},\ \ i = 0, \dots, 31;$$

z is a 64-bit integer value and allows several equivalent representations too:

$$z = (z_{63}, \dots, z_0) = \sum_{i=0}^{63} z_i 2^i \in \mathbb{Z}/2^{64}\mathbb{Z},\ \ z_i \in \{0,1\},\ \ i = 0, \dots, 63;$$

$$z \in \mathbb{V}_{64} = \mathrm{GF}(2)^{64};$$

$$z = (\bar{z}_1, \bar{z}_0) \in (\mathbb{Z}/2^{32}\mathbb{Z})^2,\quad \bar{z}_1, \bar{z}_0 \in \mathbb{Z}/2^{32}\mathbb{Z}.$$

x and z represent the current internal state of the cipher. The initial values of x and z are defined in the course of the initialization stage. y denotes the 32-bit output of the key stream generator.

Moreover, ABC uses some further variables that are calculated from the key and initial value at the initialization stage by applying a special key expansion routine:

$e,\ e_0, \dots, e_{31} \in \mathbb{Z}/2^{32}\mathbb{Z}$ are 32-bit integer values;

$d_0 = (d_{0,31}, \dots, d_{0,0}),\ d_1 = (d_{1,31}, \dots, d_{1,0}) \in \mathbb{Z}/2^{32}\mathbb{Z}$ are 32-bit integer values.

Having been defined once the variables d_0, d_1, e and $\{e_i\}_{i=0}^{31}$ remain unchanged during the whole subsequent encryption stage as distinct from x and z.

In the description of cryptographic primitives we will also require a 32-bit integer $\zeta \in \mathbb{Z}/2^{32}\mathbb{Z}$ for storing intermediate computation results.

To describe some optimization techniques we will need an auxiliary w-bit integer variable $j \in \mathbb{Z}/2^w\mathbb{Z}$:

$$j = (j_{w-1}, \ldots, j_0) = \textstyle\sum_{i=0}^{w-1} j_i 2^i \in \mathbb{Z}/2^w\mathbb{Z},\ j_i \in \{0,1\},\ i = 0, \ldots, w-1.$$

Finally, in the description of operations below we will require a pair of 32-bit integer variables

$$a = (a_{31}, \ldots, a_0),\ b = (b_{31}, \ldots, b_0) \in \mathbb{Z}/2^{32}\mathbb{Z},\ a_i, b_i \in \{0,1\},\ i = 0, \ldots, 31,$$

for representing operands of some operators.

The ABC cipher requires the following operations for its specification:

Addition modulo 2^{32}, $+$, represents an ordinary arithmetic addition of 2 operands in $\mathbb{Z}/2^{32}\mathbb{Z}$ as 32-bit integers;

Bitwise addition modulo 2, XOR, defines a binary addition of 2 operands in $\mathbb{V}_{32}$, or bitwise exclusive 'OR' of 2 32-bit integer operands as follows:

$$a \,\mathsf{XOR}\, b = (a_{31} \oplus b_{31}, \ldots, a_0 \oplus b_0),$$

where

$$a_i \oplus b_i = \begin{cases} 0, & \text{if } a_i = b_i, \\ 1, & \text{otherwise;} \end{cases}$$

Bitwise multiplication modulo 2, AND, defines a bitwise 'AND' of 2 32-bit integer operands as follows:

$$a \,\mathsf{AND}\, b = (a_{31} \wedge b_{31}, \ldots, a_0 \wedge b_0),$$

where

$$a_i \wedge b_i = \begin{cases} 1, & \text{if } a_i = b_i = 1, \\ 0, & \text{otherwise;} \end{cases}$$

Bitwise disjunction, OR, defines a bitwise inclusive 'OR' of 2 32-bit integer operands as follows:

$$a \,\mathsf{OR}\, b = (a_{31} \vee b_{31}, \ldots, a_0 \vee b_0),$$

where

$$a_i \vee b_i = \begin{cases} 0, & \text{if } a_i = b_i = 0, \\ 1, & \text{otherwise;} \end{cases}$$

The i-th bit selection, $\delta_i(\cdot)$, determines the i-th bit of a 32- or 64-bit integer number and can be described in the following way as applied to respectively x, z, d_1 and j:

$$\delta_i : \mathbb{Z}/2^{32}\mathbb{Z} \to \{0,1\},\ \delta_i(x) = x_i,\ \ i = 0, \dots, 31,$$
$$\delta_i : \mathbb{Z}/2^{64}\mathbb{Z} \to \{0,1\},\ \delta_i(z) = z_i,\ \ i = 0, \dots, 63,$$
$$\delta_i : \mathbb{Z}/2^{32}\mathbb{Z} \to \{0,1\},\ \delta_i(d_1) = d_{1i},\ \ i = 0, \dots, 31,$$
$$\delta_i : \mathbb{Z}/2^{w}\mathbb{Z} \to \{0,1\},\ \delta_i(j) = j_i,\ \ i = 0, \dots, w-1;$$

Bit substring selection, $[\cdot]_u^v$, denotes a substring of bits in positions from u to v, $u, v \in \mathbb{Z}/2^5\mathbb{Z}$, in the binary expansion of a 32-bit integer number and is defined as follows:

$$[a]_u^v = (\delta_v(a), \dots, \delta_u(a)) = (a_v, \dots, a_u),\ \ u < v,$$

for example,

$$\begin{array}{lcl} a & = & 00000000000000010000000000111010_2, \\ [a]_1^{16} & = & 1000000000011101_2; \end{array}$$

Right shift, $\cdot \gg c$, denotes right zero-fill bit shift of binary expansion of a 32-bit integer number by c bits, $c \in \mathbb{Z}/2^5\mathbb{Z}$, and can be described as follows:

$$a \gg c = (\underbrace{0, \dots, 0}_{c}, a_{31}, \dots, a_c);$$

Left shift, $\cdot \ll c$, denotes left zero-fill bit shift of binary expansion of a 32-bit integer number by c bits, $c \in \mathbb{Z}/2^5\mathbb{Z}$, and can be described as follows:

$$a \ll c = (a_{31-c}, \dots, a_0, \underbrace{0, \dots, 0}_{c});$$

Right rotation, $\cdot \ggg c$, denotes right bit-wise rotation of binary expansion of a 32-bit integer number by c bits, $c \in \mathbb{Z}/2^5\mathbb{Z}$, and can be described as follows:

$$a \ggg c = (a_{c-1}, \dots, a_0, a_{31}, \dots, a_c).$$

2 Primitives

ABC uses 3 main primitives (A, B and C respectively):

A: $\mathbb{Z}/2^{64}\mathbb{Z} \to \mathbb{Z}/2^{64}\mathbb{Z}$ is a linear feedback shift register of length 64 (LFSR), z representing its state;

B: $\mathbb{Z}/2^{32}\mathbb{Z} \to \mathbb{Z}/2^{32}\mathbb{Z}$ represents a single cycle mapping based on arithmetical addition in $\mathbb{Z}/2^{32}\mathbb{Z}$, bitwise addition modulo 2 (XOR), and left bit shift ($\ll$), transforming x;

C: $\mathbb{Z}/2^{32}\mathbb{Z} \to \mathbb{Z}/2^{32}\mathbb{Z}$ specifies a filter function based on lookup tables, arithmetical addition in $\mathbb{Z}/2^{32}\mathbb{Z}$ and right bit-wise rotation ($\ggg$), assuming x as argument and delivering y.

A: Linear feedback shift register, counter

A is a linear transformation of the vector space $\mathbb{V}_{64} = \mathrm{GF}(2)^{64}$, $z = A(z)$, and is defined by a LFSR. Since bit operations are relatively slow on general purpose processors, we use a word oriented representation of the LFSR. It is of length 64 and its characteristic polynomial is $\phi(\theta) = \psi(\theta)\theta$, where $\psi(\theta) = \theta^{63} + \theta^{31} + 1$ is primitive. Moreover, as in [9] we produce the next 32 bits at once, which is done as follows:

$$\begin{aligned} &\zeta \leftarrow \bar{z}_1 \,\mathsf{XOR}(\bar{z}_0 \gg 1)\,\mathsf{XOR}(\bar{z}_1 \ll 31) \bmod 2^{32}, \\ &\bar{z}_0 \leftarrow \bar{z}_1, \\ &\bar{z}_1 \leftarrow \zeta. \end{aligned} \tag{1}$$

It is important to stress here that the above representation is just another (word-oriented) representation of the 63-bit LFSR with primitive polynomial $\psi(\theta)$; we obtain two outputs from this LFSR, the first one for the state transition procedure and the second one for updating the output function. Thus, the cycle length of this LFSR is $2^{63} - 1$, and *not* $2^{64} - 1$.

This also leads to the fact that the cycle length becomes 1 in case the initial state $z = (\bar{z}_1, \bar{z}_0)$ of A is either $(0, 0)$ or $(0, 1)$. We eliminate this danger by forcing $\delta_1(z)$ to 1 in ABC key setup and IV setup procedures.

B: Single cycle function, state transition

The single cycle function B used in the ABC cipher can be specified through the following equation:

$$B(x) = d_0 + 5(x \;\;\mathsf{XOR}\;\; d_1) \bmod 2^{32}, \tag{2}$$

where d_0, $d_1 \in \mathbb{Z}/2^{32}\mathbb{Z}$, $d_0 \equiv 1 \pmod 2$, $d_1 \equiv 0 \pmod 4$. *These restrictions guarantee that B is a single cycle map modulo 2^{32}, see Proposition 1 in Appendix A.*

Since $5_{10} = 101_2$ $B(x)$ can be computed via 1 left zero-fill shift and 1 extra addition modulo 2^{32}. The form of this function is determined at the initialization stage (through setting the 61 bits of d_0 and d_1 in a key- and IV-dependent manner).

C: Filter function, output

Here we give two representations of the ABC filter C. At first the coordinate representation is discussed. Then we show how to speed up the computation of the function through the introduction of several tables of variable length.

Let $S : \mathbb{Z}/2^{32}\mathbb{Z} \to \mathbb{Z}/2^{32}\mathbb{Z}$ be a mapping defined by

$$S(x) = e + \sum_{i=0}^{31} e_i \delta_i(x) \bmod 2^{32}, \tag{3}$$

where $e_{31} \equiv 2^{16} \pmod{2^{17}}$. The filter function C takes x as argument and produces y in the following way:

$$\begin{aligned} \zeta &= S(x), \\ y &= \zeta \ggg 16. \end{aligned} \tag{4}$$

The function C is a highly non-linear mapping and is the main security block of ABC. Its cryptographical properties are studied in detail in Appendix A. Rotation by 16 bits was chosen to enable fast implementation by byte swapping on 8-bit and 16-bit processors.

Actually, to compute S it is not necessary to read x bitwise at each iteration. Instead of this we can use a window technique. To give another representation of S consider a positive integer value $w \neq 1$ or 32, $w \mid 32$, i.e. $w = 2, 4, 8$ or 16. Now divide the bit representation of x into $t = 32/w$ windows, each of length w bits :

$$x = (\hat{x}_{t-1}, \dots, \hat{x}_0), \ \hat{x}_i \in \mathbb{Z}/2^w\mathbb{Z}, \ i = 0, \dots, t-1. \tag{5}$$

Let $T_0, \dots, T_{t-1}$ be tables, each holding 2^w 32-bit elements. These tables can be precomputed in the following way:

$$T_i[j] = \sum_{l=0}^{w-1} \delta_l(j) \cdot e_{w \cdot i + l} \bmod 2^{32}, \quad j = 0, \dots, 2^w - 1, \tag{6}$$

for $i = 1, \dots, t-1$ and

$$T_0[j] = e + \sum_{l=0}^{w-1} \delta_l(j) \cdot e_l \bmod 2^{32}, \quad j = 0, \dots, 2^w - 1, \tag{7}$$

for $i = 0$. Then we can rewrite S in the corresponding way:

$$S(x) = \sum_{s=0}^{t-1} T_s[\hat{x}_s], \quad \hat{x}_s \in \mathbb{Z}/2^w\mathbb{Z}, s = 0, \dots, t-1. \tag{8}$$

Using this window optimization method it is possible to vary memory consumption. We have computed the respective values and give them in Table

1. Depending on the available hardware or software resources, users can select the optimal value for their specific purposes. More generally, the bit lengths of windows do not need to be equal. For example, one can use three windows of bit lengths 12, 12 and 8 bit respectively. This approach makes memory consumption much more flexible. Additionally it is possible to make no use of this optimization method in case of strictly limited memory resources. However, this approach is not recommended in applications subject to side-channel attacks. The last case will be referred to as $w = 1$; it requires 33 32-bit values $e, \{e_i\}_{i=0}^{31}$ and therefore 132 byte of memory, which is the minimum value for ABC.

Table 1: Memory consumption and window bit length

w = window bit length	t = number of tables	memory, $4 \cdot t \cdot 2^w$ byte
2	16	256
4	8	512
8	4	4096
16	2	524288

3 Key stream generator

Now we describe the kernel of the ABC cipher. The key stream generation routine of ABC involves the primitives described in Section 2 and consists of 3 steps.

ABC KEY STREAM GENERATOR

INPUT: $z \in \mathbb{Z}/2^{64}\mathbb{Z}$, $x \in \mathbb{Z}/2^{32}\mathbb{Z}$

$$z \leftarrow A(z)$$
$$x \leftarrow \bar{z}_1 + B(x) \bmod 2^{32}$$
$$y \leftarrow \bar{z}_0 + C(x) \bmod 2^{32}$$

OUTPUT: $z \in \mathbb{Z}/2^{64}\mathbb{Z}$, $x \in \mathbb{Z}/2^{32}\mathbb{Z}$, $y \in \mathbb{Z}/2^{32}\mathbb{Z}$

This routine generates the next 32 key stream bits, y. The newly computed values x and z form the input of the next iteration of the key stream generation routine. Here A is a counter which makes the state transition function and the output function clock-dependent (see Figure 1).

It is also worth noticing that the sequence of states $(x; z)$ of the ABC key stream generator forms a cycle of length $2^{32} \cdot (2^{63} - 1)$. The cycle is totally determined by the LFSR A and function B: Pairwise distinct functions B (that correspond to distinct pairs of coefficients d_0, d_1) determine pairwise distinct cycles. A pair of initial states z (of the LFSR A) and x (of the function B) determine a unique initial position on the cycle.

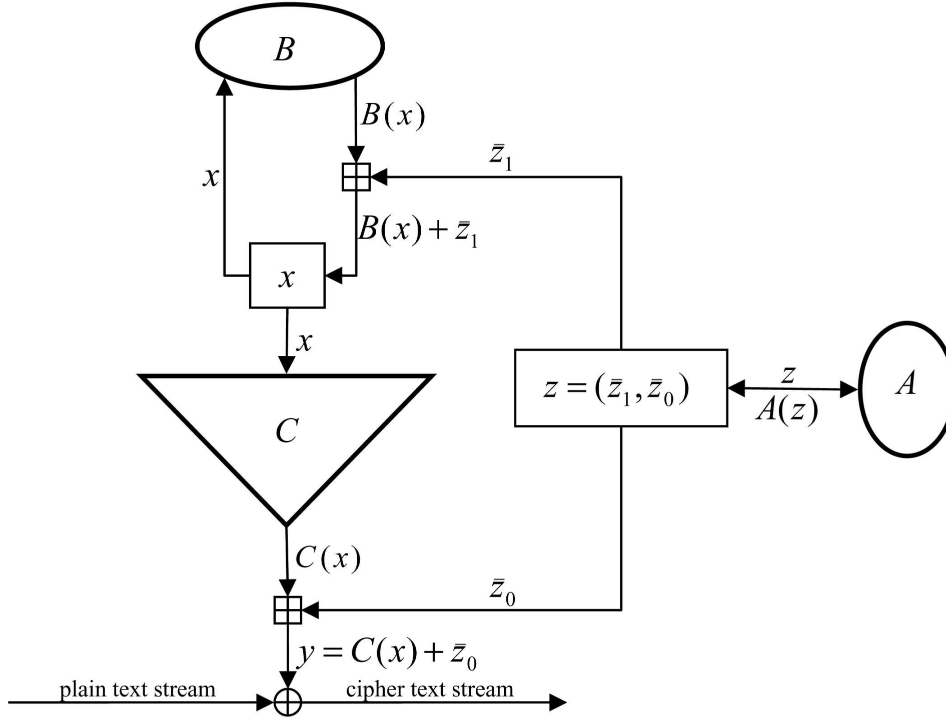

Figure 1: ABC key stream generator. $\bigoplus$ denotes bitwise addition modulo 2 (XOR). $+$ and $\boxplus$ represent arithmetical addition modulo 2^{32}.

3.1 Properties of the key stream

The following properties of the key stream produced by ABC key stream generator *are proved*:

- The length P of the shortest period of the key stream sequence of 32-bit words is $2^{32} \cdot (2^{63} - 1)$
- The distribution of the key stream sequence of 32-bit words is uniform; that is, for each 32-bit word a the number $\mu(a)$ of occurrences of a at the period of the key stream satisfies the following inequality:

$$\left| \frac{\mu(a)}{P} - \frac{1}{2^{32}} \right| < \frac{1}{\sqrt{P}}$$

- The linear complexity of the key stream bit sequence exceeds 2^{31}

Proofs are given in the Appendix (see Sections A and B). Note that it could be proved that linear complexity λ of the key stream satisfies the inequality $2^{31} \cdot (2^{63} - 1) + 1 \geq \lambda \geq 2^{31} + 1$. However, a reduced model of the

cipher (with reduced bit lengths of variables) shows that the lower bound is too pessimistic: In all cases we obtained values of the linear complexity close to the upper bound.

Note also that for a truly random sequence of length P of 32-bit words with probability $> 1 - \frac{1}{2^{32}}$ one has $\left|\frac{\mu(a)}{P} - \frac{1}{2^{32}}\right| < \frac{1}{\sqrt{P}}$.

4 Key expansion and nonce setup

As we already mentioned at the beginning, the ABC cipher offers large variability concerning key management. This statement applies also to the key expansion procedure. After a short preliminary remark we present several extreme cases and some different variants of initialization routine for the case of a 128-bit key k and a 128-bit initial value iv. Although it is up to the user which setup procedure to employ, we define some basic routine which seems to be optimal in general. It can be not the case for all special cases.

For its operation the ABC cipher requires:

- 2 32-bit integer values for the initial fill of $z = (\bar{z}_1, \bar{z}_0)$,
- 1 32-bit integer value for the initial fill of x,
- 2 32-bit integer values d_0, d_1 for the coefficients of B,
- 33 32-bit integer values e, $\{e_i\}_{i=0}^{31}$ for the coefficients of C.

Altogether we have to generate 38 32-bit integer values, which makes 1195 bits of data concerning the restrictions on $\bar{z}_0$, d_0, d_1 and e_{31} from (1), (2) and (3). Note that some of these bit values can be fixed and be therefore a part of the cipher itself. Then the user has to generate the rest of the values only. It is important to stress here that strictly speaking all the distinct key initialization procedures defined below lead to *distinct* ciphers. They are only given here in order to demonstrate the flexibility of ABC with respect to key material usage and to provide the designer of a concrete cryptographical system with a (relatively loose) solution framework. We hope that people interested in secure and efficient end product can use the ideas stated here to adjust the ABC stream cipher to their specific applications.

Extreme cases

1. Let the user disbelieve all the variants of key expansion routine suggested below. Then a 1195-bit key k' can be used and all the coefficients can be filled with the subkeys directly. In this case we have:

$$
\begin{aligned}
&k' \in \mathbb{Z}/2^{1195}\mathbb{Z},\\
&k' = (\bar{k}'_{37}, ..., \bar{k}'_0) \in (\mathbb{Z}/2^{32}\mathbb{Z})^{32} \times (\mathbb{Z}/2^{15}\mathbb{Z}) \times (\mathbb{Z}/2^{30}\mathbb{Z})^2 \times (\mathbb{Z}/2^{32}\mathbb{Z})^3,\\
&\bar{k}'_i \in \mathbb{Z}/2^{32}\mathbb{Z},\ 0 \le i \le 37,\ i \ne 32, 33, 34,\\
&\bar{k}'_{32} \in \mathbb{Z}/2^{15}\mathbb{Z},\\
&\bar{k}'_{33}, \bar{k}'_{34} \in \mathbb{Z}/2^{30}\mathbb{Z}.
\end{aligned} \tag{9}
$$

So the key initialization routine can be realized in the following way:

$$
\begin{array}{lcl}
e & \leftarrow & \bar{k}'_0;\\
e_i & \leftarrow & \bar{k}'_{1+i},\ i = 0, \dots, 30;\\
e_{31} & \leftarrow & (\bar{k}'_{32} \lll 17) \text{ OR } 2^{16};\\
d_0 & \leftarrow & (\bar{k}'_{33} \lll 2) \text{ OR } 1;\\
\delta_1(d_0) & \leftarrow & \delta_1(\bar{k}'_{36});\\
d_1 & \leftarrow & \bar{k}'_{34} \lll 2;\\
x & \leftarrow & \bar{k}'_{35};\\
\bar{z}_0 & \leftarrow & \bar{k}'_{36} \text{ OR } 2;\\
\bar{z}_1 & \leftarrow & \bar{k}'_{37}.
\end{array} \tag{10}
$$

It is worth putting here that there exist some attacks that require less than 2^{1195} operations in this case.

2. Alternatively, the user can define his own key expansion routine, say $G(k)$, where k is the primary key of some bit length m and

$$
G : \mathbb{Z}/2^m\mathbb{Z} \to \mathbb{Z}/2^{1195}\mathbb{Z}. \tag{11}
$$

Then by setting $k' \leftarrow G(k)$ we come to the previous case and proceed with (10). $G(k)$ has to satisfy the properties of a hash function, that is,

(a) preimage resistance,

(b) second preimage resistance,

(c) collision resistance,

(d) strong propagation of changes in preimage k onto the image $G(k)$.

A review of the first three properties can be found in [18].

3. Let the user have extremely restricted computational resources at his disposal for some reasons and wish to significantly speed up the initialization phase. Then it can be advisable he fix $e, \{e_i\}_{i=0}^{31}$ and some bits of d_0, d_1, taking the values $\mathfrak{e}, \{\mathfrak{e}_i\}_{i=0}^{31}, \mathfrak{d}_0, \mathfrak{d}_1$ with some statistical requirements fulfilled (a set of such values can be found in Table 2):

$$
\begin{array}{lcl}
e & = & \mathfrak{e},\\
e_i & = & \mathfrak{e}_i, \quad i = 0, \dots, 31,\\
d_0 & = & \mathfrak{d}_0,\\
d_1 & = & \mathfrak{d}_1,
\end{array}
$$

and fill the rest of the cipher state only. Note that the fixed coefficients $\mathfrak{d}_0, \mathfrak{d}_1$ and $\mathfrak{e}_{31}$ satisfy the restrictions imposed by (2) and (3). Here we state no security properties of this type of key initialization and declaim no security level of this solution. The properties of this key initialization routine are to be studied in more detail.

Table 2: Coefficients for key setup routines, in hexadecimal notation

$\mathfrak{d}_0 =$	0C376D75	$\mathfrak{e}_9 =$	EAA84751	$\mathfrak{e}_{21} =$	923DDD55
$\mathfrak{d}_1 =$	BBB5B0B4	$\mathfrak{e}_{10} =$	77F1CE29	$\mathfrak{e}_{22} =$	A6461E22
$\mathfrak{e} =$	A883B17D	$\mathfrak{e}_{11} =$	EB94AD46	$\mathfrak{e}_{23} =$	CBF825B8
$\mathfrak{e}_0 =$	8BBC7B0A	$\mathfrak{e}_{12} =$	FFD624D0	$\mathfrak{e}_{24} =$	1139265E
$\mathfrak{e}_1 =$	E774A906	$\mathfrak{e}_{13} =$	89581695	$\mathfrak{e}_{25} =$	B9CF4535
$\mathfrak{e}_2 =$	13040EC0	$\mathfrak{e}_{14} =$	F0BBFBD3	$\mathfrak{e}_{26} =$	E7C87F14
$\mathfrak{e}_3 =$	EA149BD0	$\mathfrak{e}_{15} =$	83404B20	$\mathfrak{e}_{27} =$	F4F855D3
$\mathfrak{e}_4 =$	32E3281D	$\mathfrak{e}_{16} =$	9E66ABEA	$\mathfrak{e}_{28} =$	7C77F154
$\mathfrak{e}_5 =$	38C15589	$\mathfrak{e}_{17} =$	798CE417	$\mathfrak{e}_{29} =$	46C0F13C
$\mathfrak{e}_6 =$	BDC92EA9	$\mathfrak{e}_{18} =$	8D1ADFB3	$\mathfrak{e}_{30} =$	2D1229E6
$\mathfrak{e}_7 =$	6B587BA0	$\mathfrak{e}_{19} =$	B8C6BF9F	$\mathfrak{e}_{31} =$	CF390000
$\mathfrak{e}_8 =$	E1009816	$\mathfrak{e}_{20} =$	3BBAD552		

Note that instead of the fixed coefficients of C it is preferable to use the fixed optimization tables $\{\mathfrak{T}_i\}_{i=0}^{t-1}$ precomputed from $\mathfrak{e}, \{\mathfrak{e}_i\}_{i=0}^{31}$ for a window of size w bits in the following way:

$$\mathfrak{T}_i[j] = \sum_{l=0}^{w-1} \delta_l(j) \cdot \mathfrak{e}_{w \cdot i + l} \bmod 2^{32}, \quad j = 0, \ldots, 2^w - 1, \tag{12}$$

for $i = 1, \ldots, t-1$ and

$$\mathfrak{T}_0[j] = \mathfrak{e} + \sum_{l=0}^{w-1} \delta_l(j) \cdot \mathfrak{e}_l \bmod 2^{32}, \quad j = 0, \ldots, 2^w - 1, \tag{13}$$

for $i = 0$.

Let k be the key of length 96 bit,

$$k \in \mathbb{Z}/2^{96}\mathbb{Z},$$
$$k = (\bar{k}_2, \bar{k}_1, \bar{k}_0) \in (\mathbb{Z}/2^{32}\mathbb{Z})^3, \ \bar{k}_i \in \mathbb{Z}/2^{32}\mathbb{Z}, \ i = 0, \ldots, 2.$$

Then the key initialization stage can be carried out in the following

way:

$$\begin{array}{lcl} e & \leftarrow & \mathfrak{e}; \\ e_i & \leftarrow & \mathfrak{e}_i,\ i=0,\ldots,31; \\ d_0 & \leftarrow & \mathfrak{d}_0; \\ \delta_1(d_0) & \leftarrow & \delta_1(\mathfrak{d}_0) \ \mathsf{XOR}\ \delta_1(\bar{k}_1); \\ d_1 & \leftarrow & \mathfrak{d}_1; \\ x & \leftarrow & \bar{k}_2; \\ \bar{z}_0 & \leftarrow & \bar{k}_1 \ \mathsf{OR}\ 2; \\ \bar{z}_1 & \leftarrow & \bar{k}_0. \end{array}$$

For a 128-bit key k,

$$k \in \mathbb{Z}/2^{128}\mathbb{Z},$$
$$k = (\bar{k}_3, \bar{k}_2, \bar{k}_1, \bar{k}_0) \in (\mathbb{Z}/2^{32}\mathbb{Z})^4,\ \bar{k}_i \in \mathbb{Z}/2^{32}\mathbb{Z},\ i=0,\ldots,3,$$

the key initialization stage can be performed as follows:

$$\begin{array}{lcl} e & \leftarrow & \mathfrak{e}; \\ e_i & \leftarrow & \mathfrak{e}_i,\ i=0,\ldots,31; \\ d_0 & \leftarrow & \mathfrak{d}_0; \\ [d_0]_2^{17} & \leftarrow & [\mathfrak{d}_0]_2^{17} \ \mathsf{XOR}\ \left[\bar{k}_3\right]_0^{15}; \\ \delta_1(d_0) & \leftarrow & \delta_1(\mathfrak{d}_0) \ \mathsf{XOR}\ \delta_1(\bar{k}_1); \\ d_1 & \leftarrow & \mathfrak{d}_1; \\ [d_1]_2^{17} & \leftarrow & [\mathfrak{d}_1]_2^{17} \ \mathsf{XOR}\ \left[\bar{k}_3\right]_{16}^{32}; \\ x & \leftarrow & \bar{k}_2; \\ \bar{z}_0 & \leftarrow & \bar{k}_1 \ \mathsf{OR}\ 2; \\ \bar{z}_1 & \leftarrow & \bar{k}_0. \end{array}$$

Note that the user does not have to compute the optimization tables for the efficient computation of S since S is fixed. In this case some additional effort is required to select the appropriate values of the coefficients. This can be a subject of the further work.

The last case may require several runs of the state transition function with feedback before allowing it to produce some output in order for the diffusion characteristics to be acceptable. A similar technique is used in one of the following key expansion methods. But there are applications that do not require this warmup.

Standard case

As standard we regard the case involving no fixed bits of the coefficients of B or C except for those implied by (1), (2) and (3) and using a 128-bit primary key and a 128-bit initial value. In all the following cases let k be the 128-bit primary key,

$$k \in \mathbb{Z}/2^{128}\mathbb{Z},\ k = (\bar{k}_3, ..., \bar{k}_0) \in (\mathbb{Z}/2^{32}\mathbb{Z})^4, \tag{14}$$

and iv be a 128-bit initial value,

$$iv \in \mathbb{Z}/2^{128}\mathbb{Z},\ iv = (\bar{iv}_3, ..., \bar{iv}_0) \in (\mathbb{Z}/2^{32}\mathbb{Z})^4. \tag{15}$$

By applying the notation from (11) we get $m = 128$. So we need to specify the key expansion function $G : \mathbb{Z}/2^{128}\mathbb{Z} \to \mathbb{Z}/2^{1195}\mathbb{Z}$ and obtain $k' \in \mathbb{Z}/2^{1195}\mathbb{Z}$ as a result.

1. `A 128-bit version of C`. This mapping is a version of the primitive C with 128-bit arguments and coefficients. To guarantee some good properties of the mapping we use 128-bit coefficients of a special form and take 2 similar 128-bit mappings in sequence. Let u be the first function and v the second one. They can be specified through the corresponding sets of coefficients:

$$\begin{aligned} u(x) &= \alpha + \textstyle\sum_{i=0}^{127} \alpha_i \delta_i(x) \bmod 2^{128}, \\ v(x) &= \beta + \textstyle\sum_{i=0}^{127} \beta_i \delta_i(x) \bmod 2^{128}, \end{aligned} \tag{16}$$

where $x \in \mathbb{Z}/2^{128}\mathbb{Z}$, $\delta_i : x \mapsto x_i$, $i = 0, \ldots, 127$, selects the i-th bit of x. α and β are in $\mathbb{Z}/2^{128}\mathbb{Z}$ and have 1 in their least significant binary positions ($\delta_0(\alpha) = \delta_0(\beta) = 1$), the other bits being specified at will. α_0 and β_{127} are in $\mathbb{Z}/2^{128}\mathbb{Z}$ and have 1 in their least significant binary positions, 0 in the next-to-least binary positions ($\alpha_0 \equiv \beta_{127} \equiv 1 \pmod 4$), the other bits being specified at will. $\alpha_{127} = 2^{127} \in \mathbb{Z}/2^{128}\mathbb{Z}$, $\beta_0 = 2^{127} \in \mathbb{Z}/2^{128}\mathbb{Z}$. α_i, $i = 1, 2, \ldots, 126$, has 0 in bits $0, 1, \ldots, i-1$ and 1 in bit i, the other bits being specified at will. β_i, $i = 1, 2, \ldots, 126$, has 0 in bits $0, 1, \ldots, 126-i$ and 1 in bit $127-i$, the other bits being specified at will. Then we define the function $h : \mathbb{Z}/2^{128}\mathbb{Z} \to \mathbb{Z}/2^{128}\mathbb{Z}$ as the subsequent application of u and v to x:

$$h : x \mapsto v(u(x)), \tag{17}$$

and define function $h^2(k) = h(h(k))$. u and v are both single cycle functions. This property excludes the possibility to face two equivalent values of k' using different values of k in the following key setup algorithm:

INPUT: $k = (\bar{k}_3, \ldots, \bar{k}_0)$
TEMPORARY VARIABLES: i
  $k \leftarrow h^2(k)$;
  for i from 0 to 7 do
    $k \leftarrow h(k)$;
    $\bar{k}'_{4\cdot i} \leftarrow \bar{k}_0$;
    $\bar{k}'_{4\cdot i+1} \leftarrow \bar{k}_1$;

$\bar{k}'_{4\cdot i+2} \leftarrow \bar{k}_2;$
$\bar{k}'_{4\cdot i+3} \leftarrow \bar{k}_3;$
end for
$k \leftarrow h(k);$
$\bar{k}'_{32} \leftarrow \left[\bar{k}_0\right]_0^{30};$
$\bar{k}'_{33} \leftarrow \left[\bar{k}_1\right]_0^{29};$
$\bar{k}'_{34} \leftarrow \left[\bar{k}_2\right]_0^{29};$
$\bar{k}'_{35} \leftarrow \bar{k}_3;$
$k \leftarrow h(k);$
$\bar{k}'_{36} \leftarrow \bar{k}_0;$
$\bar{k}'_{37} \leftarrow \bar{k}_1;$
OUTPUT: $k' = (\bar{k}'_{37}, \ldots, \bar{k}'_0)$

To be able to use u and v in this algorithm it is necessary to somehow define their coefficients. The transformation h could be used as a good stream cipher itself but for its relatively low throughput (about 1,3 Gbps on a standard 3,2 GHz Intel Pentium 4 processor).

2. `AES round function`. In this case the function G uses the non-linear round function of AES (without adding any subkeys) for a 128-bit block as a primitive. Let f denote the round function of AES:

$$f : \mathbb{Z}/2^{128}\mathbb{Z} \to \mathbb{Z}/2^{128}\mathbb{Z}.$$

f performs 16 byte substitutions, 3 right byte rotations, and 1 matrix multiplication. It is well known [21] that this function turns out be to statistically good after 3 subsequent applications. Moreover, the change of a single bit spreads to the whole block after 2 rounds for 128-bit blocks. We build G relying upon these facts. Having defined $f^2(k) = f(f(k))$ and $f^3(k) = f(f^2(k))$, we can specify G through the following algorithm, the idea similar to that in [11] being used:

INPUT: $k = (\bar{k}_3, \ldots, \bar{k}_0)$
TEMPORARY VARIABLES: $c \in \mathbb{Z}/2^{128}\mathbb{Z}, i$
$c \leftarrow f^2(k);$
for i from 0 to 7 do
$k \leftarrow f^3(k) \,\mathsf{XOR}\, c;$
$\bar{k}'_{4\cdot i} \leftarrow \bar{k}_0;$
$\bar{k}'_{4\cdot i+1} \leftarrow \bar{k}_1;$
$\bar{k}'_{4\cdot i+2} \leftarrow \bar{k}_2;$
$\bar{k}'_{4\cdot i+3} \leftarrow \bar{k}_3;$

end for
$k \leftarrow f^3(k)$ XOR c;
$\bar{k}'_{32} \leftarrow [\bar{k}_0]_0^{30}$;
$\bar{k}'_{33} \leftarrow [\bar{k}_1]_0^{29}$;
$\bar{k}'_{34} \leftarrow [\bar{k}_2]_0^{29}$;
$\bar{k}'_{35} \leftarrow \bar{k}_3$;
$k \leftarrow f^3(k)$;
$\bar{k}'_{36} \leftarrow \bar{k}_0$;
$\bar{k}'_{37} \leftarrow \bar{k}_1$;
OUTPUT: $k' = (\bar{k}'_{37}, \ldots, \bar{k}'_0)$

3. `Self-initialization with feedback`. In this case G is based upon the key stream generator of ABC, defined in Section 3 as algorithm "ABC key stream generator". We will denote a single call of ABC key stream generator as a function $g(\bar{z}_0, \bar{z}_1, x, d_0, d_1, e, \{e_i\}_{i=0}^{31})$:

$$g : \mathbb{Z}/2^{1216}\mathbb{Z} \rightarrow \mathbb{Z}/2^{32}\mathbb{Z}.$$

If the optimization tables $\{T_i\}_{i=0}^{t-1}$ are precomputed, the ABC key stream generator can be alternatively called as $g(\bar{z}_0, \bar{z}_1, x, d_0, d_1, \{T_i\}_{i=0}^{t-1})$. In the latter definition the optimization tables are passed to the generator instead of the coefficients of C. We note that this is a preferable way of calling g, the former definition being the most suitable one in case no window optimization is used. In both calls variables $\bar{z}_0, \bar{z}_1, x$ are changed in a call (i.e. within the function g).

In this algorithm we first use a set of fixed values $\mathfrak{e}, \{\mathfrak{e}_i\}_{i=0}^{31}, \mathfrak{d}_0, \mathfrak{d}_1$ and a 128-bit key k exactly as in 128-bit version of extreme case 3 to fill the temporary initial values $\bar{z}'_0, \bar{z}'_1, x', d'_0, d'_1 \in \mathbb{Z}/2^{32}\mathbb{Z}$. We note again that the fixed values with some statistical properties satisfied are given in table 2. These values $\mathfrak{e}, \{\mathfrak{e}_i\}_{i=0}^{31}, \mathfrak{d}_0, \mathfrak{d}_1$ satisfy the requirements imposed by (2) and (4). Moreover, when optimization is employed, the fixed precomputed optimization tables $\{\mathfrak{T}_i\}_{i=0}^{t-1}$ can be used instead of the fixed coefficients of C. We describe the way of the fixed optimization tables precomputation from $\mathfrak{e}, \{\mathfrak{e}_i\}_{i=0}^{31}$ by equations (12) and (13).

Then we perform a number of the ABC key stream generator warmup iterations, calling $g(\bar{z}'_0, \bar{z}'_1, x', d'_0, d'_1, \mathfrak{e}, \{\mathfrak{e}_i\}_{i=0}^{31})$. As already mentioned, it is preferable to substitute this call by $g(\bar{z}'_0, \bar{z}'_1, x', d'_0, d'_1, \{\mathfrak{T}_i\}_{i=0}^{t-1})$ in case optimization is used. After each iteration a 32-bit output of generator is feeded back to some part of the state by means of bitwise addition modulo 2.

Finally, after the warmup phase we call the ABC key stream generator 38 times to obtain the values for $e, \{e_i\}_{i=0}^{31}, d_0, d_1, x, \bar{z}_0, \bar{z}_1$, composing the main state of ABC.

In case optimization is not used, the initialization function G can be outlined in the following way:

INPUT: $k = (\bar{k}_3, \ldots, \bar{k}_0)$, $\bar{z}_0, \bar{z}_1, x, d_0, d_1, e, \{e_i\}_{i=0}^{31}$, $\mathfrak{d}_0, \mathfrak{d}_1, \mathfrak{e}, \{\mathfrak{e}_i\}_{i=0}^{31}$

TEMPORARY VARIABLES: $\bar{z}'_0, \bar{z}'_1, x', d'_0, d'_1, i, \zeta$

Initialization:

$d'_0 \leftarrow \mathfrak{d}_0;$
$[d'_0]_2^{17} \leftarrow [d'_0]_2^{17} \,\mathsf{XOR}\, \left[\bar{k}_3\right]_0^{15};$
$\delta_1(d'_0) \leftarrow \delta_1(\mathfrak{d}_0) \,\mathsf{XOR}\, \delta_1(\bar{k}_1);$
$d'_1 \leftarrow \mathfrak{d}_1;$
$[d'_1]_2^{17} \leftarrow [d'_1]_2^{17} \,\mathsf{XOR}\, \left[\bar{k}_3\right]_{16}^{31};$
$x' \leftarrow \bar{k}_2;$
$\bar{z}'_0 \leftarrow \bar{k}_1 \,\mathsf{OR}\, 2;$
$\bar{z}'_1 \leftarrow \bar{k}_0.$

Initial state warm-up with feedback:

$\zeta \leftarrow g(\bar{z}'_0, \bar{z}'_1, x', d'_0, d'_1, \mathfrak{e}, \{\mathfrak{e}_i\}_{i=0}^{31});$
$x' \leftarrow x' \,\mathsf{XOR}\, \zeta;$
$\zeta \leftarrow g(\bar{z}'_0, \bar{z}'_1, x', d'_0, d'_1, \mathfrak{e}, \{\mathfrak{e}_i\}_{i=0}^{31});$
$d'_0 \leftarrow (d'_0 \,\mathsf{XOR}\, \zeta) \,\mathsf{OR}\, 1;$
$\zeta \leftarrow g(\bar{z}'_0, \bar{z}'_1, x', d'_0, d'_1, \mathfrak{e}, \{\mathfrak{e}_i\}_{i=0}^{31});$
$d'_1 \leftarrow (d'_1 \,\mathsf{XOR}\, \zeta) \,\mathsf{AND}\, \underbrace{1\ldots1}_{30}00_2;$
$\zeta \leftarrow g(\bar{z}'_0, \bar{z}'_1, x', d'_0, d'_1, \mathfrak{e}, \{\mathfrak{e}_i\}_{i=0}^{31});$
$\bar{z}'_0 \leftarrow (\bar{z}'_0 \,\mathsf{XOR}\, \zeta) \,\mathsf{OR}\, 2;$
$\zeta \leftarrow g(\bar{z}'_0, \bar{z}'_1, x', d'_0, d'_1, \mathfrak{e}, \{\mathfrak{e}_i\}_{i=0}^{31});$
$\bar{z}'_1 \leftarrow \bar{z}'_1 \,\mathsf{XOR}\, \zeta;$

Main state filling:

$e \leftarrow g(\bar{z}'_0, \bar{z}'_1, x', d'_0, d'_1, \mathfrak{e}, \{\mathfrak{e}_i\}_{i=0}^{31});$
for i from 0 to 30 do
    $e_i \leftarrow g(\bar{z}'_0, \bar{z}'_1, x', d'_0, d'_1, \mathfrak{e}, \{\mathfrak{e}_i\}_{i=0}^{31});$
end for
$e_{31} \leftarrow (g(\bar{z}'_0, \bar{z}'_1, x', d'_0, d'_1, \mathfrak{e}, \{\mathfrak{e}_i\}_{i=0}^{31}) \,\mathsf{AND}\, \underbrace{1\ldots1}_{16}\underbrace{0\ldots0}_{16}{}_2) \,\mathsf{OR}\, \underbrace{0\ldots0}_{15}1\underbrace{0\ldots0}_{16}{}_2;$
$d_0 \leftarrow g(\bar{z}'_0, \bar{z}'_1, x', d'_0, d'_1, \mathfrak{e}, \{\mathfrak{e}_i\}_{i=0}^{31}) \,\mathsf{OR}\, 1;$
$d_1 \leftarrow g(\bar{z}'_0, \bar{z}'_1, x', d'_0, d'_1, \mathfrak{e}, \{\mathfrak{e}_i\}_{i=0}^{31}) \,\mathsf{AND}\, \underbrace{1\ldots1}_{30}00_2;$
$x \leftarrow g(\bar{z}'_0, \bar{z}'_1, x', d'_0, d'_1, \mathfrak{e}, \{\mathfrak{e}_i\}_{i=0}^{31});$
$\bar{z}_0 \leftarrow g(\bar{z}'_0, \bar{z}'_1, x', d'_0, d'_1, \mathfrak{e}, \{\mathfrak{e}_i\}_{i=0}^{31}) \,\mathsf{OR}\, 2;$
$\bar{z}_1 \leftarrow g(\bar{z}'_0, \bar{z}'_1, x', d'_0, d'_1, \mathfrak{e}, \{\mathfrak{e}_i\}_{i=0}^{31}).$

OUTPUT: $\bar{z}_0, \bar{z}_1, x, d_0, d_1, e, \{e_i\}_{i=0}^{31}$

In case optimization is used, the initialization function G can be described as above with fixed C coefficients $\mathfrak{e}, \{\mathfrak{e}_i\}_{i=0}^{31}$ substituted by fixed precomputed optimization tables $\{\mathfrak{T}_i\}_{i=0}^{t-1}$. In Appendix C we give some statistical properties of the function G described here as used in key setup routine.

Initial value setup

Assume that the selected standard key setup routine has been already run and the tables $\{T_i\}_{i=0}^{t-1}$ have been precomputed (in case optimization is employed). We suggest making the nonce setup in the following way:

INPUT: $iv = (\bar{iv}_3, \ldots, \bar{iv}_0), \{T_i\}_{i=0}^{t-1}$

TEMPORARY VARIABLES: ζ

IV application:

$[d_0]_2^{17} \leftarrow [d_0]_2^{17} \,\mathsf{XOR}\, [\bar{iv}_3]_0^{15};$
$\delta_1(d_0) \leftarrow \delta_1(d_0) \,\mathsf{XOR}\, \delta_1(\bar{iv}_1);$
$[d_1]_2^{17} \leftarrow [d_1]_2^{17} \,\mathsf{XOR}\, [\bar{iv}_3]_{16}^{31};$
$x \leftarrow x \,\mathsf{XOR}\, \bar{iv}_2;$
$\bar{z}_0 \leftarrow (\bar{z}_0 \,\mathsf{XOR}\, \bar{iv}_1) \,\mathsf{OR}\, 2;$
$\bar{z}_1 \leftarrow \bar{z}_1 \,\mathsf{XOR}\, \bar{iv}_0;$

Warm-up with feedback:

$\zeta \leftarrow g(\bar{z}_0, \bar{z}_1, x, d_0, d_1, \{T_i\}_{i=0}^{t-1});$
$x \leftarrow x \,\mathsf{XOR}\, \zeta;$
$\zeta \leftarrow g(\bar{z}_0, \bar{z}_1, x, d_0, d_1, \{T_i\}_{i=0}^{t-1});$
$d_0 \leftarrow (d_0 \,\mathsf{XOR}\, \zeta) \,\mathsf{OR}\, 1;$
$\zeta \leftarrow g(\bar{z}_0, \bar{z}_1, x, d_0, d_1, \{T_i\}_{i=0}^{t-1});$
$d_1 \leftarrow (d_1 \,\mathsf{XOR}\, \zeta) \,\mathsf{AND}\, \underbrace{1\ldots1}_{30}00_2;$
$\zeta \leftarrow g(\bar{z}_0, \bar{z}_1, x, d_0, d_1, \{T_i\}_{i=0}^{t-1});$
$\bar{z}_0 \leftarrow (\bar{z}_0 \,\mathsf{XOR}\, \zeta) \,\mathsf{OR}\, 2;$
$\zeta \leftarrow g(\bar{z}_0, \bar{z}_1, x, d_0, d_1, \{T_i\}_{i=0}^{t-1});$
$\bar{z}_1 \leftarrow \bar{z}_1 \,\mathsf{XOR}\, \zeta.$

OUTPUT: $\bar{z}_0, \bar{z}_1, x, d_0, d_1$

If optimization is not used, $\{T_i\}_{i=0}^{t-1}$ in the above algorithm are substituted by $e, \{e_i\}_{i=0}^{31}$.

So we add iv to $\bar{z}_0, \bar{z}_1, x, d_0, d_1$ bitwise modulo 2 and then let the key stream generator warm up with feedback. As a matter of fact we influence the initial state of the whole key stream generation mapping by making it run from a new start point. Empirical statistical evaluation shows strong propagation properties of this algorithm (see Appendix C).

Summing up: The basic setup routine

Although it is possible to make use of any combination of key expansion/nonce setup functions described above, it is necessary to describe the basic method used at the initialization stage. We assume that the table optimization is employed and use $\{\mathfrak{T}_i\}_{i=0}^{t-1}$ or $\{T_i\}_{i=0}^{t-1}$ in calls to the ABC key stream generator g.

ABC SETUP ROUTINE

KEY EXPANSION

INPUT: $k = (\bar{k}_3, \ldots, \bar{k}_0)$, $\bar{z}_0, \bar{z}_1, x, d_0, d_1, e, \{e_i\}_{i=0}^{31}$, $\mathfrak{d}_0, \mathfrak{d}_1, \{\mathfrak{T}_i\}_{i=0}^{t-1}$

TEMPORARY VARIABLES: $\bar{z}'_0, \bar{z}'_1, x', d'_0, d'_1, i, \zeta$

Initialization:

$d'_0 \leftarrow \mathfrak{d}_0;$
$[d'_0]_2^{17} \leftarrow [d'_0]_2^{17} \,\mathsf{XOR}\, \left[\bar{k}_3\right]_0^{15};$
$\delta_1(d'_0) \leftarrow \delta_1(\mathfrak{d}_0) \,\mathsf{XOR}\, \delta_1(\bar{k}_1);$
$d'_1 \leftarrow \mathfrak{d}_1;$
$[d'_1]_2^{17} \leftarrow [d'_1]_2^{17} \,\mathsf{XOR}\, \left[\bar{k}_3\right]_{16}^{31};$
$x' \leftarrow \bar{k}_2;$
$\bar{z}'_0 \leftarrow \bar{k}_1 \,\mathsf{OR}\, 2;$
$\bar{z}'_1 \leftarrow \bar{k}_0.$

Initial state warm-up with feedback:

$\zeta \leftarrow g(\bar{z}'_0, \bar{z}'_1, x', d'_0, d'_1, \{\mathfrak{T}_i\}_{i=0}^{t-1});$
$x' \leftarrow x' \,\mathsf{XOR}\, \zeta;$
$\zeta \leftarrow g(\bar{z}'_0, \bar{z}'_1, x', d'_0, d'_1, \{\mathfrak{T}_i\}_{i=0}^{t-1});$
$d'_0 \leftarrow (d'_0 \,\mathsf{XOR}\, \zeta) \,\mathsf{OR}\, 1;$
$\zeta \leftarrow g(\bar{z}'_0, \bar{z}'_1, x', d'_0, d'_1, \{\mathfrak{T}_i\}_{i=0}^{t-1});$
$d'_1 \leftarrow (d'_1 \,\mathsf{XOR}\, \zeta) \,\mathsf{AND}\, \underbrace{1\ldots1}_{30}00_2;$
$\zeta \leftarrow g(\bar{z}'_0, \bar{z}'_1, x', d'_0, d'_1, \{\mathfrak{T}_i\}_{i=0}^{t-1});$
$\bar{z}'_0 \leftarrow (\bar{z}'_0 \,\mathsf{XOR}\, \zeta) \,\mathsf{OR}\, 2;$

$\zeta \leftarrow g(\bar{z}'_0, \bar{z}'_1, x', d'_0, d'_1, \{\mathfrak{T}_i\}_{i=0}^{t-1});$
$\bar{z}'_1 \leftarrow \bar{z}'_1 \text{ XOR } \zeta;$

Main state filling:

$e \leftarrow g(\bar{z}'_0, \bar{z}'_1, x', d'_0, d'_1, \{\mathfrak{T}_i\}_{i=0}^{t-1});$
for i from 0 to 30 do
$e_i \leftarrow g(\bar{z}'_0, \bar{z}'_1, x', d'_0, d'_1, \{\mathfrak{T}_i\}_{i=0}^{t-1});$
end for
$e_{31} \leftarrow (g(\bar{z}'_0, \bar{z}'_1, x', d'_0, d'_1, \{\mathfrak{T}_i\}_{i=0}^{t-1}) \text{ AND } \underbrace{1\ldots1}_{16}\underbrace{0\ldots0}_{16}{}_2) \text{ OR } \underbrace{0\ldots0}_{15}1\underbrace{0\ldots0}_{16}{}_2;$
$d_0 \leftarrow g(\bar{z}'_0, \bar{z}'_1, x', d'_0, d'_1, \{\mathfrak{T}_i\}_{i=0}^{t-1}) \text{ OR } 1;$
$d_1 \leftarrow g(\bar{z}'_0, \bar{z}'_1, x', d'_0, d'_1, \{\mathfrak{T}_i\}_{i=0}^{t-1}) \text{ AND } \underbrace{1\ldots1}_{30}00_2;$
$x \leftarrow g(\bar{z}'_0, \bar{z}'_1, x', d'_0, d'_1, \{\mathfrak{T}_i\}_{i=0}^{t-1});$
$\bar{z}_0 \leftarrow g(\bar{z}'_0, \bar{z}'_1, x', d'_0, d'_1, \{\mathfrak{T}_i\}_{i=0}^{t-1}) \text{ OR } 2;$
$\bar{z}_1 \leftarrow g(\bar{z}'_0, \bar{z}'_1, x', d'_0, d'_1, \{\mathfrak{T}_i\}_{i=0}^{t-1}).$

OUTPUT: $\bar{z}_0, \bar{z}_1, x, d_0, d_1, e, \{e_i\}_{i=0}^{31}$

OPTIMIZATION TABLES PRECOMPUTATION

INPUT: $w, t, e, \{e_i\}_{i=0}^{31}$
TEMPORARY VARIABLES: i, j, l

for i from 1 to $t-1$ do
for j from 0 to $2^w - 1$ do
$T_i[j] = 0$
for l from 0 to $w-1$ do
$T_i[j] \leftarrow T_i[j] + \delta_l(j) \cdot e_{w \cdot i + l}$
for j from 0 to $2^w - 1$ do
$T_0[j] = e;$
for l from 0 to $w-1$ do
$T_0[j] \leftarrow T_0[j] + \delta_l(j) \cdot e_l;$

OUTPUT: $T_0, \ldots, T_{t-1}$

IV SETUP

INPUT: $iv = (\bar{iv}_3, \bar{iv}_2, \bar{iv}_1, \bar{iv}_0), \{T_i\}_{i=0}^{t-1}$
TEMPORARY VARIABLES: ζ

IV application:

$[d_0]_2^{17} \leftarrow [d_0]_2^{17} \text{ XOR } [\bar{iv}_3]_0^{15};$
$\delta_1(d_0) \leftarrow \delta_1(d_0) \text{ XOR } \delta_1(\bar{iv}_1);$
$[d_1]_2^{17} \leftarrow [d_1]_2^{17} \text{ XOR } [\bar{iv}_3]_{16}^{31};$

$x \leftarrow x \,\mathsf{XOR}\, \bar{iv}_2$;
$\bar{z}_0 \leftarrow (\bar{z}_0 \,\mathsf{XOR}\, \bar{iv}_1) \,\mathsf{OR}\, 2$;
$\bar{z}_1 \leftarrow \bar{z}_1 \,\mathsf{XOR}\, \bar{iv}_0$.

Warm-up with feedback:

$\zeta \leftarrow g(\bar{z}_0, \bar{z}_1, x, d_0, d_1, \{T_i\}_{i=0}^{t-1})$;
$x \leftarrow x \,\mathsf{XOR}\, \zeta$;
$\zeta \leftarrow g(\bar{z}_0, \bar{z}_1, x, d_0, d_1, \{T_i\}_{i=0}^{t-1})$;
$d_0 \leftarrow (d_0 \,\mathsf{XOR}\, \zeta) \,\mathsf{OR}\, 1$;
$\zeta \leftarrow g(\bar{z}_0, \bar{z}_1, x, d_0, d_1, \{T_i\}_{i=0}^{t-1})$;
$d_1 \leftarrow (d_1 \,\mathsf{XOR}\, \zeta) \,\mathsf{AND}\, \underbrace{1\ldots1}_{30}00_2$;
$\zeta \leftarrow g(\bar{z}_0, \bar{z}_1, x, d_0, d_1, \{T_i\}_{i=0}^{t-1})$;
$\bar{z}_0 \leftarrow (\bar{z}_0 \,\mathsf{XOR}\, \zeta) \,\mathsf{OR}\, 2$;
$\zeta \leftarrow g(\bar{z}_0, \bar{z}_1, x, d_0, d_1, \{T_i\}_{i=0}^{t-1})$;
$\bar{z}_1 \leftarrow \bar{z}_1 \,\mathsf{XOR}\, \zeta$;

OUTPUT: $\bar{z}_0, \bar{z}_1, x, d_0, d_1$

It is important to note that once the key setup routine was run and the tables were precomputed, there is no need in the table precomputation when the same key is being set up again. Moreover, the state variables $\bar{z}_0, \bar{z}_1, x, d_0, d_1$ can be stored immediately after the completion of the key expansion routine and then be used to restore the state prior to IV setup, thus shortening the setup routine.

5 Brief security analysis

To analyse the security of the ABC stream cipher it is important to stress that the cipher is very flexible; that is, we impose any extra restrictions on coefficients neither of function B nor of function C, except for the ones that are imposed above. In other words, we assume that the rest of bits of these coefficients are produced in a key-dependent way out of the key by a certain routine, which is, in fact, a sort of PRNG, and which could be user-defined (for a concrete routine see Section 4). That is, a key expansion procedure is applied to a key to produce sufficiently many pseudorandom bits to fill the coefficients of the functions B and C, and the registers that store initial values of x and z. This implies that *neither coefficients of the functions B and C nor the initial states x and z are known to an adversary.*

General attacks

We have discovered no vulnerabilities of the ABC stream cipher with respect to a set of commonly considered attacks and have not found any weak keys.

Time-memory-data tradeoffs, either suggested by Biryukov and Shamir in [8] or the generalized ones suggested by Hong and Sarkar in [13] (see [10] for discussion), are not posing a threat to the ABC cipher. For the former case the TMD complexity is greater than 2^{128} exhaustive search due to large state size (about 1200 bits). The latter case also does not lower the exhaustive search threshold even if unlimited precomputational resources are considered, as ABC supports the 128-bit initial value size, which is equal to key size.

Related-key and resynchronization attacks are withstood by proper choice of key setup and IV initialization algorithms. Our suggestion for both algorithms (see Section 4) uses self-initializing with feedback, which does not show possibilities for applying these techniques.

Algebraic attacks are withstood by the non-linear properties of the output primitive (see Appendix A). The latter also does not provide possibilities for *correlation* and *linear* attacks.

Empirical statistical testing, performed with NIST suite with respect to AES candidates evaluation (see [22]), has not indicated any deviation of ABC key stream from a random sequence. Moreover, our testing has shown that the key stream statistical properties provided by ABC are at least as good as those of AES finalists, given in [21]).

Resistance to side-channel attacks

Ciphers subject to implementation on hardware or some constraint platforms including chip cards are often required to be resistant to side-channel attacks. The introduction of the table optimization technique was motivated (among other things) by the considerations of the security of ABC with respect to side-channel attacks. Note that if the table optimization is not employed and S is computed as in (3), the implementation can be vulnerable with regard to simple side-channel attacks such as simple timing attack or simple power analysis, since the Hamming weight of x can be easily restored. Note that *we do not recommend using equation (3) to compute S if the implementation is likely to be subject to side-channel attacks*. The users are referred to equation (8) with an appropriate proper value of w instead.

Here we introduce a simple technique which can be used to randomize the computation of the function S defined by (3) and, hence, to hamper different types of differential analysis including differential power analysis (DPA) or possible analogues of the Goubin-type analysis for stream ciphers. In the unprotected version of S we use t tables, each containing 2^w 32-bit elements computed according to (6) and (7). Each table except T_0 has at least one zero element $T_1[0] = \cdots = T_{t-1}[0] = 0$ which means that if we have a zero window in x then this case stands out against the background of power consumption in all the other cases. This can lead to an attack and should be avoided.

Let t be even and $r \in \mathbb{Z}/2^{32}\mathbb{Z}$ be a random number (taken for instance from a physical random number generator available on some chip cards). Assume that the key expansion routine has already been run. This means that we have generated $e, e_0, \ldots, e_{31}$ and precomputed the tables $T_0, \ldots, T_{t-1}$ according to (6) and (7). Now we modify each table by adding r or $-r$ to the table elements depending on the parity of the table number:

$$T_i[j] = -r + T_i[j] \pmod{2^{32}}, \quad j = 0, \ldots, 2^w - 1, \tag{18}$$

for $i = 1, 3, \ldots, t-1$ and

$$T_i[j] = r + T_i[j] \pmod{2^{32}}, \quad j = 0, \ldots, 2^w - 1, \tag{19}$$

for $i = 0, 2, \ldots, t-2$.

During the computation of S in accordance with (8) $-r$ and r annul each other because of t being even. If t has been selected to be odd, then one can add r to the elements of T_0, $-2 \cdot r$ to T_1, and then proceed with (18) and (19). One has to perform $2^w \cdot t$ extra additions[1] modulo 2^{32} at the initialization stage to apply this trick.

In applications where it it possible to compute the optimization tables only once and then to store these for future use (performing the initial setup routine when needed), this masking operation can be performed only once during the key expansion procedure. This does not randomize the computation of S, but enables to avoid additions with zeros in case of zero windows within x, which raises the security of ABC with regard to side-channel attacks. This (simplified) countermeasure requires no additional operations at the initialization stage.

6 Implementation

Software implementation for 32-bit processors

Although the ABC stream cipher shows very good throughput results on every software platform, it is optimized to be used on 32-bit processors such as Intel Pentium 4 or PowerPC G4+.

Our C reference implementation was compiled using gcc 3.3.1 without any processor-specific options. Hence the throughput estimates we give here are of generic nature. We expect that usage of additional processor-specific compiler options or some specific compilers, such as Intel C++ compiler, could lead to performance improvement. The corresponding programs were run under Linux with a 2.4.21 kernel on a desktop with a 3.2 GHz Intel Pentium 4 processor with 8KB L1 cache and 512KB L2 cache, and 512 MB main memory. We measured the execution time in ms needed to produce

[1] In fact the number of extra additions is different if t is odd, since the lengths of the windows are unequal.

a 1024-byte output buffer 1310720 times (in total producing 10 Gbit of data) and then calculated the throughput in Gbps, processor cycles per byte and per 64-byte block of data. The best throughput for this hardware configuration was achieved with $w = 8$. The results of the throughput measurement for C reference implementation including the memory needed for the lookup tables can be found in Table 3 for different measurement conditions.

Table 3: ABC throughput for Intel Pentium 4

w	Speed, Gbps	Cycles per byte	Cycles per block	Memory, bytes
1	0.21	121.91	7802	132
2	2.25	11.38	728	256
4	4.24	6.04	387	512
8	6.86	3.73	239	4096

Additionally we measured the cost of key setup, including the precomputation of optimization tables, and of the initial value setup in processor cycles on the same hardware platform. Also the cost of optimization tables precomputation was separately measured. The initialization routines are implemented in C. The results can be found in Table 4. Here one mode of operation was used only. We ran the key initialization routine and the IV setup routine 1000 times separately, measuring the number of clocks needed. Then we divided the values through 1000 to get the result in its final shape.

Table 4: Cost of ABC setup routines in processor cycles for Intel Pentium 4

w	Key setup with precomputation	IV setup	Table precomputation	Key setup without precomputation
1	22952	5732	0	22952
2	2906	776	682	2224
4	5509	690	3916	1593
8	89313	631	88053	1260

Measurement results make it clear that precomputation of the optimization tables has the major impact on the total cost of the key setup procedure. Thus, we recommend choosing smaller window sizes when dealing with encryption of short packets, which would raise the total performance of ABC in this case. Exact values depend, however, on a specific platform.

Our implementation can be flexibly tuned for maximum performance on a specific platform by choosing the appropriate values of two implementation parameters. The first parameter is the length of optimization window w. The second parameter, to which we refer as unroll depth, is the number of

ABC core transform iterations explicitly unrolled within cycles producing a number of key stream bytes. In our C reference implementation we allow users to choose one of the 10 predefined variants of optimization window length and unroll depth combinations at compile time. We expect that different variants will show different performance on specific platforms. The choice of the variant that shows the best performance depends on various parameters of a platform, such as processor architecture, relative costs of processor operations, L1 and L2 cache sizes, relative costs of the access operations to different types of RAM/ROM, and also type of operation system, version of compiler, compiler options, and many others.

We also expect a speedup of ABC for the implementation in assembly language, which invokes the usage of SIMD extensions available for specific processors.

In Table 5 and Table 6 we give the results of reference C implementation throughput and setup costs measurement for the 32-bit ARM7TDMI processor. The code was cross compiled using ARM Developer Suite 1.2 and the running time was measured on the Evaluator-7T board with KS32C50100 ARM7TDMI microprocessor clocked at 50 Mhz. It has a 512 kByte flash EPROM with 512 kByte SRAM and 8 kByte cache. The testing methodology was same as that used for Intel Pentium 4 processor. 10 Gbit of output data was generated and key setup and IV setup procedures were run 1000000 times.

Table 5: ABC throughput for ARM7TDMI

w	Speed Mbps	Cycles per byte	Cycles per block	Data Memory byte
1	3.4	117.65	7466	132
2	4.2	95.24	6095	256
4	7.2	55.56	3556	512
8	11.6	34.48	2207	4096
$12-12-8$	12.4	32.26	2065	33792

Table 6: Cost of ABC setup procedures in processor cycles for ARM7TDMI

w	Key setup with precomputation	IV setup	Table precomputation
1	17300	2250	0
2	15450	1950	1350
4	19450	1200	12000
8	166150	850	163950
$12-12-8$	1915000	800	1913000

Software implementation for 64-bit processors

The ABC cipher can be efficiently implemented on 64-bit processors such as Intel Itanium or PowerPC G5. Moreover, a natural extension of the digit capacity of its primitives can lead to a more secure and efficient cipher. For example, let A be a 128-bit word-oriented LFSR, B be a 64-bit mapping of the form (2), and C be a 64-bit lookup based filter of the form (4). Then the resulting cipher will have a longer period, at least $2^{64}(2^{127}-1)$, than its 32-bit prototype, a higher linear complexity, exceeding 2^{63}, and will allow a faster implementation on a 64-bit processor, assuming w has been adjusted with the amount of available cache memory. The changes suggested here lead to a *different* stream cipher. This demonstrates that our solutions are very flexible and can be easily adjusted to specific needs of designers without worsening the cryptographical properties of the cipher.

Software implementation for 8-bit microprocessors

Stream ciphers are particularly interesting in constrained environments like smart cards and newly emerging sensor networks. In such devices memory, code size and computational power are very limited to provide security features. The ABC cipher with its flexibility to tradeoff between precomputation and performance is ideally suited for such environments.

We have implemented an optimized assembly implementation of the ABC cipher on a generic 8051 micro-controller. The measurements were done using the Keil μVision2 tools using the Philips 80/87C51 microcontroller belonging to the MCS-51 family. It is a 8051 based CMOS controller with 32 I/O Lines, 3 Timers/Counters, 6 Interrupts/4 Priority Levels, 4K Bytes ROM/OTP, 128 Bytes on-chip RAM. Though the ABC cipher is optimized for a 32-bit architecture, its implementation on 8-bit architectures is not constrained in any way. We have implemented here the version with 2-bit and 4-bit wide precomputation. The results below show that ABC cipher is suited also for very constrained environments.

Table 7: Performance of ABC key stream generation for 8-bit implementation

Precomp	Code Size byte	Const byte	Ram Size IDATA+XRAM,byte	Clock cycles per byte
2-bit wide	1372	264	57+452	241
4-bit wide	1216	520	57+708	162

Table 8: Cost of ABC setup procedures for 8-bit implementation

Procedure	Clock cycles	
	w=2-bit	w=4-bit
Key setup(incl. precomp)	54775	63383
IV setup	5202	3622

Conclusion

In this paper we presented ABC – a fast flexible synchronous stream cipher for software applications. ABC advantages are high performance, provable security properties and extremely high flexibility. The cipher could serve as an example of special design techniques, of which the mathematical background has being developed since early 90-th (see [1]–[7]), and which exploit some ideas of p-adic dynamical systems theory.

An underlying mathematical theory is a p-adic theory of certain mappings, which were introduced to crypto community in 2002 under the name of T-functions (see [14]), and which have been studied in mathematics since early 70-th. Actually, the ABC cipher utilizes some T-functions that were proved to have the single cycle property (see [1], [2]) in 1993, i.e., nearly ten years before the notion 'T-function' was invented.

The key stream generator of ABC is counter-dependent; that is, both its state transition and output (filter) functions are being modified dynamically during the encryption. The notion of a counter-dependent generator was originally introduced in [20]. We use this notion in a broader sense: In ABC not only the state transition function, but also the output function is being modified while encrypting. Moreover, our techniques provide the long period, uniform distribution, and high linear complexity of output sequences; cf. [20], where the diversity is guaranteed only.

References

[1] V. S. Anashin. *Uniformly distributed sequences over p-adic integers.* In: Number theoretic and algebraic methods in computer science. Proceedings of the Int'l Conference (Moscow, June–July, 1993) (A. J. van der Poorten, I. Shparlinsky and H. G. Zimmer, eds.), World Scientific, 1995, pp. 1–18. 26, 30, 31

[2] V. S. Anashin. *Uniformly distributed sequences over p-adic integers*, Mat. Zametki, vol. 55 (1994), no. 2, 3–46 (in Russian; English transl. in Mathematical Notes, vol. 55 (1994), no. 2, pp. 109–133.) 26, 30, 31, 35

[3] V. S. Anashin *Uniformly distributed sequences in computer algebra, or how to construct program generators of random numbers*, J. Math. Sci. (Plenum Publishing Corp., New York), **89** (1998), No 4, 1355 – 1390. 30

[4] V. S. Anashin. *Uniformly distributed sequences of p-adic integers, II*, (Russian) Diskret. Mat. vol. 14 (2002), no. 4, pp. 3–64; English translation in Discrete Math. Appl. vol. 12 (2002), no. 6, pp. 527–590. A preprint in English available from http://arXiv.org/math.NT/0209407 30, 31

[5] V. S. Anashin. *On finite pseudorandom sequences.* In: Kolmogorov and contemporary mathematics. Abstracts of the Int'l Conference, (Moscow, 16–21 June, 2003), RAS–MSU, Moscow, 2003, pp. 382–383. 30

[6] V. Anashin. *Pseudorandom Number Generation by p-adic Ergodic Transformations*, 2004. Available from http://arXiv.org/abs/cs.CR/0401030 30, 32, 33, 36, 37

[7] V. Anashin. *Pseudorandom Number Generation by p-adic Ergodic Transformations: An addendum*, 2004. Available from http://arXiv.org/abs/cs.CR/0402060 26, 30

[8] A. Biryukov and A. Shamir, *Cryptanalytic Time/Memory/Data Tradeoffs for Stream Ciphers*, in:*Asiacrypt 2000*, Lect. Notes in Comp. Sci., Vol. 1976, Springer-Verlag, 2000, pp.1–13 21

[9] C. Carroll, A. Chan and M. Zhang *The Software-Oriented Stream Cipher SSC2*, In Proceedings of Fast Software Encryption - FSE 2000, LNCS volume 1978, 2001 5

[10] C. De Cannière, J. Lano and B. Preneel, *Comments on the Rediscovery of Time Memory Data Tradeoffs.* Available from http://www.ecrypt.eu.org/stream/TMD.pdf 21

[11] D. Coppersmith, S. Halevi and C. Jutla, *Scream: a software-efficient stream cipher*, Cryptology ePrint Archive, Report 2002/019, 2002. Available from http://eprint.iacr.org/ 14

[12] Diehard Battery of Tests of Randomness v0.2 beta. Available from http://www.cs.hku.hk/~diehard/ 39

[13] J. Hong and P. Sarkar, *Rediscovery of Time Memory Tradeoffs*, Cryptology ePrint Archive, Report 2005/090, 2005. Available from http://eprint.iacr.org/ 21

[14] A. Klimov and A. Shamir. *A new class of invertible mappings*, in: *Cryptographic Hardware and Embedded Systems 2002* (B.S.Kaliski Jr.et al., eds.), Lect. Notes in Comp. Sci.,Vol. 2523, Springer-Verlag, 2003, pp.470–483. 26, 30

[15] A. Klimov and A. Shamir. *Cryptographic applications of T-functions*, in: *Selected Areas in Cryptography -2003* 30

[16] A. Klimov and A. Shamir, *New Cryptographic Primitives Based on Multiword T-functions*, in: *Fast Software Encryption -2004, 11th Int'l Workshop*. Lect. Notes Comp. Sci., Vol. 3017, Springer-Verlag, 2004, pp. 1–15 30

[17] D. Knuth. *The Art of Computer Programming. Vol. 2: Seminumerical Algorithms*, (Third edition) Addison-Wesley, Reading M.A. 1998. 36

[18] A. Menezes, P. van Oorshot and S. Vanstone, *Handbook of Applied Cryptography*, CRC Press, 1996. 10

[19] NIST Statistical Test Suite, Version 1.8. Available from http://csrc.nist.gov/rng/sts-1.8.zip 39

[20] A. Shamir and B. Tsaban, *Guaranteeing the diversity of number generators*, Information and Computation, vol. 171 (2001), pp. 350–363. Available from http: //arXiv.org/ abs/ cs.CR/ 0112014 26, 29

[21] J. Soto, L. Bassham, *Randomness Testing of the Advanced Encryption Standard Finalist Candidates*, NIST IR 6483. Available from http://csrc.nist.gov/rng/aes-report-final.doc 14, 21

[22] J. Soto, *Randomness Testing of the Advanced Encryption Standard Candidate Algorithms*, NIST IR 6390. Available from http://csrc.nist.gov/rng/AES-REPORT2.doc 21

A Design rationale

Preliminaries

Basically, ABC is a counter-dependent pseudorandom generator. An (ordinary) generator usually could be thought of as a finite automaton $\mathfrak{A} = \langle \mathcal{N}, \mathcal{M}, f, C, u_0 \rangle$ with a finite state set $\mathcal{N}$, state transition function $f : \mathcal{N} \to \mathcal{N}$, finite output alphabet $\mathcal{M}$, output function $C : \mathcal{N} \to \mathcal{M}$ and an initial state (seed) $u_0 \in \mathcal{N}$. Thus, the generator produces a sequence

$$\mathcal{C} = \{c(u_0), C(f(u_0)), C(f^{(2)}(u_0)), \ldots, C(f^{(j)}(u_0)), \ldots\}$$

over the set $\mathcal{M}$, where

$$f^{(j)}(u_0) = \underbrace{f(\ldots f(}_{j \text{ times}} u_0) \ldots) \quad (j = 1, 2, \ldots); \quad f^{(0)}(u_0) = u_0.$$

Automata of the form $\mathfrak{A}$ could be used either as pseudorandom generators per se, or as components of more complicated pseudorandom generators, the so called *counter-dependent generators*. The latter produce sequences $\{y_0, y_1, y_2, \ldots\}$ over $\mathcal{M}$ according to the rule

$$y_0 = C_0(u_0), u_1 = f_0(u_0); \ldots y_i = C_i(u_i), u_{i+1} = f_i(u_i); \ldots \qquad (20)$$

That is, at the $(i+1)$-th round the automaton $\mathfrak{A}_i = \langle \mathcal{N}, \mathcal{M}, f_i, C_i, u_i \rangle$ is applied to the state $u_i \in \mathcal{N}$, producing a new state $u_{i+1} = f_i(u_i) \in \mathcal{N}$, and outputting $y_i = C_i(u_i) \in \mathcal{M}$.

The notion of a counter-dependent generator was originally introduced in [20]. We use this notion in a broader sense: In ABC not only the state transition function, but also the output function depends on i. Moreover, our constructions provide long period, uniform distribution, and high linear complexity of output sequences; cf. [20], where only the diversity is guaranteed.

Usually the dynamical change of the *clock state transition function* f_i (as well as of the *clock output function* C_i) of a counter-dependent generator is controlled by another generator, which in turn could be either an ordinary, or a counter-dependent one. So a counter-dependent generator could be implemented as a cascaded scheme. In the ABC stream cipher the number of cascades is 2.

Notation and terminology

Further throughout this section we assume that $\mathcal{N} = \mathbb{I}_n = \{0, 1, \ldots, 2^n - 1\}$, $\mathcal{M} = \mathbb{I}_m$ for suitable natural numbers m and n, $m \le n$. It is convenient to think of elements $z \in \mathbb{I}_n$ as base-2 expansions of rational integers:

$$z = \delta_0(z) + \delta_1(z) \cdot 2 + \cdots + \delta_{n-1}(z) \cdot 2^{n-1};$$

here $\delta_j(z) \in \{0,1\}$. We usually identify $\mathbb{I}_n$ with the ring $\mathbb{Z}/2^n\mathbb{Z}$ of residues modulo 2^n or with the vector space $\mathbb{V}_n$ of dimension n over a field of two elements. The latter also could be thought of as the set $\mathbb{B}^n$ of all n-bit words.

We construct both the state transition and the output functions of the generators as composition of mappings, some of which are of the following form:

$$F\colon (\alpha_0^{\downarrow}, \alpha_1^{\downarrow}, \alpha_2^{\downarrow}, \ldots) \mapsto (\Phi_0(\alpha_0^{\downarrow}), \Phi_1(\alpha_0^{\downarrow}, \alpha_1^{\downarrow}), \Phi_2(\alpha_0^{\downarrow}, \alpha_1^{\downarrow}, \alpha_2^{\downarrow}), \ldots).$$

Here $\alpha_i^{\downarrow} \in \mathbb{B}^m$ is a Boolean columnar m-dimensional vector; $\mathbb{B} = \{0,1\}$; $\Phi_i\colon (\mathbb{B}^m)^{(i+1)} \to \mathbb{B}^r$ maps $(i+1)$ Boolean columnar m-dimensional vectors $\alpha_0^{\downarrow}, \ldots, \alpha_i^{\downarrow}$ to the r-dimensional Boolean vector $\Phi_i(\alpha_0^{\downarrow}, \ldots, \alpha_i^{\downarrow})$, $i = 0, 1, 2, \ldots, n-1$. These mappings were introduced to crypto community in 2002 (see [14]) as (m-variate) *T-functions* on n-bit words.

In fact, in mathematics these mappings have been studied at least since the beginning of 70-th: They are known in the theory of Boolean functions under the name of *triangle mappings*, in automata theory they are studied as *determined functions*, in general algebra these are referred to as *compatible functions* on the residue ring $\mathbb{Z}/2^n$. In early 90-th deep connections between these mappings and 2-adic continuous functions were understood; the connections resulted in usage of the non-Archimedean analysis machinery in study of these functions (see [1], [2]). So a significant part of results of works [14], [15], and [16] is not new for mathematicians.[2]

The most important for cryptography are such properties of T-functions as invertibility (that is, bijectivity), single cycle property, and balance. Recall that the mapping $f\colon \mathcal{A} \to \mathcal{B}$ of $\mathcal{A}$ onto $\mathcal{B}$ is called *balanced* iff the cardinality of each preimage $f^{-1}(x)$ does not depend on x; that is, $|f^{-1}(x)| = |f^{-1}(y)|$ for every pair $x, y \in \mathcal{B}$. In case $|\mathcal{A}| = |\mathcal{B}|$, the balanced mapping is bijective, so it is called *invertible*. For $\mathcal{A} = \mathcal{B}$ the invertible mapping f is a permutation on $\mathcal{A}$; in case this permutation is a cycle of length $|\mathcal{A}|$, f is said to have *a single cycle property*.

It worth noticing here that namely these properties (i.e., balance, invertibility and single cycle property) have been studied since early 90-th with the use of techniques of non-Archimedean analysis, under the names of measure preservation and ergodicity, see [1],[2],[3], [4], [5], [6], and [7] for details. Results of these works have direct applications to the subject since they give conditions for T-functions to be balanced, invertible, or to have a single cycle property.

Obviously, any m-variate T-function F could be considered as a mapping from a Cartesian product $(\mathbb{Z}/2^n\mathbb{Z})^m$ into a Cartesian product $(\mathbb{Z}/2^n\mathbb{Z})^r$, since one could identify n-bit words (rows of the matrix $(\alpha_0^{\downarrow}\alpha_1^{\downarrow}\ldots\alpha_{n-1}^{\downarrow})$)

[2] The observation that a significant part of basic microprocessor instructions (arithmetic and bitwise logical ones), and hence, all their compositions, are T-functions, also goes back to 1993, see [1], [2].

with numbers represented in their base-2 expansions in inverse bit order, less significant bits left.

Note that then any m-variate T-function F on n-bit words defines a unique m-variate T-function $\bar{F}_k$ on k-bit words for any $k \le n$: One just truncate $n-k$ more significant bits of each n-bit word. It is obvious that $\bar{F}_k$ is balanced (resp., invertible, or with a single cycle property) whenever F is balanced (resp., invertible, or with a single cycle property). So we can speak of these properties modulo some 2^k, since the truncation of $n-k$ most significant bits of an n-bit word is merely a reduction of the corresponding integer modulo 2^k. Further for short we say that the T-function F is, respectively, *balanced*, *bijective*, or *transitive* modulo 2^k meaning the function $\bar{F}_k$ is, respectively, balanced, invertible, or has a single cycle property. Often it turns out that these properties hold modulo 2^n for *every* $n = 1, 2, \ldots$ whenever they hold modulo 2^k for *some* k. [3]

Some properties of T-functions

The following example of a univariate T-function with a single cycle property, which was published in 1993, i.e., nearly 10 years before the term 'T-function' was introduced by Klimov and Shamir.

Proposition 1 (see [1, Proposition 4.8], [2, Proposition 4.12])[4] *The T-function $h(x) = a_0 + b_1 \cdot (x \oplus a_1) + b_2 \cdot (x \oplus a_2) + \cdots$ has a single cycle property iff it is transitive modulo* 4. (Here and further $\oplus$ stands for XOR, a bitwise addition modulo 2.)

Our construction of the ABC stream cipher uses a T-function that is a special case of those described by proposition 1; namely, a T-function B defined by (2). The proof of the above criterion for this particular case could be obtained with the use of more than 30 years old folklore theorem, which concerns the so-called triangle Boolean mappings, and which completely describe univariate T-functions that are invertible, or have a single cycle property, in terms of algebraic normal forms. We recall this theorem further (see theorem 1 below) while giving a proof that our T-function B of equation (2) has a single cycle property.

In the sequel we need one more notion: For arbitrary sequence $\mathcal{V} = \{v_0, v_1, \ldots\}$ of n-bit words we call the sequence $\mathcal{V}_j = \delta_j(\mathcal{V}) = \{\delta_j(v_i) \colon i = 0, 1, 2, \ldots\}$ the j-th *coordinate sequence* of the sequence $\mathcal{V}$. That is, the j-th

[3] This could be naturally explained within 2-adic analysis, since T-functions are approximations of continuous functions on the space $\mathbb{Z}_2$ of 2-adic integers, see [1], [2] and [4] for details

[4] The original proof of Proposition 1 used 2-adic analysis techniques. In fact, this proposition served just an illustration to the interpolation series technique (the technique determines whether a given T-function is invertible/has a single cycle property); the technique was originally introduced in 1993, see [1] and [2].

coordinate sequence is just a bit sequence obtained by reading each j-th bit of the outputted word. The following obvious proposition holds.

Proposition 2 *Whenever f is a T-function with a single cycle property, the j-th coordinate sequence of the sequence $x_0, x_1 = f(x_0), x_2 = f(x_1), \ldots$ is purely periodic, and the length of it shortest period is 2^{j+1}.*

Our goal is to construct counter-dependent automata that produce uniformly distributed sequences of long period. Obviously, if the state transition function f of the automaton $\mathfrak{A}$ is transitive on the state set $\mathcal{N}$, i.e., if f is a permutation with a single cycle of length $|\mathcal{N}|$; if, further, $|\mathcal{N}|$ is a multiple of $|\mathcal{M}|$, and if the output function $C : \mathcal{N} \to \mathcal{M}$ is balanced (in particular, is bijective), then the output sequence $\mathcal{C}$ of the automaton $\mathfrak{A}$ is purely periodic with a period of length $|\mathcal{N}|$ (i.e., maximum possible), and each element of M occurs at the period the same number of times: $\frac{|\mathcal{N}|}{|\mathcal{M}|}$ exactly. That is, the output sequence $\mathcal{C}$ is uniformly distributed.

Further in the section we call a sequence $\mathcal{C} = \{y_i \in \mathcal{M}\}$ over a finite set $\mathcal{M}$ purely periodic with a period of length t iff $y_{i+t} = y_i$ for all $i = 0, 1, 2, \ldots$. The sequence $\mathcal{C}$ is called *strictly uniformly distributed* iff it is purely periodic with a period of length t, and every element of $\mathcal{M}$ occurs at the period the same number of times, i.e., exactly $\frac{t}{|\mathcal{M}|}$.

State sequences of counter-dependent generators

In [6, Theorem 4.10] there were established general conditions a counter-dependent generator must meet to produce a sequence of states of a maximum possible period length. We need a special case of that theorem.

Proposition 3 (see [6, Proposition 4.6]) *Let $M > 1$ be an odd number, and let $\{h_0, \ldots, h_{M-1}\}$ be a finite sequence of T-functions with a single cycle property; let further $\{c_0, \ldots, c_{M-1}\}$ be a finite sequence of integers such that*

1. $\sum_{j=0}^{M-1} c_j \equiv 0 \pmod 2$, *and*
2. *the sequence $\{c_{i \bmod M} \bmod 2 \colon i = 0, 1, 2, \ldots\}$ is purely periodic with the shortest period of length M.*

Put $f_j(x) = c_j \oplus h_j(x)$ (respectively, $f_j(x) = c_j + h_j(x)$). Then the recurrence sequence $\mathcal{Z}$ defined by the relation $x_{i+1} = f_{i \bmod M}(x_i)$ is strictly uniformly distributed modulo 2^r, $r = 1, 2, \ldots$: That is, modulo 2^r the sequence $\mathcal{Z}$ is purely periodic, its shortest period is of length $M \cdot 2^r$, and each element of $\mathbb{Z}/2^r\mathbb{Z}$ occurs at the period exactly M times.

Note 1 *It is obvious that the sequence of states of a linear feedback shift register with a primitive polynomial of degree s (and with a non-zero initial state c_0), which could be considered as a sequence $\{c_0, \ldots, c_{M-1}\}$ over integers in their base-2 expansions, satisfy conditions of Proposition 3 for $M = 2^s - 1$.*

The sequence of states of a counter-dependent generator that satisfies conditions of Proposition 3 suffers the weakness similar to that of any T-function does (see Proposition 2):

Proposition 4 (see [6, Theorem 5.6]) *Under conditions of Proposition 3 the j-th coordinate sequence $\mathcal{Z}_j$ of the sequence $\mathcal{Z}$ of Proposition 3 is purely periodic, and the length of its shortest period is $2^{j+1} \cdot t$, where $1 \le t \le M$.*

Constructions

Now we describe a general construction of a counter-dependent generator of the ABC family. The generator is controlled by a linear feedback shift register, and produces uniformly distributed sequences of a maximum possible period length. Basically the construction could be described as follows.

Let A be a linear transformation of the vector space $\mathbb{V}_n$ of dimension n over a field $GF(2) = \mathbb{Z}/2\mathbb{Z}$, let the characteristic polynomial (of degree n) of A be primitive. We use linear recurrence sequence $\{z_{i+1} = A(z_i)\}$ as a sequence that controls modifications of the clock state transition functions f_i in the following way: $f_i(x_i) \equiv \tilde{z}_i + h_i(x_i) \pmod{2^k}$, where h_i are T-functions on k-bit words with a single cycle property of proposition 3, and $\tilde{z}_i$ is a k-bit part of the n-bit word z_i (in the current version of ABC we assume $n = 2k - 1 = 63$, $k = 32$). Note that since the characteristic polynomial of A is primitive, and its degree is n, the sequence $\{\tilde{z}_i\}$ is purely periodic, and the length of its shortest period is $2^n - 1$, *despite of which k bits of the n-bit state of the LFSR we choose to form $\tilde{z}_i$.*[5]

Mappings h_i satisfy conditions of Proposition 3: In ABC we assume all these to be equal to the same T-function B of equation (2) that satisfies proposition 1; that is, B has a single cycle property. Note that then in view of Proposition 3 and note 1 *the sequence of states of the ABC stream cipher formes a cycle of length $2^k \cdot (2^n - 1)$. The cycle is totally determined by the LFSR A and the function B: Pairwise distinct functions B* (that correspond to distinct pairs $d_0, d_1 \in \mathbb{Z}/2^k\mathbb{Z}$) *determine pairwise distinct cycles. A pair $(z; x)$ of initial states $z \in (\mathbb{Z}/2^n\mathbb{Z})\setminus\{0\}$ (of the LFSR A) and $x \in \mathbb{Z}/2^k\mathbb{Z}$ (of the function B) determine a unique initial position at the cycle.*

Now we choose clock output functions to guarantee that the length of the *shortest* period of the output sequence is also $(2^n - 1) \cdot 2^k$, i.e., the maximum possible.

[5] Of course, the numbers $j_0 < \cdots < j_{k-1}$ of bits in z_i we choose to form $\tilde{z}_i$ must be fixed, i.e., must not depend on i.

Choosing output functions for ABC family

In ABC stream cipher clock output functions are of the form $C_i(x) = \check{c}_i + (S(x) \ggg t) \bmod 2^k$, where

$$S(x) = e + \sum_{j=0}^{k-1} e_j \cdot \delta_j(x), \tag{21}$$

Actually, in the ABC stream cipher we assume that $C_i(x) = \bar{z}_0 + C(x) \pmod{2^k}$, where $\bar{z}_0$ is the low order k-bit word of the output of the LFSR A at the i-th round (thus, $\bar{z}_0 = \bar{z}_{i,0}$ depends on i), and

$$C(x) = (S(x) \ggg t).$$

Note here that C (whence, C_i) are not T-functions. However, the following proposition holds.

Proposition 5 *Whenever $e_{k-1} \equiv 2^t \pmod{2^{t+1}}$, the length of the shortest period of the key-stream sequence $\{C_i(x_i)\}$ is exactly $(2^n - 1) \cdot 2^k$, and the linear complexity λ of this sequence $> 2^{k-1}$.*

Actually, it could be proved that $2^{k-1} \cdot (2^n - 1) + 1 \geq \lambda \geq 2^{k-1} + 1$. Our experiments with a reduced model of the cipher (with reduced bit lengths of variables) show that the lower bound is too pessimistic: In all cases we obtained values of the linear complexity close to the upper bound.

Henceforth, in view of Proposition 5, *we assume that $t = \frac{k}{2} = 16$ and $e_{31} \equiv 2^{16} \pmod{2^{17}}$ in the current version of the ABC stream cipher.* This restriction also guarantees that the distribution of the key stream is close to uniform:

Proposition 6 *For $a \in \mathbb{Z}/2^k\mathbb{Z}$ let $\mu(a)$ be the number of occurrences of a k-bit word a at the shortest period* (which is of length $P = (2^n - 1) \cdot 2^k$ since we assume $e_{k-1} \equiv 2^{\frac{k}{2}} \pmod{2^{\frac{k}{2}+1}}$, see Proposition 5) *of the key stream sequence $\{C_i(x_i)\}$. Then*

$$\left| \frac{\mu(a)}{P} - \frac{1}{2^k} \right| < \frac{1}{\sqrt{P}}.$$

Note that for truly random sequence of length P of k-bit words with probability $> 1 - \frac{1}{2^k}$ one has $\left| \frac{\mu(a)}{P} - \frac{1}{2^k} \right| < \frac{1}{\sqrt{P}}$.

B Proofs and extra results

B.1 Proof of the special case of Proposition 1

We are going to prove that *a T-function $h(x) = a + b_1 \cdot (x \oplus a_1)$ has a single cycle property whenever simultaneously $a \equiv 1 \pmod 2$, $b_1 \equiv 1 \pmod 4$,*

and $a_1 \equiv 0 \pmod 4)$. To do this, we need some known results. The first of these is *a folklore criteria of bijectivity/transitivity of triangle Boolean mappings.*

Recall that the *algebraic normal form*, ANF, of the Boolean function $\psi_j(\chi_0, \dots, \chi_j)$ is the representation of this function via $\oplus$ (addition modulo 2= logical 'exclusive or') and $\cdot$ (multiplication modulo 2 =logical 'and'= conjunction). In other words, the ANF of the Boolean function ψ is

$$\psi(\chi_0, \dots, \chi_j) = \beta \oplus \beta_0\chi_0 \oplus \beta_1\chi_1 \oplus \dots \oplus \beta_{0,1}\chi_0\chi_1 \oplus \dots,$$

where $\beta, \beta_0, \dots \in \{0, 1\}$. The ANF is sometimes called a *Boolean polynomial.*

The number of factors in the longest monomial $\chi_{j_0}\chi_{j_1}\dots$ with a non-zero coefficient $\beta_{j_0,j_1,\dots}$ is called a (full) *degree* of the function ψ; Boolean functions of a (full) degree 1 are called *linear.* The degree of the functions with respect to a variable χ_j is the number of terms in the longest monomial that contains χ_j. Recall that the *weight* of the Boolean function ψ_j in $(j+1)$ variables is the number of $(j+1)$-bit words that *satisfy* ψ_j; that is, weight is the cardinality of the truth set of ψ_j.

The following theorem is known at least 30 years; however, since it is a folklore, we can not attribute it, yet an interested reader could find a proof in, e.g., [2, Lemma 4.8].

Theorem 1 *A univariate T-function F*

$$(\chi_0, \chi_1, \chi_2, \dots) \overset{F}{\mapsto} (\psi_0(\chi_0); \psi_1(\chi_0, \chi_1); \psi_2(\chi_0, \chi_1, \chi_2); \dots),$$

is invertible iff for each $j = 0, 1, \dots$ the Boolean function ψ_j in Boolean variables $\chi_0, \dots, \chi_j$ is linear with respect to the variable χ_j; that is, F is invertible iff the ANF of each ψ_j is of the form

$$\psi_j(\chi_0, \dots, \chi_j) = \chi_j \oplus \varphi_j(\chi_0, \dots, \chi_{j-1}),$$

where φ_j is the Boolean function that does not depend on the variable χ_j. The mapping F has a single cycle property iff, additionally, the Boolean function φ_j is of odd weight. The latter takes place if and only if $\varphi_0 = 1$, and the full degree of the Boolean function φ_j for $j \geq 1$ is exactly j, that is, the ANF of φ_j contains a monomial $\chi_0 \cdots \chi_{j-1}$. Thus, F has a single cycle property iff $\psi_0(\chi_0) = \chi_0 \oplus 1$, and for $j \geq 1$ the ANF of each ψ_j is of the form

$$\psi_j(\chi_0, \dots, \chi_j) = \chi_j \oplus \chi_0 \cdots \chi_{j-1} \oplus \theta_j(\chi_0, \dots, \chi_{j-1}),$$

where the weight of θ_j is even; i.e., $\deg \theta_j \leq j - 1$.

With the use of this theorem the following result was obtained:

Proposition 7 ([6, Proposition 3.15]) *For any T-function F with a single cycle property and for arbitrary T-function V the following T-functions have a single cycle property: $F(x + 4 \cdot V(x))$, $F(x \oplus (4 \cdot V(x)))$, $F(x) + 4 \cdot V(x)$, and $F(x) \oplus (4 \cdot V(x))$.* Here $\oplus$ stands for a XOR instruction; that is, for a bitwise addition modulo 2 (i.e., without a carry to more significant bits) of n-bit words.

In view of this Proposition 7 the statement at the beginning of this subsection becomes obvious: Indeed, $h(x) = a+b_1 \cdot (x+a_1)$ is a composition of a linear congruential generator $H(x) = a + b_1 \cdot x$, which has a single cycle property since $a \equiv 1 \pmod 2$ and $b_1 \equiv 1 \pmod 4$ (for a well-known transitivity criterion for linear congruential generators see e.g. [17]), with a mapping $x \mapsto x \oplus b_1$, where $b_1 = 4r$ for a suitable r; now just apply Proposition 7.

B.2 Proof of Proposition 5

Indeed, in view of Proposition 4, the length of the shortest period of the $(k-1)$-th coordinate sequence $\{\delta_{k-1}(x_i)\}$ of internal states is $2^k \cdot t_0$ for some integer $t_0 > 0$; that is, the length of the shortest period of the sequence $\{e_{31} \cdot \delta_{k-1}(x_i)\}$ of k-bit words is also is $2^k \cdot t_0$, whereas the length of the shortest period of any other sequence $\{e_j \cdot \delta_j(x_i)\}$ for $j = 0, 1, 2, \dots, k-1$ is $2^j \cdot t_j$ for a suitable $t_j > 0$ (or might be 1 in case $e_j = 0$). Moreover, since the length of shortest period of the j-th coordinate sequence $\{\delta_j(x_i)\}$ is a divisor of the length of the shortest period of the whole sequence $\{x_i\}$ (which is $(2^n - 1) \cdot 2^k$ in view of Proposition 3 and note 1), all r_j are divisors of $(2^n - 1)$, i.e., odd, $j = 0, 1, 2, \dots, k-1$. Hence, the length of the shortest period of the sequence

$$\left\{\delta_t\Big(e + \sum_{j=1}^{k-1} e_j \cdot \delta_j(x_i)\Big) : i = 0, 1, 2, \dots\right\}$$

is a divisor of $2^{k-1} \cdot (2^n - 1)$. Now, as $\delta_t(e_{k-1}) = 1$, and $\delta_j(e_{k-1}) = 0$ for $j = 0, 1, \dots, t-1$, we conclude that the length of the shortest period of the bit sequence $\{\delta_0(C(x_i))\}$ is $2^k \cdot r$ for a suitable $r > 0$, since

$$\delta_0(C(x_i)) \equiv \delta_t\Big(e + \sum_{j=0}^{k-2} e_j \cdot \delta_j(x_i)\Big) + \delta_t(e_{k-1}) \cdot \delta_{k-1}(x_i) \pmod 2.$$

Yet then the length of the shortest period of the sum $\{C(x_i) \oplus \bar{z}_{i,0} \pmod 2\}$ of two sequences modulo 2 is at least $(2^n - 1) \cdot 2^k$. However, it can not be greater than $(2^n - 1) \cdot 2^k$. This implies that the length of the shortest period of the 0-th coordinate sequence $\mathcal{F}_0 = \{\delta_0(C_i(x_i)\}$ (and whence of the whole output sequence $\mathcal{F} = \{C_i(x_i)\}$) is exactly $(2^n - 1) \cdot 2^k$.

To prove that linear complexity over a field $\mathbb{Z}/2$ of the output sequence $\mathcal{F}$ exceeds 2^{k-1} it is sufficient to demonstrate that linear complexity of the 0-th coordinate sequence $\mathcal{F}_0$ exceeds 2^{k-1}.

It was shown above that the length of the shortest period of the sequence $\mathcal{F}_0$ is $2^k \cdot (2^n - 1)$. Hence the polynomial $u(X) = X^{2^k \cdot (2^n-1)} - 1 = (X^{2^n-1} - 1)^{2^k}$ is a characteristic polynomial of the sequence $\mathcal{F}_0$. Thus, the minimal polynomial $m(X)$ of the sequence $\mathcal{F}_0$ is a factor of $u(X)$. On the other hand, $m(X)$ is not a factor of $w(X) = (X^{2^n-1} - 1)^{2^{k-1}}$ since otherwise the sequence has a period of length $2^{k-1} \cdot (2^n - 1)$. Since the both polynomials $u(X)$ and $w(X)$ have in their common splitting field the same set of roots (namely, $2^n - 1$ pairwise distinct roots of a unity of degree $2^n - 1$), at least one of these roots must be a root of $m(X)$ with multiplicity exceeding 2^{k-1}. Thus, $\deg m(X) > 2^{k-1}$.

B.3 Proof of Proposition 6

We will use the following lemma, which originally was used in the proof of [6, Theorem 4.10].

Lemma 1 (see [6, Lemma 4.7]) *Let $c_0, \ldots, c_{M-1}$ be a finite sequence of integers, and let $f_0, \ldots, f_{M-1}$ be a finite sequence of T-functions such that*

i. $f_j(x) \equiv x + c_j \pmod 2$ *for* $j = 0, 1, \ldots, M-1$,

ii. $\sum_{j=0}^{M-1} c_j \equiv 1 \pmod 2$,

iii. *the sequence* $\{c_{i \bmod M} \bmod 2 \colon i = 0, 1, 2, \ldots\}$ *is purely periodic, its shortest period is of length* M,

iv. $\delta_t(f_j(z)) \equiv \zeta_t + \varphi_k^j(\zeta_0, \ldots, \zeta_{t-1}) \pmod 2$, $t = 1, 2, \ldots$, *where* $\zeta_\ell = \delta_r(z)$, $\ell = 0, 1, 2, \ldots$,

v. *for each* $t = 1, 2, \ldots$ *an odd number of Boolean polynomials* φ_t^j *in Boolean variables* $\zeta_0, \ldots, \zeta_{t-1}$ *are of odd weight.*

Then the recurrence sequence $\mathcal{Y} = \{x_i\}$ defined by the relation $x_{i+1} = f_{i \bmod M}(x_i)$ is strictly uniformly distributed: It is purely periodic modulo 2^r for all $r = 1, 2, \ldots$; its shortest period is of length $M \cdot 2^r$; each element of $\mathbb{Z}/2^r\mathbb{Z}$ occurs at the period exactly M times. Moreover,

1. *the sequence* $\mathcal{D}_s = \{\delta_s(x_i) \colon i = 0, 1, 2, \ldots\}$ *is purely periodic; it has a period of length* $M \cdot 2^{s+1}$,

2. $\delta_s(x_{i+M\cdot 2^s}) \equiv \delta_s(x_i) + 1 \pmod 2$ *for all* $s = 0, 1, \ldots, k-1$, $i = 0, 1, 2, \ldots$,

3. *for each* $m = 1, 2, \ldots, r$ *and each* $\ell = 0, 1, 2, \ldots$ *the sequence*

$$x_\ell \bmod 2^m, x_{\ell+M} \bmod 2^m, x_{\ell+2M} \bmod 2^m, \ldots$$

is purely periodic, its shortest period is of length 2^m*, each element of* $\mathbb{Z}/2^m\mathbb{Z}$ *occurs at the period exactly once.*

In fact, the lemma says that the sequence $\{x_i\}$ of states of a counter-dependent generator satisfies conditions 1–3 whenever f_j are of Proposition 3.

Now for $i = 0, 1, \ldots, 2^n - 2 = M - 1$ denote $\nu_i(a)$ the number of occurrences of a k-bit word $a \in \mathbb{Z}/2^k\mathbb{Z}$ at the period of length 2^k of the sequence

$$\mathcal{S}^{(i)} = \{C(x_{i+M\cdot t}) \pmod{2^k} \colon t = 0, 1, 2, \ldots\}.$$

In view of statement 3 of lemma 1 we conclude that $\nu_i(a) = \nu_j(a) = \tilde{\nu}(a)$ for $i, j \in \{0, 1, \ldots, M-1\}$.

Now consider a subsequence

$$\mathcal{Y}^{(i)} = \{C(x_{i+M\cdot t}) + \bar{z}_{i+M\cdot t,0} \pmod{2^k} \colon t = 0, 1, 2, \ldots\}$$

of the output sequence of our generator. Note that $\bar{z}_{i+M\cdot t,0} = \bar{z}_{i,0}$ since the length of the period of the sequence $\{\bar{z}_{i,0} \colon i = 0, 1, 2, \ldots\}$ is M; hence, the subsequence $\mathcal{Y}^{(i)}$ is periodic, and has a period of length 2^k. Now denoting $\mu_i(a)$ the number of occurrences of a k-bit word $a \in \mathbb{Z}/2^k\mathbb{Z}$ at the period of length 2^k of the sequence $\mathcal{Y}^{(i)}$, we conclude that the number of occurrences $\mu(a)$ of a at the shortest period, which is of length $M \cdot 2^k$, of the output sequence (i.e., of the key stream) of our generator is

$$\mu(a) = \sum_{i=0}^{M-1} \mu_i(a) = \sum_{i=0}^{M-1} \nu_i((a - \bar{z}_{i,0}) \bmod 2^k) = \sum_{i=0}^{M-1} \tilde{\nu}((a - \bar{z}_{i,0}) \bmod 2^k).$$

Yet in the sequence $\{\bar{z}_{i,0} \colon i = 0, 1, 2, \ldots, M-1\}$ each non-zero k-bit word occurs exactly 2^{n-k} times, whereas a zero word occurs $2^{n-k} - 1$ times. Thus, in view of the above equality we conclude that

$$\mu(a) = 2^{n-k} \cdot \sum_{w \in \mathbb{B}^k} \tilde{\nu}(w) - \tilde{\nu}(a) = 2^n - \tilde{\nu}(a)$$

since $\sum_{w \in \mathbb{B}^k} \tilde{\nu}(w) = 2^k$ in view of statement 3 of lemma 1. Now easy exercise in inequalities finishes the proof.

C Statistical properties of key and IV setup procedure

ABC key setup and IV setup algorithms employ the same mechanism of warm-up with feedback. Here we present an empirical statistical evaluation

of propagation properties of this mechanism for the procedures described in Section 4.

An extreme case of single-bit change was chosen for assessment, concerning implementations that use simple counter to produce IV value or cases of improper key management. Empirical statistical testing shows that the procedures provide a significant impact of minor ABC state change upon generated key stream.

Let $(iv, \hat{iv})$ be a pair of 128-bit initial values, where $\hat{iv}$ differs from iv in one arbitrary bit position, and $(s, \hat{s})$ be a pair of corresponding L-bit key stream vectors. That is, s is obtained directly after setup of some key and iv, and $\hat{s}$ is obtained directly after setup of same key and $\hat{iv}$. $H(u, v)$ denotes Hamming distance between N-bit vectors u and v. Thus, $H(iv, \hat{iv}) = 1$ and $H(s, \hat{s}) \in [0, L]$.

Two targets of evaluation were chosen, each employing some statistics over a number of $(s, \hat{s})$ pairs.

Hamming distance distribution. Let us take the null-hypothesis H_0 that s and $\hat{s}$ were taken randomly and independently from $\mathrm{GF}(2)^L$. In this case the value of $H(s, \hat{s})$ would theoretically follow binomial distribution with known parameters. Key stream length $L = 32$ bits and a sample size of 10^7 $(s, \hat{s})$ pairs for a fixed key and for a key varying with each pair were chosen for simulation. In either cases obtained empirical distribution indicates high order of single-bit state change propagation to key stream.

Naive correlation. Another indicator of propagation properties is an L-bit product of bitwise XOR of k and $\hat{s}$. For a fixed key and for a key varying with each IV pair, 384 such sequences of length $L = 10^6$ were obtained and empirically evaluated. Results of NIST Statistical Test Suite ([19]) and DIEHARD Battery of Tests ([12]) application do not show any deviation from random behaviour in either case.

D ABC reference code

Here we give the reference ANSI C code of the ABC core transform iteration, key setup and IV setup procedures. Throughout this section it is assumed that $w = 8$. The precomputation of the optimization tables from the fixed initial values in Table 2 is omitted.

```
typedef unsigned long int u32;

u32 z0, z1;       /* state of A transform                       */
u32 x;            /* state of B transform                       */
u32 d0, d1;       /* coefficients of B transform                */
u32 z0i, z1i;     /* state of A transform after key setup       */
```

```
u32 xi;           /* state of B transform after key setup       */
u32 d0i, d1i;     /* coefficients of B transform after key setup */
u32 t[1024];      /* optimization table
u32 ptable[1024]; /* precomputed optimization table for key
                     setup (initialization omitted)             */

/* Keystream generator */
u32 abc_keystream(u32 *z0, u32 *z1, u32 *x, u32 d0, u32 d1, u32 *t)
{
    u32 r, s; /* local temporary variables */

    s = *z1 ^ (*z0 >> 1) ^ (*z1 << 31);
    *z0 = *z1;
    *z1 = s;
    s = *x ^ d1;
    *x = *z1 + d0 + s + (s << 2);
    s = *x;
    r = t[s & 0xff];
    s >>= 8;
    r += t[256 + (s & 0xff)];
    s >>= 8;
    r += t[512 + (s & 0xff)];
    s >>= 8;
    r += t[768 + (s & 0xff)];
    return *z0 + ((r >> 16) | (r << 16));
}

/* Key setup routine */
void abc_keysetup(const u32* key)
{
    u32 e[33];    /* C coefficients for main state */
    u32 i, r;     /* local temporary variables     */

    /* Key-dependent part of initial state */
    u32 id0 =
        0x0c376d75UL ^ ((key[3] & 0x0000ffffUL) << 2) ^ (key[1] & 2UL);
    u32 id1 = 0xbbb5b0b4UL ^ ((key[3] & 0xffff0000UL) >> 14);
    u32 ix = key[2];
    u32 iz0 = key[1] | 2UL;
    u32 iz1 = key[0];

    /* Warm up initial state with feedback */
    r = abc_keystream(&iz0, &iz1, &ix, id0, id1, ptable);
    ix ^= r;
    r = abc_keystream(&iz0, &iz1, &ix, id0, id1, ptable);
    id0 ^= r;
    id0 |= 1UL;
    r = abc_keystream(&iz0, &iz1, &ix, id0, id1, ptable);
    id1 ^= r;
```

```
id1 &= 0xfffffffcUL;
r = abc_keystream(&iz0, &iz1, &ix, id0, id1, ptable);
iz0 ^= r;
iz0 |= 2UL;
r = abc_keystream(&iz0, &iz1, &ix, id0, id1, ptable);
iz1 ^= r;

/* Fill main state */
e[32] = abc_keystream(&iz0, &iz1, &ix, id0, id1, ptable);
for (i = 0; i < 32; ++i)
    e[i] = abc_keystream(&iz0, &iz1, &ix, id0, id1, ptable);
d0 = abc_keystream(&iz0, &iz1, &ix, id0, id1, ptable);
d1 = abc_keystream(&iz0, &iz1, &ix, id0, id1, ptable);
x = abc_keystream(&iz0, &iz1, &ix, id0, id1, ptable);
z0 = abc_keystream(&iz0, &iz1, &ix, id0, id1, ptable);
z1 = abc_keystream(&iz0, &iz1, &ix, id0, id1, ptable);

/* Apply restrictions and save changeable part of state */
z0i = z0 = z0 | 2UL;
z1i = z1;
xi = x;
d0i = d0 = d0 | 1UL;
d1i = d1 = d1 & 0xfffffffcUL;
e[31] = (e[31] & 0xffff0000UL) | 0x00010000UL;

/* Precompute optimization tables */
for(i = 0; i < 256; ++i)
{
    t[i] = e[32];
    t[i] += (i & 0x01) ? e[8 * 0 + 0] : 0;
    t[i] += (i & 0x02) ? e[8 * 0 + 1] : 0;
    t[i] += (i & 0x04) ? e[8 * 0 + 2] : 0;
    t[i] += (i & 0x08) ? e[8 * 0 + 3] : 0;
    t[i] += (i & 0x10) ? e[8 * 0 + 4] : 0;
    t[i] += (i & 0x20) ? e[8 * 0 + 5] : 0;
    t[i] += (i & 0x40) ? e[8 * 0 + 6] : 0;
    t[i] += (i & 0x80) ? e[8 * 0 + 7] : 0;
}
for(i = 0; i < 256; ++i)
{
    t[256 + i] = 0;
    t[256 + i] += (i & 0x01) ? e[8 * 1 + 0] : 0;
    t[256 + i] += (i & 0x02) ? e[8 * 1 + 1] : 0;
    t[256 + i] += (i & 0x04) ? e[8 * 1 + 2] : 0;
    t[256 + i] += (i & 0x08) ? e[8 * 1 + 3] : 0;
    t[256 + i] += (i & 0x10) ? e[8 * 1 + 4] : 0;
    t[256 + i] += (i & 0x20) ? e[8 * 1 + 5] : 0;
    t[256 + i] += (i & 0x40) ? e[8 * 1 + 6] : 0;
    t[256 + i] += (i & 0x80) ? e[8 * 1 + 7] : 0;
```

```
    }
    for(i = 0; i < 256; ++i)
    {
        t[512 + i] = 0;
        t[512 + i] += (i & 0x01) ? e[8 * 2 + 0] : 0;
        t[512 + i] += (i & 0x02) ? e[8 * 2 + 1] : 0;
        t[512 + i] += (i & 0x04) ? e[8 * 2 + 2] : 0;
        t[512 + i] += (i & 0x08) ? e[8 * 2 + 3] : 0;
        t[512 + i] += (i & 0x10) ? e[8 * 2 + 4] : 0;
        t[512 + i] += (i & 0x20) ? e[8 * 2 + 5] : 0;
        t[512 + i] += (i & 0x40) ? e[8 * 2 + 6] : 0;
        t[512 + i] += (i & 0x80) ? e[8 * 2 + 7] : 0;
    }
    for(i = 0; i < 256; ++i)
    {
        t[768 + i] = 0;
        t[768 + i] += (i & 0x01) ? e[8 * 3 + 0] : 0;
        t[768 + i] += (i & 0x02) ? e[8 * 3 + 1] : 0;
        t[768 + i] += (i & 0x04) ? e[8 * 3 + 2] : 0;
        t[768 + i] += (i & 0x08) ? e[8 * 3 + 3] : 0;
        t[768 + i] += (i & 0x10) ? e[8 * 3 + 4] : 0;
        t[768 + i] += (i & 0x20) ? e[8 * 3 + 5] : 0;
        t[768 + i] += (i & 0x40) ? e[8 * 3 + 6] : 0;
        t[768 + i] += (i & 0x80) ? e[8 * 3 + 7] : 0;
    }
}

/* IV setup routine */
void abc_ivsetup(const u32* iv)
{
    u32 r; /* local temporary variables */

    /* Apply IV to restored state */
    d0 =
        d0i ^ ((iv[3] & 0x0000ffffUL) << 2) ^ (iv[1] & 2UL);
    d1 = d1i ^ ((iv[3] & 0xffff0000UL) >> 14);
    x = xi ^ iv[2];
    z0 = (z0i ^ iv[1]) | 2UL;
    z1 = z1i ^ iv[0];

    /* Warm up with feedback */
    r = abc_keystream(&z0, &z1, &x, d0, d1, t);
    x ^= r;
    r = abc_keystream(&z0, &z1, &x, d0, d1, t);
    d0 ^= r;
    d0 |= 1UL;
    r = abc_keystream(&z0, &z1, &x, d0, d1, t);
    d1 ^= r;
    d1 &= 0xfffffffcUL;
```

```
    r = abc_keystream(&z0, &z1, &x, d0, d1, t);
    z0 ^= r;
    z0 |= 2UL;
    r = abc_keystream(&z0, &z1, &x, d0, d1, t);
    z1 ^= r;
}
```