ABC – a New Fast Flexible Stream Cipher

Specification, Version 3

Vladimir Anashin[1], Andrey Bogdanov[2], and Ilya Kizhvatov[1]

[1] Russian State University for the Humanities,
Institute for Information Sciences and Security Technologies,
Faculty of Information Security,
Kirovogradskaya Str. 25/2, 117534 Moscow, Russia
{anashin,ilya.kizhvatov}@rsuh.ru
[2] escrypt GmbH – Embedded Security
Lise-Meitner-Allee 4, D-44801 Bochum, Germany
abogdanov@escrypt.com

1 Introduction

ABC is a synchronous stream cipher optimized for software applications. Its key length is 128 bits. It accommodates a 128-bit initial vector. Here a version of ABC with a 128-bit key and 32-bit internal variables is presented.

A new approach to the design of stream ciphers has been used which results in a cipher based upon key- and clock-dependent state transition and filter functions. More precisely, ABC combines two building blocks: a wreath product of a LFSR and a non-linear single-cycle T-function, as well as a derivation of the knapsack function. Our techniques guarantee the period of $2^{32} \cdot (2^{127} - 1)$ words, uniform distribution, and high linear complexity of the keystream of ABC. The allowed length of a single stream (for a given key/IV pair) for ABC is 2^{64} 32-bit words.

During Phase 1 of eSTREAM the previous versions of ABC ([10] and [11]) were cryptanalyzed. [13], [18] and [26] discovered successful attacks. [19] contains erroneous results that are not applicable [9] to ABC. The current version of ABC includes tweaks against all of the mounted attacks (a longer LFSR, another T-function, modified key key setup procedure). It offers a security level of 2^{128}. No hidden weaknesses have been incorporated in the design of ABC.

ABC can be efficiently implemented in software. Our C implementation encryption performance is about 4 clocks per byte on a standard 1.73 GHz Pentium M processor. The flexibility property results in the possibility of its efficient application on a variety of computer platforms by choosing proper implementation parameters.

2 Notation

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

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

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

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

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

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

z is a 128-bit integer value and allows several equivalent representations too:

$$z = (z_{127}, \ldots, z_0) = \sum_{i=0}^{127} z_i 2^i \in \mathbb{Z}/2^{128}\mathbb{Z}, \;\; z_i \in \{0,1\}, \; i = 0, \ldots, 127;$$
$$z \in \mathbb{V}_{128} = \mathrm{GF}(2)^{128};$$
$$z = (\bar{z}_3, \bar{z}_2, \bar{z}_1, \bar{z}_0) \in (\mathbb{Z}/2^{32}\mathbb{Z})^4, \quad \bar{z}_3, \bar{z}_2, \bar{z}_1, \bar{z}_0 \in \mathbb{Z}/2^{32}\mathbb{Z}.$$

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 keystream 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:

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

$d_0 = (d_{0,31}, \ldots, d_{0,0}), \; d_1 = (d_{1,31}, \ldots, d_{1,0}) \; d_2 = (d_{2,31}, \ldots, d_{2,0}) \in \mathbb{Z}/2^{32}\mathbb{Z}$ are 32-bit integer values.

Having been defined once at the initialization stage, the variables d_0, d_1, d_2, 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 need an auxiliary w-bit integer variable $j \in \mathbb{Z}/2^w\mathbb{Z}$ will be needed:

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

Finally, in the description of operations below two 32-bit integer variables are required:

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

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 128-bit integer number and can be described in the following way as applied to respectively x, z, d_1 and j:

$$\begin{aligned} &\delta_i : \mathbb{Z}/2^{32}\mathbb{Z} \to \{0,1\},\ \delta_i(x) = x_i,\ \ i = 0, \ldots, 31, \\ &\delta_i : \mathbb{Z}/2^{128}\mathbb{Z} \to \{0,1\},\ \delta_i(z) = z_i,\ \ i = 0, \ldots, 127, \\ &\delta_i : \mathbb{Z}/2^{32}\mathbb{Z} \to \{0,1\},\ \delta_i(d_1) = d_{1,i},\ \ i = 0, \ldots, 31, \\ &\delta_i : \mathbb{Z}/2^{w}\mathbb{Z} \to \{0,1\},\ \delta_i(j) = j_i,\ \ i = 0, \ldots, w-1; \end{aligned}$$

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:

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

for example,

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

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:

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

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:

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

Right rotation, $\cdot \ggg c$, denotes right bitwise 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:

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

3 Algorithm Description

This section contains the description of the ABC primitives, keystream generator and initialization routines. The considerations are given in Section 5.

3.1 Primitives

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

A: $\mathbb{Z}/2^{128}\mathbb{Z} \to \mathbb{Z}/2^{128}\mathbb{Z}$ is a linear feedback shift register of length 128 (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 $\mathbb{Z}/2^{32}\mathbb{Z}$ and bitwise addition modulo 2 (XOR), 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 bitwise rotation ($\ggg$), assuming x as argument.

A: Linear feedback shift register, counter

A is a linear transformation of the vector space $\mathbb{V}_{128} = \mathrm{GF}(2)^{128}$, $z = A(z)$, and is defined by a LFSR as follows:

$$\begin{aligned} &\zeta \leftarrow \bar{z}_2 \,\mathsf{XOR}(\bar{z}_1 \ll 31) \,\mathsf{XOR}(\bar{z}_0 \gg 1) \bmod 2^{32}, \\ &\bar{z}_0 \leftarrow \bar{z}_1, \\ &\bar{z}_1 \leftarrow \bar{z}_2, \\ &\bar{z}_2 \leftarrow \bar{z}_3, \\ &\bar{z}_3 \leftarrow \zeta. \end{aligned} \tag{1}$$

B: Single-cycle function, state transition

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

$$B(x) = ((x \,\mathsf{XOR}\, d_0) + d_1) \,\mathsf{XOR}\, d_2 \bmod 2^{32}, \tag{2}$$

where $d_0,\ d_1,\ d_2 \in \mathbb{Z}/2^{32}\mathbb{Z}$, $d_0 \equiv 0 \pmod 4$, $d_1 \equiv 1 \pmod 4$, $d_2 \equiv 0 \pmod 4$. In other words, the following equations should hold simultaneously:

$$\begin{cases} d_{0,0} = d_{0,1} = 0, \\ d_{1,0} = 1, d_{1,1} = 0, \\ d_{2,0} = d_{2,1} = 0. \end{cases} \tag{3}$$

C: Filter function, output

The filter function C is defined in the following way:

$$C(x) = S(x) \ggg 16, \tag{4}$$

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

$$S(x) = e + \sum_{i=0}^{31} e_i \delta_i(x) \bmod 2^{32}. \tag{5}$$

Here $e_{31} \equiv 2^{16} \pmod{2^{17}}$.

3.2 Keystream Generator

The keystream generation routine of ABC involves the primitives described in Subsection 3.1 and consists of 3 steps.

ABC KEYSTREAM GENERATOR

INPUT: $z \in \mathbb{Z}/2^{128}\mathbb{Z}$, $x \in \mathbb{Z}/2^{32}\mathbb{Z}$

$z \leftarrow A(z)$
$x \leftarrow \bar{z}_3 + B(x) \bmod 2^{32}$
$y \leftarrow \bar{z}_0 + C(x) \bmod 2^{32}$

OUTPUT: $z \in \mathbb{Z}/2^{128}\mathbb{Z}$, $x \in \mathbb{Z}/2^{32}\mathbb{Z}$, $y \in \mathbb{Z}/2^{32}\mathbb{Z}$

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

The maximum length of a single stream (stream generated for a single key/IV pair) is restricted to 2^{64} outputs of the keystream generator.

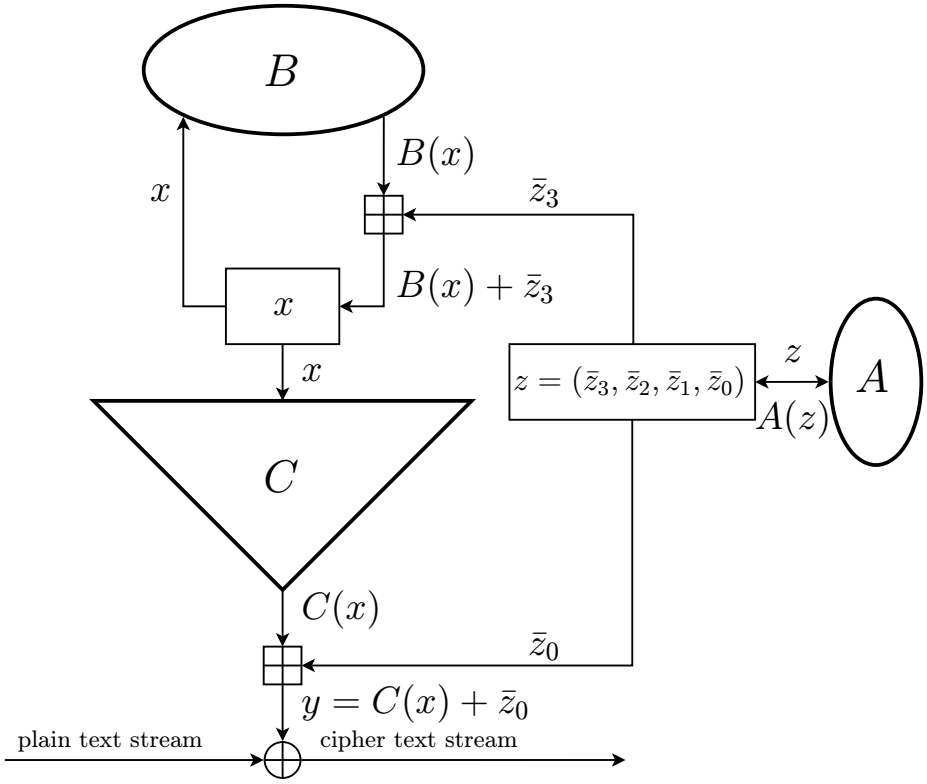

Figure 1: ABC keystream generator. $\bigoplus$ denotes bitwise addition modulo 2 (XOR). $+$ and $\boxplus$ represent arithmetical addition modulo 2^{32}.

3.3 Key Expansion and Nonce Setup

Here the ABC key expansion and IV setup routines are defined. The ABC initialization procedure supports 128-bit keys and 128-bit initial vectors.

The main idea behind the initialization routine is to use the ABC keystream generator (defined in Subsection 3.2 as algorithm "ABC keystream generator") with feedback. A single call of ABC keystream generator will be denoted here as the function g:

$$\begin{array}{l} g(z, x, d_0, d_1, d_2, e, \{e_i\}_{i=0}^{31}), \\ g : \mathbb{Z}/2^{1312}\mathbb{Z} \to \mathbb{Z}/2^{32}\mathbb{Z}. \end{array} \tag{6}$$

If the optimization tables $\{T_i\}_{i=0}^{t-1}$ have been precomputed out of coefficients $e, \{e_i\}_{i=0}^{31}$ as described in Subsection 6.1, the ABC keystream generator can be alternatively called as

$$g(z, x, d_0, d_1, d_2, \{T_i\}_{i=0}^{t-1}).$$

In both calls variables z and x are changed in a call[3] (i.e. within the function g) as implied by the algorithm "ABC keystream generator" in Subsection 3.2. The ABC initialization procedure consists of 3 stages:

Key expansion. Let k be the 128-bit primary key:

$$k \in \mathbb{Z}/2^{128}\mathbb{Z},\ k = (\bar{k}_3, ..., \bar{k}_0) \in (\mathbb{Z}/2^{32}\mathbb{Z})^4. \tag{7}$$

During the key expansion the following temporary variables

$$z' = (\bar{z}'_3, \bar{z}'_2, \bar{z}'_1, \bar{z}'_0) \in \mathbb{Z}/2^{128}\mathbb{Z},\ \bar{z}'_i \in \mathbb{Z}/2^{32}\mathbb{Z},\ i = 0, 1, 2, 3;$$

$$x' \in \mathbb{Z}/2^{32}\mathbb{Z}, d'_0 \in \mathbb{Z}/2^{32}\mathbb{Z}, d'_1\mathbb{Z}/2^{32}\mathbb{Z}, d'_2\mathbb{Z}/2^{32}\mathbb{Z}$$

are set directly using the primary key k. Then after a number of warmups with feedback the ABC keystream generator is run to fill the ABC secret state. At each warmup iteration a 32-bit output of generator is fed back to some part of the state by means of bitwise addition modulo 2. After the warmup phase the ABC keystream generator is used to obtain the values for $d_0, d_1, d_2, x, z, \{e_i\}_{i=0}^{31}, e$, composing the secret state of ABC.

The values for $\{e_i\}_{i=0}^{31}$ are obtained in a special way to avoid the combinations of values that are weak, see Subsections 5.3 and 5.5 for the explanation.

In all ABC keystream generator calls during the key expansion stage the fixed values[4] $\mathfrak{e}, \{\mathfrak{e}_i\}_{i=0}^{31}$ of the coefficients of C are used (see Table 1). Since the values are fixed, it is recommended to precompute the corresponding optimization tables (once for all possible keys and initial values) following the idea from Subsection 6.1. The fixed precomputed optimization tables for the key setup routine are denoted below as $\{\mathfrak{T}_i\}_{i=0}^{t-1}$.

[3] Strictly speaking one should define g as

$$\begin{array}{l} g:\ z \times x \times d_0 \times d_1 \times d_2 \times e \times \{e_i\}_{i=0}^{31} \mapsto\ y \times x \times z, \\ g:\ \mathbb{Z}/2^{1312}\mathbb{Z}\ \to\ \mathbb{Z}/2^{32}\mathbb{Z} \times \mathbb{Z}/2^{32}\mathbb{Z} \times \mathbb{Z}/2^{128}\mathbb{Z}. \end{array}$$

In the following the notation $g:\ \mathbb{Z}/2^{1312}\mathbb{Z}\ \to\ \mathbb{Z}/2^{32}\mathbb{Z}$ will be used for simplification considerations. One should always keep in mind that the variables x and z are changed in the corresponding way within each call of g.

[4] Note that the restrictions imposed by (5) are fulfilled.

Table 1: Coefficients for key setup routine, in hexadecimal notation

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

Optimization table precomputation. At this stage the optimization tables $\{T_i\}_{i=0}^{t-1}$ are computed out of the coefficients $e, \{e_i\}_{i=0}^{31}$ using formulae (10) and (11).

IV setup. Let iv be a 128-bit initial value:

$$iv \in \mathbb{Z}/2^{128}\mathbb{Z},\ iv = (\bar{iv}_3, ..., \bar{iv}_0) \in (\mathbb{Z}/2^{32}\mathbb{Z})^4. \tag{8}$$

During this stage the idea of self-initialization with feedback is made use of too. The value iv is added to z, x, d_0, d_1, d_2 bitwise modulo 2 and then the keystream generator is warmed up with feedback in a way similar to that at the key expansion stage, but different (key-dependent) coefficients are used.

Now we are going to define the ABC setup routine formally. In the following it is assumed that the table optimization is employed and $\{\mathfrak{T}_i\}_{i=0}^{t-1}$ or $\{T_i\}_{i=0}^{t-1}$ are used in calls to the ABC keystream generator g.

ABC SETUP ROUTINE

KEY EXPANSION

INPUT: $k = (\bar{k}_3, \ldots, \bar{k}_0), z = (\bar{z}_3, \bar{z}_2, \bar{z}_1, \bar{z}_0), x, d_0, d_1, d_2, e, \{e_i\}_{i=0}^{31}, \{\mathfrak{T}_i\}_{i=0}^{t-1}$

TEMPORARY VARIABLES: $z' = (\bar{z}'_3, \bar{z}'_2, \bar{z}'_1, \bar{z}'_0), x', d'_0, d'_1, d'_2, i, j, \zeta$

Initialization:

$d'_0 \leftarrow \bar{k}_3$ AND $\underbrace{1\ldots1}_{30}00_2$;

$d'_1 \leftarrow (\bar{k}_2$ AND $\underbrace{1\ldots1}_{30}00_2)$ OR 1;

$d'_2 \leftarrow \bar{k}_1$ AND $\underbrace{1\ldots1}_{30}00_2$;

$x' \leftarrow \bar{k}_0$;

$\bar{z}'_0 \leftarrow (\bar{k}_0 \ggg 16)$ OR 2;

$\bar{z}'_1 \leftarrow \bar{k}_1 \ggg 16$;

$\bar{z}'_2 \leftarrow \bar{k}_2 \ggg 16$;

$\bar{z}'_3 \leftarrow \bar{k}_3 \ggg 16;$

Initial state warm-up with feedback:

$\zeta \leftarrow g(z', x', d'_0, d'_1, d'_2, \{\mathfrak{T}_i\}_{i=0}^{t-1});$

$x' \leftarrow x' \,\mathsf{XOR}\, \zeta;$

$\zeta \leftarrow g(z', x', d'_0, d'_1, d'_2, \{\mathfrak{T}_i\}_{i=0}^{t-1});$

$d'_0 \leftarrow (d'_0 \,\mathsf{XOR}\, \zeta) \,\mathsf{AND}\, \underbrace{1\ldots1}_{30}00_2;$

$\zeta \leftarrow g(z', x', d'_0, d'_1, d'_2, \{\mathfrak{T}_i\}_{i=0}^{t-1});$

$d'_1 \leftarrow ((d'_1 \,\mathsf{XOR}\, \zeta) \,\mathsf{AND}\, \underbrace{1\ldots1}_{30}00_2) \,\mathsf{OR}\, 1;$

$\zeta \leftarrow g(z', x', d'_0, d'_1, d'_2, \{\mathfrak{T}_i\}_{i=0}^{t-1});$

$d'_2 \leftarrow (d'_2 \,\mathsf{XOR}\, \zeta) \,\mathsf{AND}\, \underbrace{1\ldots1}_{30}00_2;$

$\zeta \leftarrow g(z', x', d'_0, d'_1, d'_2, \{\mathfrak{T}_i\}_{i=0}^{t-1});$

$\bar{z}'_2 \leftarrow \bar{z}'_2 \,\mathsf{XOR}\, \zeta;$

$\zeta \leftarrow g(z', x', d'_0, d'_1, d'_2, \{\mathfrak{T}_i\}_{i=0}^{t-1});$

$\bar{z}'_2 \leftarrow \bar{z}'_2 \,\mathsf{XOR}\, \zeta;$

$\zeta \leftarrow g(z', x', d'_0, d'_1, d'_2, \{\mathfrak{T}_i\}_{i=0}^{t-1});$

$\bar{z}'_2 \leftarrow \bar{z}'_2 \,\mathsf{XOR}\, \zeta;$

$\zeta \leftarrow g(z', x', d'_0, d'_1, d'_2, \{\mathfrak{T}_i\}_{i=0}^{t-1});$

$\bar{z}'_2 \leftarrow \bar{z}'_2 \,\mathsf{XOR}\, \zeta;$

$\bar{z}'_0 \leftarrow \bar{z}'_0 \,\mathsf{OR}\, 2;$

Main state filling:

$d_0 \leftarrow g(z', x', d'_0, d'_1, d'_2, \{\mathfrak{T}_i\}_{i=0}^{t-1}) \,\mathsf{AND}\, \underbrace{1\ldots1}_{30}00_2;$

$d_1 \leftarrow (g(z', x', d'_0, d'_1, d'_2, \{\mathfrak{T}_i\}_{i=0}^{t-1}) \,\mathsf{AND}\, \underbrace{1\ldots1}_{30}00_2) \,\mathsf{OR}\, 1;$

$d_2 \leftarrow g(z', x', d'_0, d'_1, d'_2, \{\mathfrak{T}_i\}_{i=0}^{t-1}) \,\mathsf{AND}\, \underbrace{1\ldots1}_{30}00_2;$

$x \leftarrow g(z', x', d'_0, d'_1, d'_2, \{\mathfrak{T}_i\}_{i=0}^{t-1});$

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

$\bar{z}_1 \leftarrow g(z', x', d'_0, d'_1, d'_2, \{\mathfrak{T}_i\}_{i=0}^{t-1});$

$\bar{z}_2 \leftarrow g(z', x', d'_0, d'_1, d'_2, \{\mathfrak{T}_i\}_{i=0}^{t-1});$

$\bar{z}_3 \leftarrow g(z', x', d'_0, d'_1, d'_2, \{\mathfrak{T}_i\}_{i=0}^{t-1});$

$e \leftarrow g(z', x', d'_0, d'_1, d'_2, \{\mathfrak{T}_i\}_{i=0}^{t-1});$

for i from 0 to 31 do

$e_i \leftarrow g(z', x', d'_0, d'_1, d'_2, \{\mathfrak{T}_i\}_{i=0}^{t-1});$

$e_i \leftarrow e_i \,\mathsf{AND}\, g(z', x', d'_0, d'_1, d'_2, \{\mathfrak{T}_i\}_{i=0}^{t-1});$

$e_i \leftarrow e_i \,\mathsf{AND}\, g(z', x', d'_0, d'_1, d'_2, \{\mathfrak{T}_i\}_{i=0}^{t-1});$

$e_i \leftarrow e_i \,\mathsf{AND}\, g(z', x', d'_0, d'_1, d'_2, \{\mathfrak{T}_i\}_{i=0}^{t-1});$

end for

for i from 0 to 31 do

for j from 0 to 2 do

$\zeta \leftarrow g(z', x', d'_0, d'_1, d'_2, \{\mathfrak{T}_i\}_{i=0}^{t-1})$;
$\delta_i(e_{[\zeta]_0^4}) \leftarrow 1$;
$\delta_i(e_{[\zeta]_5^9}) \leftarrow 1$;
$\delta_i(e_{[\zeta]_{10}^{14}}) \leftarrow 1$;
$\delta_i(e_{[\zeta]_{15}^{19}}) \leftarrow 1$;
$\delta_i(e_{[\zeta]_{20}^{24}}) \leftarrow 1$;
$\delta_i(e_{[\zeta]_{25}^{29}}) \leftarrow 1$;
end for
$\zeta \leftarrow g(z', x', d'_0, d'_1, d'_2, \{\mathfrak{T}_i\}_{i=0}^{t-1})$;
$\delta_i(e_{[\zeta]_0^4}) \leftarrow 1$;
$\delta_i(e_{[\zeta]_5^9}) \leftarrow 1$;
end for
$e_{30} \leftarrow e_{30}$ OR$(e_{31}$ AND$\underbrace{0\ldots0}_{16}\underbrace{1\ldots1}_{16}{}_2)$;
$e_{31} \leftarrow (e_{31}$ AND$\underbrace{1\ldots1}_{16}\underbrace{0\ldots0}_{16}{}_2)$ OR$\underbrace{0\ldots0}_{15}1\underbrace{0\ldots0}_{16}{}_2$;

OUTPUT: $z, x, d_0, d_1, d_2, e, \{e_i\}_{i=0}^{31}$

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:

$d_0 \leftarrow (d_0$ XOR $\bar{iv}_3)$ AND $\underbrace{1\ldots1}_{30}00_2$;
$d_1 \leftarrow ((d_1$ XOR $\bar{iv}_2)$ AND $\underbrace{1\ldots1}_{30}00_2)$ OR 1;
$d_2 \leftarrow (d_2$ XOR $\bar{iv}_1)$ AND $\underbrace{1\ldots1}_{30}00_2$;
$x \leftarrow x$ XOR $\bar{iv}_0$;

$\bar{z}_0 \leftarrow (\bar{z}_0 \,\mathsf{XOR}(\bar{iv}_0 \ggg 16))\,\mathsf{OR}\,2;$
$\bar{z}_1 \leftarrow \bar{z}_1 \,\mathsf{XOR}(\bar{iv}_1 \ggg 16);$
$\bar{z}_2 \leftarrow \bar{z}_2 \,\mathsf{XOR}(\bar{iv}_2 \ggg 16);$
$\bar{z}_3 \leftarrow \bar{z}_3 \,\mathsf{XOR}(\bar{iv}_3 \ggg 16);$

Warm-up with feedback:

$\zeta \leftarrow g(z, x, d_0, d_1, d_2, \{T_i\}_{i=0}^{t-1});$
$x \leftarrow x \,\mathsf{XOR}\,\zeta;$
$\zeta \leftarrow g(z, x, d_0, d_1, d_2, \{T_i\}_{i=0}^{t-1});$
$d_0 \leftarrow (d_0 \,\mathsf{XOR}\,\zeta)\,\mathsf{AND}\,\underbrace{1\ldots1}_{30}00_2;$
$\zeta \leftarrow g(z, x, d_0, d_1, d_2, \{T_i\}_{i=0}^{t-1});$
$d_1 \leftarrow ((d_1 \,\mathsf{XOR}\,\zeta)\,\mathsf{AND}\,\underbrace{1\ldots1}_{30}00_2)\,\mathsf{OR}\,1;$
$\zeta \leftarrow g(z, x, d_0, d_1, d_2, \{T_i\}_{i=0}^{t-1});$
$d_2 \leftarrow (d_2 \,\mathsf{XOR}\,\zeta)\,\mathsf{AND}\,\underbrace{1\ldots1}_{30}00_2;$
$\zeta \leftarrow g(z, x, d_0, d_1, d_2, \{T_i\}_{i=0}^{t-1});$
$\bar{z}_2 \leftarrow \bar{z}_2 \,\mathsf{XOR}\,\zeta;$
$\zeta \leftarrow g(z, x, d_0, d_1, d_2, \{T_i\}_{i=0}^{t-1});$
$\bar{z}_2 \leftarrow \bar{z}_2 \,\mathsf{XOR}\,\zeta;$
$\zeta \leftarrow g(z, x, d_0, d_1, d_2, \{T_i\}_{i=0}^{t-1});$
$\bar{z}_2 \leftarrow \bar{z}_2 \,\mathsf{XOR}\,\zeta;$
$\zeta \leftarrow g(z, x, d_0, d_1, d_2, \{T_i\}_{i=0}^{t-1});$
$\bar{z}_2 \leftarrow \bar{z}_2 \,\mathsf{XOR}\,\zeta;$
$\bar{z}_0 \leftarrow \bar{z}_0 \,\mathsf{OR}\,2;$

OUTPUT: $z = (\bar{z}_3, \bar{z}_2, \bar{z}_1, \bar{z}_0), x, d_0, d_1, d_2$

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 z, x, d_0, d_1, d_2 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.

4 Tweaks

This section contains an outline of tweaks that were introduced in the current version of ABC during Phase 1 of eSTREAM.

4.1 128-bit LFSR A

ABC v.1 [10] was the original submission to eSTREAM. It included a 64-bit long LFSR. This enabled C. Berbain and H. Gilbert in [13] and independently S. Khazaei in [18] to mount a divide-and-conquer attack. The attack exploited the fact that the filter function C is non-balanced and its output is efficiently distinguishable. Therefore, the state of the LFSR A could be determined by the full search over the possible LFSR states.

This attack becomes impractical when the LFSR A is 128 bits long. The 128-bit LFSR used in the current version of ABC (see Subsection 3.1) was introduced in [8].

4.2 Modified Single-Cycle Function B

To compensate for the overhead caused by the 128-bit LFSR A, the single-cycle transform of B was exchanged with a similar but faster single-cycle transform. The new transform is used in the current version of ABC (see Subsection 3.1) and was also introduced in in [8], which lead to ABC v.2 [11].

4.3 Modified Key Setup Procedure

H. Wu and B. Preneel in [26] discovered the weak keys of ABC v.2 for which an efficient correlation attack can be mounted. The key is weak if it causes the least significant bit of every coefficient $e_0, e_1, \ldots, e_{31}$ of the function C to be 0 after the key setup.

This kind of weak keys and also other potential kinds of weak keys can be eliminated at the key setup procedure. The modified key setup routine is introduced in the current version of ABC – see Subsections 3.3 and 5.5.

4.4 Restricted Length of a Single Stream

The allowed length of a single stream for the current ABC version is 2^{64} 32-bit words. That is, for a given pair of a key and an initial value the output of ABC is restricted to 2^{64} 32-bit words

5 Design Rationale and Brief Security Analysis

The ABC stream cipher is an example of special design techniques whose mathematical background has being developed since early 90-th (see [2]–[7]), and which exploit some ideas of the p-adic theory of dynamical systems .

The underlying mathematical theory concerns certain mappings that have been studied in mathematics since early 70-th under different names (triangle mappings, compatible functions, determined functions); these mappings turned out to be Lipschitzian p-adic functions, see [5], [2], [3], and [4]. To cryptographic community special mappings of this kind were introduced only in 2002 under the name of T-functions (see [20]). The ABC cipher utilizes some single-cycle T-functions.

The keystream 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 [23]. 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. [23], where the diversity is guaranteed only.

Mathematical base of our techniques are skew products, also known as skew shifts (in ergodic theory) , or wreath products (in algebra and automata theory),

see [6] and [7]. It is worth noticing here that T-functions are just skew shifts of a special kind.

On the other hand, the ABC stream cipher can be seen as a generalization of the hard knapsack stream cipher [22] .

5.1 LFSR A

Linear feedback shift register A is of length 128 and its characteristic polynomial is $\phi(\theta) = \psi(\theta)\theta$, where $\psi(\theta) = \theta^{127} + \theta^{63} + 1$ is primitive. Since bit operations are relatively slow on general purpose processors, a word oriented representation (1) of the LFSR A is used, as in [15]. Two outputs from this LFSR are obtained, the first one for the state transition procedure and the second one for updating the output function.

It is important to stress here that (1) is just another (word-oriented) representation of the 127-bit LFSR with primitive polynomial $\psi(\theta)$. Thus, the cycle length of this LFSR is $2^{127} - 1$, and not $2^{128} - 1$.

This also leads to the fact that the cycle length becomes 1 in case the initial state $z = (\bar{z}_3, \bar{z}_2, \bar{z}_1, \bar{z}_0)$ of A is either $(0, 0, 0, 0)$ or $(0, 0, 0, 1)$. This danger is eliminated by forcing $\delta_1(z)$ to 1 in the ABC key setup and IV setup procedures, thus reducing the secret state of A primitive to 126 bits.

5.2 Single-Cycle Function B

The restrictions defined in (3) guarantee that B is a single-cycle mapping modulo 2^{32} (that follows from results in [6] and [21]). The form of $B(x)$ is determined at the initialization stage through setting the 90 bits of d_0, d_1 and d_2 in a key- and IV-dependent manner.

5.3 Filter Function C

The function C is a non-linear mapping and is the main security block of ABC. In fact, C is based on the knapsack function proved to be highly non-linear and to have a high algebraic degree with respect to almost all output bit coordinates. The imposed restriction on the coefficient e_{31} together with (4) guarantee the longest possible period and high linear complexity of the keystream, see Subsection 5.4.

The coefficients $e, e_0, e_1, \ldots, e_{31}$ of C are key-dependent. This makes the cumulative internal state of the cipher sufficiently large. On the other hand, the arbitrary filling of the coefficients of C can lead to the weak cases enabling attacks like [26]. The filling is weak if one or more bit columns $(\delta_k(e_0), \delta_k(e_1), \ldots, \delta_k(e_{31}))^T$ possess low Hamming weight (especially, if it is equal to zero). The weak fillings are avoided by the special steps in the key setup procedure, see Subsections 3.3 and 5.5.

Rotation of the output of the function C moves its least significant bit to the position where the carries occur in the addition with $\bar{z}_0$. Rotation by 16 bits was chosen to enable fast implementation by byte swapping on 8-bit and 16-bit processors.

5.4 Keystream Generator

The sequence of states $(x; z)$ of the ABC keystream generator forms a cycle of length $2^{32} \cdot (2^{127} - 1)$. The cycle is totally determined by the LFSR A and function B: Pairwise distinct functions B (that correspond to distinct triples of coefficients d_0, d_1, d_2) 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.

Proven Keystream Properties

The following properties of the keystream produced by ABC keystream generator are proved:

- The length P of the shortest period of the keystream sequence of 32-bit words is $2^{32} \cdot (2^{127} - 1)$
- The distribution of the keystream 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 keystream satisfies the following inequality:

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

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

Proofs follow from the results presented in [2]–[7].

Note that it could be proved that linear complexity λ of the keystream satisfies the inequality $2^{31} \cdot (2^{127} - 1) + 1 \geq \lambda \geq 2^{31} + 1$. 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 $\left| \frac{\mu(a)}{P} - \frac{1}{2^{32}} \right| < \frac{1}{\sqrt{P}}$.

5.5 Key Expansion and Nonce Setup

The design of the key setup routine prevents the weak fillings of C coefficients described in Subsection 3.3. After the key setup, the number of ones n_1 in every column $(\delta_k(e_0), \delta_k(e_1), \ldots, \delta_k(e_{31}))$, $k = 0, 1, \ldots, 31$, is distributed in a non-binomial way with the mean $E(n_1)$ close to 16. The probability of $n_1 = 0$ is 0 for $k = 0, 1, \ldots, 31$.

Both key setup and nonce setup use feedback for the proper modification of the internal state of the cipher. Nonce setup involves 8 iterations of the keystream generator and is considerably fast.

5.6 Brief Security Analysis

To analyse the security of the ABC stream cipher it is important to stress that we impose only minor restrictions on coefficients of the functions B and C. We assume that the rest of the bits of these coefficients are produced in a key-dependent way out of the key. 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.

Time-memory-data tradeoffs, either suggested by Biryukov and Shamir in [14] or the generalized ones suggested by Hong and Sarkar in [17] (see [16] for discussion), are not posing a threat to the ABC cipher. For the former case the TMD complexity is greater than the 2^{128} exhaustive search due to large state size (above 1300 bits). The latter case also does not lower the exhaustive search threshold, even if unlimited computational resources at precomputation stage are considered, as ABC uses a 128-bit initial value and a 128-bit key, the collective enthropy of key and IV being 256.

Related-key and resynchronization attacks are withstood by the proper key setup and IV setup algorithms. Both algorithms (see Subsection 3.3) use self-initialization with feedback, which does not show possibilities for applying these techniques.

Algebraic attacks are thwarted by the non-linear properties of the output primitive (see Subsection 5.3).

Correlation attacks. In the current version of ABC, we did not find any parity checks that are sufficiently biased to mount a correlation attack having a maximum of 2^{64} ABC outputs available in a single stream.

Empirical statistical testing, performed with NIST suite with respect to AES candidates evaluation (see [24]), has not indicated any deviation of ABC keystream from a random sequence. Moreover, our testing has shown that the keystream statistical properties provided by ABC are at least as good as those of AES finalists, given in [25]).

6 Implementation

This section contains the description of the efficient computation of the function C, followed by the outline of processor-specific implementation issues.

6.1 Efficient computation of C

Here we present a way to speed up the computation of the function C through the introduction of several tables of variable length using a version of the idea outlined in [12].

To compute S as in (5) it is not necessary to read x bitwise at each iteration. Instead of this a window technique can be used. 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 :

$$x = (\hat{x}_{t-1}, \ldots, \hat{x}_0),\ \hat{x}_i \in \mathbb{Z}/2^w\mathbb{Z},\ i = 0, \ldots, t-1. \tag{9}$$

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

$$T_i[j] = \sum_{l=0}^{w-1} \delta_l(j) \cdot e_{w \cdot i + l} \bmod 2^{32},\quad j = 0, \ldots, 2^w - 1, \tag{10}$$

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

$$T_0[j] = e + \sum_{l=0}^{w-1} \delta_l(j) \cdot e_l \bmod 2^{32}, \quad j = 0, \ldots, 2^w - 1, \tag{11}$$

for $i = 0$. Then S can be rewritten in the corresponding way:

$$S(x) = \sum_{s=0}^{t-1} T_s[\hat{x}_s], \quad \hat{x}_s \in \mathbb{Z}/2^w\mathbb{Z}, s = 0, \ldots, t-1. \tag{12}$$

Using this window optimization method it is possible to vary memory consumption. We have computed the respective values and give them in Table 2. 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. It requires 33 32-bit values $e, \{e_i\}_{i=0}^{31}$ and therefore 132 byte memory, which is the minimum value for the ABC filter function C.

Table 2: 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

6.2 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 icc 9.0. The measurements were carried out within the eSTREAM testing framework [1] under Linux with a 2.6.11 kernel on a laptop with a 1.73 GHz Intel Pentium M 740 processor with 64KB L1 cache and 2MB L2 cache, and 512 MB main memory. The best throughput for this hardware configuration was achieved with the combination of three windows of bit lengths 12, 12 and 8 bits. The results of the throughput measurement for C reference implementation including the memory needed for the ABC internal state can be found in Table 3 for different measurement conditions. The results for key an IV setup procedures, and also for the table precomputation, can be found in Table 4.

The 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 for different keys, which would raise the total performance of ABC in this case. Exact values depend, however, on a specific platform.

Table 3: ABC v.3 throughput for Intel Pentium M

w	Cycles per byte	Memory, bytes
2	12.13	320
4	7.63	576
8	4.81	4160
12/12/8	4.03	33856

Table 4: Cost of ABC v.3 setup routines in processor cycles for Intel Pentium M

w	Key setup	IV setup	Table precomputation	Key setup without precomputation
2	14188	407	60	14128
4	11581	269	1861	9720
8	52741	214	45891	6850
12/12/8	586220	209	579999	6221

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 keystream 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.

7 Conclusion

In this paper we presented ABC – a fast flexible synchronous stream cipher for software applications. We introduced tweaks increasing the security of the cipher and eliminating the attacks mounted on the previous versions of ABC.

ABC advantages are high performance, some provable security properties and high flexibility. ABC meets a number of industrial software implementation properties such as generic performance property, flexible storage consumption and flexible cost of IV/key setup procedures. This makes ABC applicable not only on standard 32-bit platforms, but in some embedded security systems with high performance requirements as well.

References

[1] eSTREAM optimized code HOWTO. http://www.ecrypt.eu.org/stream/perf. 15

[2] V. S. Anashin. Uniformly distributed sequences over p-adic integers (in Russian). *Mat. Zametki*, 55(2):3–46, 1994. English transl. in *Mathematical Notes*, 55(2):109–133, 1994. 11, 13

[3] V. S. Anashin. Uniformly distributed sequences in computer algebra, or how to construct program generators of random numbers. *J. Math. Sci.*, 89(4):1355–1390, 1998. 11

[4] V. S. Anashin. Uniformly distributed sequences of p-adic integers, II (in Russian). *Diskret. Mat.*, 14(4):3–64, 2002. English transl. in *Discrete Math. Appl.*, 12(2):527–590, 2002. A preprint in English available from http://arXiv.org/math.NT/0209407. 11

[5] Vladimir Anashin. Uniformly distributed sequences over p-adic integers. In I. Shparlinsky A. J. van der Poorten and H. G. Zimmer, editors, *Number theoretic and algebraic methods in computer science. Proceedings of the Int'l Conference (Moscow, June–July, 1993)*, pages 1–18. World Scientific, 1995. 11

[6] Vladimir Anashin. Pseudorandom number generation by p-adic ergodic transformations, 2004. Available from http://arXiv.org/abs/cs.CR/0401030. 12

[7] Vladimir Anashin. Pseudorandom number generation by p-adic ergodic transformations: An addendum, 2004. Available from http://arXiv.org/abs/cs.CR/0402060. 11, 12, 13

[8] Vladimir Anashin, Andrey Bogdanov, and Ilya Kizhvatov. Increasing the ABC stream cipher period. eSTREAM, ECRYPT Stream Cipher Project, Report 2005/050, 2005. http://www.ecrypt.eu.org/stream. 11

[9] Vladimir Anashin, Andrey Bogdanov, and Ilya Kizhvatov. Security and implementation properties of ABC v.2. SASC 2006, 2006. http://www.ecrypt.eu.org/stream, Report 2006/026. 1

[10] Vladimir Anashin, Andrey Bogdanov, Ilya Kizhvatov, and Sandeep Kumar. ABC: A new fast flexible stream cipher. eSTREAM, ECRYPT Stream Cipher Project, Report 2005/001, 2005. http://www.ecrypt.eu.org/stream. 1, 10

[11] Vladimir Anashin, Andrey Bogdanov, Ilya Kizhvatov, and Sandeep Kumar. ABC: A new fast flexible stream cipher. Version 2, 2005. http://crypto.rsuh.ru/papers/abc-spec-v2.pdf. 1, 11

[12] V. L. Arlazarov, E. A. Dinic, M. A. Kronrod, and I. A. Faradzev. On economic construction of the transitive closure of a directed graph (in Russian). *Dokl. Akad. Nauk SSSR*, 194, 1970. English translation in *Soviet Math. Dokl.*, 11, 1975, pp. 1209-1210. 14

[13] Côme Berbain and Henry Gilbert. Cryptanalysis of ABC. eSTREAM, ECRYPT Stream Cipher Project, Report 2005/048, 2005. http://www.ecrypt.eu.org/stream. 1, 10

[14] Alex Biryukov and Adi Shamir. Cryptanalytic time/memory/data tradeoffs for stream ciphers. In *Advances in Cryptology – ASIACRYPT'00*, volume 1976, pages 1–13, 2000. 14

[15] C. Carroll, A. Chan, and M. Zhang. The software-oriented stream cipher SSC2. In *Fast Software Encryption – FSE 2000*, volume 1978 of *LNCS*, 2001. 12

[16] Joseph Lano Christophe De Cannière and Bart Preneel. Comments on the rediscovery of time memory data tradeoffs. eSTREAM, ECRYPT Stream Cipher Project, Report 2005/040, 2005. http://www.ecrypt.eu.org/stream. 14

[17] Jin Hong and Palash Sarkar. Rediscovery of time memory tradeoffs. Cryptology ePrint Archive, Report 2005/090, 2005. http://eprint.iacr.org/. 14

[18] Shahram Khazaei. Divide and conquer attack on ABC stream cipher. eSTREAM, ECRYPT Stream Cipher Project, Report 2005/052, 2005. http://www.ecrypt.eu.org/stream. 1, 10

[19] Shahram Khazaei and Mohammad Kiaei. Distinguishing attack on the ABC v.1 and v.2. eSTREAM, ECRYPT Stream Cipher Project, Report 2005/061, 2005. http://www.ecrypt.eu.org/stream. 1

[20] Alexander Klimov and Adi Shamir. A new class of invertible mappings. In B.S.Kaliski Jr.et al., editor, *Cryptographic Hardware and Embedded Systems 2002*, volume 2523 of *LNCS*, pages 470–483. Springer-Verlag, 2003. 11

[21] L. Kotomina. Fast nonlinear congruential generators (in Russian). Diploma Thesis. Russian State University for the Humanities, Moscow, 1999. 12

[22] R.A. Rueppel. *Analysis and Design of Stream Ciphers*. Springer Verlag, 1986. 12

[23] A. Shamir and B. Tsaban. Guaranteeing the diversity of number generators. *Information and Computation*, 171:350–363, 2001. Available from http://arXiv.org/abs/cs.CR/0112014. 11

[24] J. Soto. Randomness testing of the advanced encryption standard candidate algorithms. NIST IR 6390. http://csrc.nist.gov/rng/AES-REPORT2.doc. 14

[25] J. Soto and L. Bassham. Randomness testing of the advanced encryption standard finalist candidates. NIST IR 6483. http://csrc.nist.gov/rng/aes-report-final.doc. 14

[26] Hongjun Wu and Bart Preneel. Cryptanalysis of ABC v2. eSTREAM, ECRYPT Stream Cipher Project, Report 2006/029, 2006. http://www.ecrypt.eu.org/stream. 1, 11, 12