THE BLOCK CIPHER NSABC (PUBLIC DOMAIN)

ALICE NGUYENOVA-STEPANIKOVA (*) AND TRAN NGOC DUONG (**)

Abstract. We introduce NSABC/w – Nice-Structured Algebraic Block Cipher using w-bit word arithmetic, a $4w$-bit analogous of Skipjack [NSA98] with $5w$-bit key. The Skipjack's internal 4-round Feistel structure is replaced with a w-bit, 2-round cascade of a binary operation $(x, z) \mapsto (x \boxdot z) \lll (w/2)$ that permutes a text word x under control of a key word z. The operation $\boxdot$, similarly to the multiplication in IDEA [LM91, LMM91], bases on an algebraic group over w-bit words, so it is also capable of decrypting by means of the inverse element of z in the group. The cipher utilizes a secret $4w$-bit tweak – an easily changeable parameter with unique value for each block encrypted under the same key [LRW02] – that is derived from the block index and an additional $4w$-bit key. A software implementation for $w = 64$ takes circa 9 clock cycles per byte on x86-64 processors.

1. Introduction

In the today's world full of crypto algorithms, one may wonder what makes a block cipher attractive.

In the authors' opinion, the answer to the question is one word: elegance. If something looks nice, then there is a big chance that it is also good.

An elegant specification makes it easier to memorize. Memorability makes it easier to realize and to analyze, that allows for fruitful cryptanalytic results, leading to deeper understanding which, in turn, makes greater confidence in the algorithm.

The elegance comprises the following features:

- Few algebraic operations. Using of many operations results in hardly-tractable and possibly undesirable interactions between them.
- Simple and regular key schedule. A complex key schedule, which effectively adds another, unrelated, function to the cipher, results in hardly-tractable and possibly undesirable interactions between the functions.

IDEA, a secure block cipher designed by Xuejia Lai and James L. Massey [LM91, LMM91] is an example of elegance. Besides being elegant with an efficient choice and arrangement of algebraic operations, it is elegant for some more features:

- The use of incompatible group operations, where *incompatible* means there are no simple relations (such as distributivity) between them. The incompatibility eliminates any exploitable algebraic property thus makes it infeasible to solve the cipher algebraically.
- The use of modular multiplication. Multiplication produces huge mathematical complexity while consuming few clock cycles on modern processors. It thus greatly contributes to security and efficiency of the cipher.

Date: May 8th, 2011. This is version 2 of the algorithm, superseding version 1 that was published on Usenet July 2010.

Key words and phrases. block cipher, tweakable, algebraic, multiplication, IDEA, Skipjack.

(*) Hradcany, Praha, Czech Republic.

(**) Pernink, Karlovy Vary, Czech Republic. E-mail: tranngocduong@gmail.com.

However, IDEA uses multiplication modulo the Fermat prime $2^w + 1$ which does not exist for $w = 32$ or $w = 64$, making it not extendable to machine word lengths nowadays. Furthermore, its key schedule is rather irregular due to the rotation of the primary key.

Skipjack, a secure block cipher designed by the U.S. National Security Agency [NSA98], is another example of elegant design. Besides being elegant with an efficient, simple and regular key schedule, it is elegant for one more feature: the use of two ciphers — an outer cipher, or *wrapper*, consisting of first and last rounds, and an inner cipher, or *core*, consisting of middle rounds.

The terms "core" and "wrapper" were introduced in the design rationale of a structural analogous of Skipjack: the block cipher MARS [IBM98]. MARS's designers justify this two-layer structure by writing that it breaks any repetitious property, it makes any iterative characteristic impossible, and it disallows any propagation of eventual vulnerabilities in either layer to the other one, thus making attacks more difficult. The wrapper is primarily aimed at fast diffusion and the core primarily at strong confusion. As Claude E. Shannon termed in his pioneer work [Sha49], *diffusion* here refers to the process of letting each input bit affect many output bits (or, equivalently, each output bit be affected by many input bits), and *confusion* here refers to the process of letting that affection very involved, possibly by doing it multiple times in very different ways. If a cipher is seen as a polynomial map in the plaintext and the key to the ciphertext, then the methods of diffusion and confusion can be described as the effort of making the polynomials as complete as possible, i.e. such that they contain virtually all terms at all degrees. This *algebraic* approach is very evident in the structure of Skipjack (see Figure 5.1). Skipjack (as opposed to MARS) was moreover sought elegant as the wrapper there is, in essence, the inverse function of the core.

However, Skipjack uses an S-box that renders it rather slow, hard to program in a secure and efficient manner, and not extendable to large machine word lengths, as such.

This article describes an attempt to combine the elegant idea of using incompatible and complex machine-oriented algebraic operations in IDEA with the elegant structure of Skipjack into a scalable and tweakable block cipher called NSABC — Nice-Structured Algebraic Block Cipher.

NSABC is scalable. It is defined for every even word length w. It encrypts a $4w$-bit text block under a $5w$-bit key, thus allows scaling up with 8-bit increment in block length and 10-bit increment in key length.

NSABC is tweakable. It can use an easily changeable $4w$-bit parameter, called *tweak* [LRW02], to make a unique version of the cipher for every block encrypted under the same key. Included in the specification is a formula for changing the tweak.

NSABC makes use of entirely the overall structure of Skipjack, including the key schedule, and only replaces the internal 4-round Feistel structure of Skipjack with another structure. The new structure consists of two rounds of the binary operation $(x, z) \mapsto (x \underset{e}{\boxdot} z) \lll (w/2)$, that encrypts a text word x using a key word z and a key-dependent word e. The operation $\underset{e}{\boxdot}$ is derived from an algebraic group over w-bit words taking e as the unit element, so it is also capable of decrypting by means of the inverse element of z in the group. The two rounds are separated by an exclusive-or (XOR) operation that modifies the current text word by a tweak word.

NSABC is put in public domain. As it bases on Skipjack, eventual users should be aware of patent(s) that may be possibly held by the U.S. Government and take steps to make sure the use is free of legal issues. We (the designers of NSABC) are not aware of any patent related to other parts of the design.

The rest of the article is organized as follows. Section 2 defines operations and notations. Section 3 specifies the cipher. Section 5 gives numerical examples. Section 4 suggests some implementation techniques. Section 6 concludes the article. Source code of software implementations are given in the Appendices.

2. Definitions

2.1. **Operations on words.** Throughout this article, w denotes the machine word length. We use the symbols $\boxplus$, $\boxminus$, $\boxtimes$ and $(.)^{-1}$ to denote addition, subtraction (and arithmetic negation), multiplication and multiplicative inversion, respectively, modulo 2^w (unless otherwise said). We use the symbols $\neg$ and $\oplus$ to denote bit-wise complement and exclusive-or (XOR) on w-bit operands (unless otherwise said). We write $x \lll n$ to denote leftward rotation (i. e. cyclic shift toward the most significant bit) of x, that is always a w-bit word, by n bits. For even w, the symbol $(.)^{\mathrm{S}}$ denotes swapping the high and low order halves, i.e. $x^{\mathrm{S}} = x \lll (w/2)$.

Let's define binary operation $\odot$ by

$$x \odot y = 2xy \boxplus x \boxplus y$$

and binary operation $\boxdot$ by

$$x \boxdot y = 2xy \boxplus x \boxminus y$$

The bivariate polynomials on the right hand side are permutation polynomials in either variable for every fixed value of the other variable [Riv99]. In other words, $\odot$ and $\boxdot$ are quasi-group operations.

Furthermore, $\odot$ is a group operation over the set of w-bit numbers[1]. This fact becomes obvious by considering an alternative definition for the $\odot$ operation [Mey97]: it can be done by dropping the rightmost bit, which is always "1", of the product modulo 2^{w+1} of the operands each appended with an "1" bit. Symbolically,

$$x \odot y = \left[(2x+1)(2y+1) - 1\right]/2 \pmod{2^w}$$

The group defined by $\odot$ is thus isomorphic to the multiplicative group of odd integers modulo 2^{w+1}, via the isomorphism

$$x \mapsto 2x + 1$$

The unit (i.e., identity) element of the group is 0. The inverse element of x, denoted $\bar{x}$, is

$$\bar{x} = \boxminus x(2x \boxplus 1)^{-1}$$

The following relations are obvious.

$$x \odot y = \boxminus \left[(\boxminus x) \boxdot y\right]$$

$$x \boxdot y = \boxminus \left[(\boxminus x) \odot y\right]$$

Since the unary operator $\boxminus$ is an involution, the following relations hold.

$$(x \boxdot y) \boxdot z = x \boxdot (y \odot z)$$

$$x \boxdot 0 = 0$$

[1] This fact, although simple and straightforward, does not seem to have been mentioned in the literature.

Notice that 0 is the right unit element w.r.t. the operation $\boxdot$. Hence

$$(x \boxdot y) \boxdot \bar{y} = x$$

which means that $\bar{y}$ is also the right inverse element of y w. r. t. the $\boxdot$ operation.

Since $(\neg x) \boxplus x = \boxminus 1$ holds for every x, the following relations hold.

$$(\neg x) \odot y = \neg(x \odot y)$$

$$(1 \boxminus x) \boxdot y = 1 \boxminus (x \boxdot y)$$

Let e be a fixed w-bit number. Let's define binary operations $\underset{e}{\odot}$ and $\underset{e}{\boxdot}$ by

$$x \underset{e}{\odot} y = (x \boxminus e) \odot (y \boxminus e) \boxplus e = 2xy \boxplus (1 - 2e)(x + y - e)$$

$$x \underset{e}{\boxdot} y = (x \boxplus e) \boxdot (y \boxminus e) \boxminus e = 2xy \boxplus (1 - 2e)(x - y + e)$$

Then $\underset{e}{\odot}$ and $\underset{e}{\boxdot}$ are quasi-group operations over the set of w-bit numbers. This follows from a more general fact that the right-hand side trivariate polynomials are permutations in either variable while keeping the other two fixed [Riv99]. Actually, the symbols $\underset{e}{\odot}$ and $\underset{e}{\boxdot}$ each defines an entire family of binary operations, of which each is uniquely determined by e.

Furthermore, from the definition it immediately follows that $\underset{e}{\odot}$ is a group operation, namely, the group is isomorphic to one defined by $\odot$ via the isomorphism

$$x \mapsto x \boxminus e$$

The unit element of the group is e.

The inverse element of x in the group, denoted $\frac{e}{x}$, is

$$\frac{e}{x} = \overline{x \boxminus e} \boxplus e = [(2e-1)x \boxminus 2e(e-1)] \boxtimes [2(x-e) \boxplus 1]^{-1}$$

Simple calculation proves the following relations.

$$x \underset{e}{\odot} y = \boxminus \left[(\boxminus x) \underset{e}{\boxdot} y \right]$$

$$x \underset{e}{\boxdot} y = \boxminus \left[(\boxminus x) \underset{e}{\odot} y \right]$$

$$(1 \boxminus 2e \boxminus x) \underset{e}{\boxdot} y = 1 \boxminus 2e \boxminus (x \underset{e}{\boxdot} y)$$

$$(x \underset{e}{\boxdot} y) \underset{e}{\boxdot} z = x \underset{e}{\boxdot} (y \underset{e}{\odot} z)$$

$$(x \underset{e}{\boxdot} y) \underset{e}{\boxdot} \frac{e}{y} = x$$

Notice that e is also the right unit element w. r. t. $\underset{e}{\boxdot}$, and $\frac{e}{y}$ also the right inverse element of y w. r. t. $\underset{e}{\boxdot}$. The operation $\underset{e}{\boxdot}$, which is non-commutative and non-associative, will be used for encryption and, due to the existence of right inversion, also for decryption.

2.2. **Order notations.** We write multi-part data values in *string* (or *number*) notation or *tuple* (or *vector*) notation. In string notation, the value is written as a sequence of symbols, possibly separated by space(s) that are insignificant. In tuple notation, the value is written as a sequence, in parentheses, of comma-separated symbols.

For examples, $z\ y\ x$ and 43 210 are in string notation, (x, y, z) and $(0, 1, 2, 3, 4)$ are in tuple notation.

The string notation indicates *high-first* order: the first (i.e. leftmost) symbol denotes the most significant part of the value when it is interpreted as a number.

Conversely, the tuple notation indicates *low-first* order: the first symbol denotes the least significant part of the value when it is interpreted as a number.

For examples, to interpret a 3-word number, x_2 denotes the most significant word of $x_2x_1x_0$ and x_0 denotes the least significant word of (x_0, x_1, x_2).

The same value may appear in either notation. Thus, for example, for every a, b, c and d,

$$a\,b\,c\,d = (d, c, b, a)$$

The term *part* introduced above usually refers to "word", but it may also refer to "digit" [of a number], "component" [of a tuple or vector], as well as group thereof. If, for example x, y, z, t are 1-digit, 2-digit, 3-digit and 4-digit values respectively, then $(x, y, z, t) = 9876543210$ means $x = 0$, $y = 21$, $z = 543$ and $t = 9876$.

Note that the "string notation" and "number notation" being used as synonyms does not mean that big-endian data ordering is mandated. In order to avoid security irrelevant details, we do not specify endianess. We nevertheless provide a "reference" implementations in C++, where every octet string is considered as a [generally multi-word] number with the first octet taken as the least significant one. The implementation thus interprets octet strings as numbers in little-endian order.

2.3. **Operations on word strings.** Let $(.)^{\mathrm{R}}$ denote the permutation that reverses the word order of a non-empty word string. For example, for $w = 8$,

$$\texttt{0x0123ABCD}^{\mathrm{R}} = \texttt{0xCDAB2301}$$

Let $(.)^{\mathrm{S}}$ denote the permutation that swaps the high order and low order halves of every word of a non-empty word string. For example, for $w = 8$,

$$\texttt{0x0123ABCD}^{\mathrm{S}} = \texttt{0x1032BADC}$$

The operator $\oplus$ on word strings denote word-wise application of $\oplus$. For example,

$$(a_0, a_1, a_2, ...) \oplus (b_0, b_1, b_2, ...) = (a_0 \oplus b_0, a_1 \oplus b_1, a_2 \oplus b_2 ...)$$

Unless otherwise said, operators $\boxplus$ and $\boxminus$ on word strings denote word-wise modular addition and subtraction, respectively. For example,

$$(a_0, a_1, a_2, \ldots) \boxplus (b_0, b_1, b_2, \ldots) = (a_0 \boxplus b_0, a_1 \boxplus b_1, a_2 \boxplus b_2, \ldots)$$

$$(a_0, a_1, a_2, \ldots) \boxminus (b_0, b_1, b_2, \ldots) = (a_0 \boxminus b_0, a_1 \boxminus b_1, a_2 \boxminus b_2, \ldots)$$

Let $\bar{(.)}$ denote the word-wise application of the inversion operator $\bar{(.)}$ on a word string. For example,

$$\overline{(a, b, c, ...)} = (\bar{a}, \bar{b}, \bar{c}, ...)$$

Given word strings E and X of the same length, let $\frac{E}{X}$ denote the word-wise application of the inversion operator $\frac{e}{x}$ on every word x of X with the index-matching word of E taken as the [right] unit element e. For example,

$$\frac{(e_1, e_2, e_3, \ldots)}{(x_1, x_2, x_3, \ldots)} = \left(\frac{e_1}{x_1}, \frac{e_2}{x_2}, \frac{e_3}{x_3}, \ldots\right)$$

Operations on word strings are used in this article only to express the decryption function explicitly.

3. Specification

This section provides details of NSABC/w. From now on w, the word length, must be even.

Throughout this article, X denotes a $4w$-bit plaintext block, Y a $4w$-bit ciphertext block, Z a $5w$-bit key, T a $4w$-bit secret *tweak*, i.e., a value that is used to encrypt only one block under the key, U a w-bit *unit key*, i.e. an additional key that generates right unit elements for the underlying quasi-groups.

Tweaking is optional. It may be disabled by keeping T constant (like Z and U) while encrypting many blocks. When tweaking is disabled, NSABC becomes a conventional, non-tweakable, block cipher.

Mathematically, the cipher is given by two functions,

ENCRYPT(X, Z, T, U), which encrypts X under control of Z, T and U,
DECRYPT(Y, Z, T, U), which decrypts Y under control of Z, T and U,

satisfying the apparent relation

$$\text{DECRYPT}(\text{ENCRYPT}(X, Z, T, U), Z, T, U) = X$$

The function ENCRYPT is defined in terms of four functions:

CRYPT,	a *text encryption* function that encrypts a plaintext block using a *key schedule*, a *unit schedule* and a *tweak schedule*;
KE,	a *key expansion* function, that generates the key schedule from Z;
UE,	a *unit element* function, that generates the unit schedule from U; and
TE,	a *tweak expansion* function, that generates the tweak schedule from T.

Algorithm 1 Function ENCRYPT

Input:

X	$4w$-bit plaintext block
Z	$5w$-bit key
T	$4w$-bit tweak
U	w-bit unit key

Output:

Y	$4w$-bit ciphertext block

Relation:

$$\text{ENCRYPT}(X, Z, T, U) = \text{CRYPT}(X,\ \text{KE}(Z), \text{UE}(U), \text{TE}(T))$$

An explicit relation for ENCRYPT is given in Algorithm 1. An explicit relation for DECRYPT is given in Algorithm 7.

Mechanically, encryption is performed on a conceptual processor with a 4-word *text register* (x_0, x_1, x_2, x_3), a 5-word *key register* $(z_0, z_1, z_2, z_3, z_4)$, a 4-word *tweak register* (t_0, t_1, t_2, t_3) and a word *unit register* u. The key register is initially loaded with the key Z. The tweak register is initially loaded with the tweak T. The unit register is initially loaded with the unit key U. The text register is initially loaded with the plaintext block X and finally it contains the ciphertext block Y.

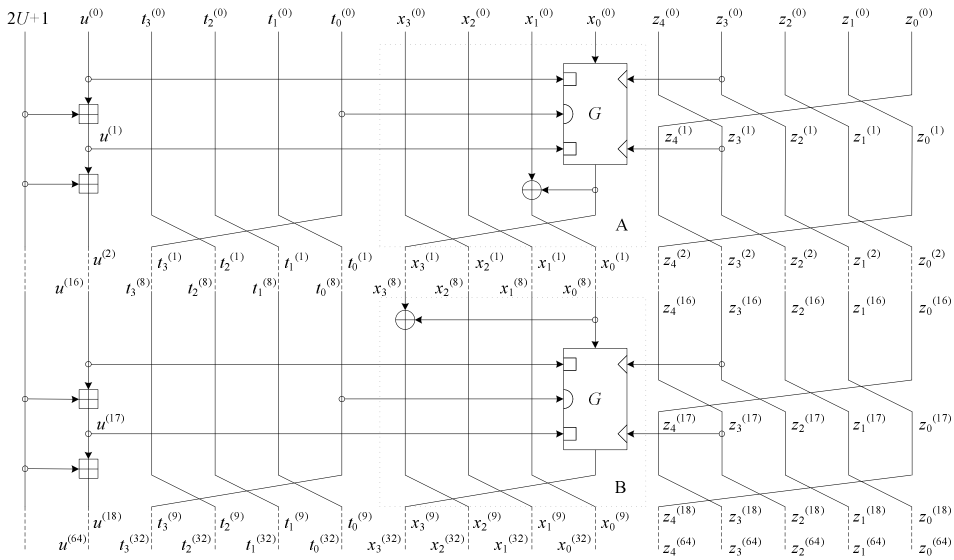

FIGURE 3.1. Representative rounds.

The concrete, vector, notation here specifies the order of words so, for example, x_0 is initially loaded with the least significant word of X and finally it contains the least significant word of Y.

3.1. **Text encryption.** The text register (x_0, x_1, x_2, x_3) is initially loaded with the plaintext block X and finally it contains the ciphertext block Y.

Text encryption proceeds in 32 rounds of operations. A round is of either *type A* or *type B*. The rounds are arranged in four *passes*: firstly eight rounds of type A, then eight rounds of type B, then eight rounds of type A again, finally eight rounds of type B again.

For k-th round, $0 \leq k \leq 31$, the text word x_0 is permuted, i.e. it is updated by an execution unit called *G-box* that implements a permutation G on the set of word values, and the contents of the text word x_0 are mixed, by exclusive-or (XOR), into an other text word that is either x_1 or x_3. The order of operations and the target of mixing depend on the round type:

- For an A-typed round (see Figure 3.1 part A), G applies first, then the mixing takes place and targets x_1. That is, the contents of x_0 enters the G-box, the output value of the G-box is stored back to x_0, then the contents of x_0 and x_1 are XOR'ed and the result is stored to x_1. The words x_2 and x_3 are left unchanged.
- For a B-typed round (see Figure 3.1 part B), the mixing takes place and targets x_3 first, then G applies. That is, the contents of x_0 and x_3 are XOR'ed and the result is stored to x_3, then the contents of x_0 enters the G-box, the output value of the G-box is stored back to x_0. The words x_1 and x_2 are left unchanged.

Besides the text input, the G-box also takes as its inputs an ordered pair of w-bit *key words* (K_{2k}, K_{2k+1}) (depicted by $\triangleleft$ in Fig. 3.1 and 3.2), an ordered pair of w-bit *unit words* (L_{2k}, L_{2k+1}) (depicted by $\square$ in Fig. 3.1 and 3.2), and a w-bit *tweak word* C_k (depicted by D in Fig. 3.1 and 3.2). The details on how key words, unit words and tweak words are generated and used will be given in the subsequent subsections.

Algorithm 2 Function CRYPT (text encryption)

Input:

X	$4w$-bit plaintext block
K	$64w$-bit key schedule
L	$64w$-bit unit schedule
C	$32w$-bit tweak schedule

Output:

Y	$4w$-bit ciphertext block

Pseudo-code:

$(x_0, x_1, x_2, x_3) \leftarrow X$
`for` $k \leftarrow 0, 1, 2, \ldots, 31$ `loop`
  `if` $0 \leqslant k < 8 \ \vee \ 16 \leqslant k < 24$ `then`
    $x_0 \leftarrow G(x_0, (K_{2k}, K_{2k+1}), (L_{2k}, L_{2k+1}), C_k)$
    $x_1 \leftarrow x_1 \oplus x_0$
  `elsif` $8 \leqslant k < 16 \ \vee \ 24 \leqslant k < 32$ `then`
    $x_3 \leftarrow x_3 \oplus x_0$
    $x_0 \leftarrow G(x_0, (K_{2k}, K_{2k+1}), (L_{2k}, L_{2k+1}), C_k)$
  `end if`
  $(x_0, x_1, x_2, x_3) \leftarrow (x_1, x_2, x_3, x_0)$
`end loop`
$Y \leftarrow (x_0, x_1, x_2, x_3)$

Relations:

$Y = (x_0^{(32)}, x_1^{(32)}, x_2^{(32)}, x_3^{(32)})$
`For` $0 \leqslant k < 8 \ \vee \ 16 \leqslant k < 24$:
  $x_0^{(k+1)} = x_1^{(k)} \oplus g^{(k)}$
  $x_1^{(k+1)} = x_2^{(k)}$
  $x_2^{(k+1)} = x_3^{(k)}$
  $x_3^{(k+1)} = g^{(k)}$
`For` $8 \leqslant k < 16 \ \vee \ 24 \leqslant k < 32$:
  $x_0^{(k+1)} = x_1^{(k)}$
  $x_1^{(k+1)} = x_2^{(k)}$
  $x_2^{(k+1)} = x_3^{(k)} \oplus x_0^{(k)}$
  $x_3^{(k+1)} = g^{(k)}$
`For` $0 \leqslant k < 32$:
  $g^{(k)} = G(x_0^{(k)}, (K_{2k}, K_{2k+1}), (L_{2k}, L_{2k+1}), C_k)$
$(x_0^{(0)}, x_1^{(0)}, x_2^{(0)}, x_3^{(0)}) = X$
$(K_0, K_1, K_2, \ldots, K_{63}) = K$
$(L_0, L_1, L_2, \ldots, L_{63}) = L$
$(C_0, C_1, C_2, \ldots, C_{31}) = C$

The encryption round is completed with a rotation by one word toward the least significant word on the text register, i.e. the text register is modified by simultaneous loading the word x_0 with the contents of the word x_1, x_1 with the contents of x_2, x_2 with the contents of x_3, and x_3 with the contents of x_0.

3.2. **Tweak schedule.** The tweak register (t_0, t_1, t_2, t_3) is initially loaded with the tweak T. The tweak words are generated in 32 rounds of operations.

For k-th round, $0 \leq k \leq 31$, the value of the word t_0 of the tweak register is taken as the tweak word C_k [which enters the G-box in the k-th encryption round]. Then,

Algorithm 3 Function TE (tweak expansion)

Input:
T $4w$-bit tweak
Output:
C $32w$-bit tweak schedule
Pseudo-code:

```
(t_0, t_1, t_2, t_3) ← T
for k ← 0, 1, 2, ..., 31 loop
  C_k ← t_0
  (t_0, t_1, t_2, t_3) ← (t_1, t_2, t_3, t_0)
end loop
```

Relations:

$C = (C_0, C_1, C_2, \ldots, C_{31})$
For $0 \leqslant k < 32$:
$C_k = t_0^{(k)}$
$t_0^{(k+1)} = t_1^{(k)}$
$t_1^{(k+1)} = t_2^{(k)}$
$t_2^{(k+1)} = t_3^{(k)}$
$t_3^{(k+1)} = t_0^{(k)}$
$(t_0^{(0)}, t_1^{(0)}, t_2^{(0)}, t_3^{(0)}) = T$

similarly to the text register, the tweak register is rotated by one word toward the least significant word (see Figure 3.1).

NOTE. For $T_3T_2T_1T_0 = T$, the tweak schedule is

$$\mathrm{TE}(T) = (T_0, T_1, T_2, T_3, T_0, T_1, T_2, T_3, \ldots, T_0, T_1, T_2, T_3)$$

3.3. **Key schedule.** The key register $(z_0, z_1, z_2, z_3, z_4)$ is initially loaded with the key Z. The key words are generated in 64 rounds of operations.

For k-th round, $0 \le k \le 63$, the value of the word z_3 of the key register is taken as the key word K_k [which enters the G-box of the $k/2$-th encryption round as the first key word if k is even, or as the second key word if k is odd]. The key register is then rotated by one word toward the least significant word. The rotation is similar to that on the text register and the tweak register. (See Figure 3.1.)

NOTE. For $Z_4Z_3Z_2Z_1Z_0 = Z$, the key schedule is

$$\mathrm{KE}(Z) = (Z_3, Z_4, Z_0, Z_1, Z_2, Z_3, Z_4, Z_0, Z_1, Z_2, \ldots, Z_3, Z_4, Z_0, Z_1)$$

3.4. **Unit schedule.** The unit register u is initially loaded with the unit key U.

Unit words are generated in 64 rounds of operations.

For k-th round, $0 \le k \le 63$, the value of the unit register u is taken as the unit word L_k [which, similarly to the key word K_k, enters the G-box of $k/2$-th encryption round as the first unit word if k is even, or as the second unit word if k is odd]. The register is then added modulo 2^w by the [key-dependent] constant $2U \boxplus 1$ to become ready for the next round. (See Figure 3.1.)

NOTE. Given unit key U, the unit schedule is

$$\mathrm{UE}(U) = (U, 3U \boxplus 1, 5U \boxplus 2, 7U \boxplus 3, \ldots, 127U \boxplus 63)$$

Algorithm 4 Function KE (key expansion)

Input:

Z $5w$-bit key

Output:

K $64w$-bit key schedule

Pseudo-code:

$(z_0, z_1, z_2, z_3, z_4) \leftarrow Z$
`for` $k \leftarrow 0, 1, 2, \ldots, 63$ `loop`
  $K_k \leftarrow z_3$
  $(z_0, z_1, z_2, z_3, z_4) \leftarrow (z_1, z_2, z_3, z_4, z_0)$
`end loop`

Relations:

$K = (K_0, K_1, K_2, \ldots, K_{63})$
`For` $0 \leqslant k < 64$:
  $K_k = z_3^{(k)}$
  $z_0^{(k+1)} = z_1^{(k)}$
  $z_1^{(k+1)} = z_2^{(k)}$
  $z_2^{(k+1)} = z_3^{(k)}$
  $z_3^{(k+1)} = z_4^{(k)}$
  $z_4^{(k+1)} = z_0^{(k)}$
$(z_0^{(0)}, z_1^{(0)}, z_2^{(0)}, z_3^{(0)}, z_4^{(0)}) = Z$

Algorithm 5 Function UE (unit element)

Input:

U w-bit unit key

Output:

L $64w$-bit unit schedule

Pseudo-code:

$u \leftarrow U$
`for` $k \leftarrow 0, 1, 2, \ldots, 63$ `loop`
  $L_k \leftarrow u$
  $u \leftarrow u \boxplus 2U \boxplus 1$
`end loop`

Relations:

$L = (L_0, L_1, L_2, \ldots, L_{63})$
`For` $k = 0, 1, 2, \ldots, 63$:
  $L_k = u^{(k)}$
  $u^{(k+1)} = u^{(k)} \boxplus 2U \boxplus 1$
$u^{(0)} = U$
`Or, equivalently, for every` k:
  $u^{(k)} = U \odot k$

3.5. **G-box.** The G-box implements a permutation G (see Figure 3.2) that takes as argument a text word and is parametrized by an ordered pair of key words (K_0, K_1), an ordered pair of unit words (L_0, L_1) and a tweak word C_0 to return a text word as the result. The G-box operates on a word register that initially contains the argument and finally contains the result. The G-box proceeds in two

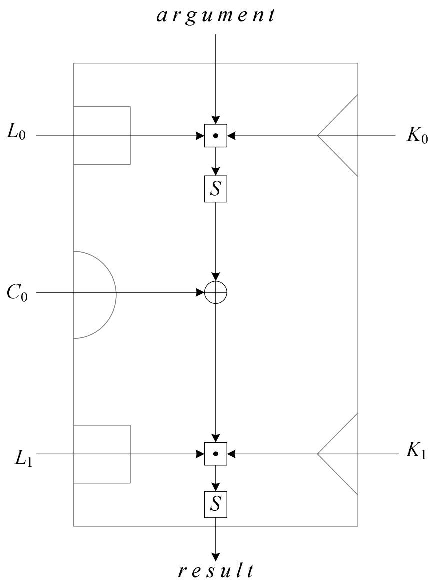

FIGURE 3.2. Permutation G.

rounds, each consisting of an operation $\underset{e}{\boxdot}$ followed by a half-word swap S. The two rounds are separated by an exclusive-or (XOR) operation.

For the first round, the operation $\underset{e}{\boxdot}$ takes the contents of the register as the left operand, K_0 as the right operand, and L_0 as its right unit element. The result is stored back to the register. The register is then modified by operation S, i.e. swapping the contents of its high and low order halves.

For the inter-round XOR operation, the register is modified by XOR'ing its contents with the tweak word C_0 and storing the result back to it.

For the second round, the register is processed similarly to the first round with K_1 and L_1 being used instead of K_0 and L_0, respectively.

NOTES.

(1) The cipher uses 64 distinct instances from the family of operations $\underset{e}{\boxdot}$.
(2) Alternatively, it may be seen as using 64 identical instances of the single operation $\boxdot$ or $\odot$, but operands and result of each instance are "biased" by adding or subtracting the constant L_0 (or L_1) that is specific to the instance, and furthermore, being seen as $\odot$, the left operand enters and the result leaves it in altered sign.
(3) Like Skipjack, the G-box permutes K_0 (or K_1) while keeping x and other parameters fixed. Unlike Skipjack, the G-box doesn't permute the word $(\mathrm{Hi}(K_0), \mathrm{Lo}(K_1))$ where Hi(.) and Lo(.) stand for the high and the low order half respectively.
(4) Unlike Skipjack, diffusion in the G-box is incomplete, i.e. not every input bit affects all output bits. Indeed, the v-th bit of the argument, with $v > w/2$, affects only all bits of the low order half and bits v through $w-1$ of the result; bits $w/2$ through $v-1$ remain unaffected.
(5) If $(K_0, K_1) = (L_0, L_1) \wedge C_0 = 0$ then G becomes the identity.

3.6. **Decryption.** Decryption can be easily derived from encryption. Namely, if

$$Y = \mathrm{CRYPT}(X, \mathrm{KE}(Z), \mathrm{UE}(U), \mathrm{TE}(T))$$

Algorithm 6 Permutation G

Input:

x	w-bit text word
(K_0, K_1)	pair of w-bit key words
(L_0, L_1)	pair of w-bit unit words
C_0	w-bit tweak word

Output:

y	w-bit text word

Pseudo-code:

$x \leftarrow x \underset{L_0}{\boxdot} K_0$
$x \leftarrow x^{\mathrm{S}}$
$x \leftarrow x \oplus C_0$
$x \leftarrow x \underset{L_1}{\boxdot} K_1$
$x \leftarrow x^{\mathrm{S}}$
$y \leftarrow x$

Relation:

$G(x, (K_0, K_1), (L_0, L_1), C_0) = (((x \underset{L_0}{\boxdot} K_0)^{\mathrm{S}} \oplus C_0) \underset{L_1}{\boxdot} K_1)^{\mathrm{S}}$

Algorithm 7 Function DECRYPT

Input:

Y	$4w$-bit ciphertext block
Z	$5w$-bit key
T	$4w$-bit tweak
U	w-bit unit key

Output:

X	$4w$-bit plaintext block

Relation:

$$\mathrm{DECRYPT}(Y, Z, T, U) = \mathrm{CRYPT}(Y^{\mathrm{RS}}, \frac{\mathrm{UE}(U)^{\mathrm{R}}}{\mathrm{KE}(Z^{\mathrm{R}})}, \mathrm{UE(U)}^{\mathrm{R}}, \mathrm{TE}(T^{\mathrm{RS}}))^{\mathrm{RS}}$$

then it immediately follows that

$$\mathrm{CRYPT}(Y^{\mathrm{RS}}, \frac{\mathrm{UE}(U)^{\mathrm{R}}}{\mathrm{KE}(Z^{\mathrm{R}})}, \mathrm{UE(U)}^{\mathrm{R}}, \mathrm{TE}(T^{\mathrm{RS}})) = X^{\mathrm{RS}}$$

In other words, encrypting the cipher block in reverse half-word order (Y^{RS}) using the tweak in reverse half-word order (T^{RS}), the unit schedule in reverse word order ($\mathrm{UE}(U)^{\mathrm{R}}$), and the key schedule consisting of inverse words of one expanded from the key in reverse word order (Z^{R}), where the inversions are of the quasi-groups defined by the operation $\underset{e}{\boxdot}$ and each quasi-group is uniquely given by its right unit that is the index-matching word of the encryption unit schedule in reverse word order ($\mathrm{UE}(U)^{\mathrm{R}}$), recovers the plain block in reverse half-word order (X^{RS}).
NOTES.

(1) The full cipher is illustrated in Figure 3.3, where $X_3X_2X_1X_0 = X$, $Y_3Y_2Y_1Y_0 = Y$, $Z_4Z_3Z_2Z_1Z_0 = Z$ and $T_3T_2T_1T_0 = T$ (the unit schedule is omitted). The figure is obtained by "unrolling" (i.e. eliminating all rotations of) the dataflow graph of the full cipher that would be obtained by cascading the individual rounds as in Figure 3.1.

(2) The overall structure is up to word indexing identical to that of Skipjack. The word re-indexing, which is cryptographically insignificant, was introduced to ease description and illustration.
(3) Like Skipjack, decryption is similar to encryption. To decrypt with Skipjack, one swaps adjacent words in the cipher and the plain block and swaps adjacent word pairs in the key. To decrypt with NSABC, one reverses the word order, i.e., swaps the first and the last words as well as the second first and the second last ones in the text block, the tweak and the key. For Skipjack, one also swaps high and low order halves of every word. For NSABC, one swaps high and low order halves of every word but that of the key.
(4) Unlike Skipjack, just swapping the words and half-words doesn't turn encryption into decryption – one needs to invert key words too. Thus although ENCRYPT and DECRYPT can be expressed explicitly in terms of CRYPT, DECRYPT cannot be expressed explicitly in terms of ENCRYPT.

3.7. **Tweak derivation.** The $4w$-bit secret tweak T is used to encrypt only one block [under a given key Z and unit key U]. In order to encrypt multiple blocks the tweak is derived from the block index and a $4w$-bit [additional] key, called *tweak key*, as follows. Let $T^{(j)}$ denote the tweak used to encrypt j-th block. For the first block ($j = 0$), the tweak key is used as the tweak directly:

$$T^{(0)} = \text{tweak key}$$

The subsequent tweak is computed from the current tweak by the recurrent relation:

$$T^{(j+1)} = T^{(j)} \boxplus 2T^{(0)} \boxplus 1$$

or, equivalently,

$$T^{(j)} = T^{(0)} \odot j$$

where all operands are regarded as $4w$-bit numbers and all operators are defined on $4w$-bit arithmetic, i.e. mod 2^{4w}.
NOTES.

(1) The third relation, where $T^{(0)}$ conveniently designates the [unnamed] tweak key, is meant for random access. The family of functions $\{T : j \mapsto T^{(0)} \odot j\}$, parametrized by the tweak key $T^{(0)}$, is not ϵ-almost 2-XOR universal according to definition in [LRW02]. Eventual application of this family in the Liskov-Rivest-Wagner construction, i.e. encryption by $\mathrm{CRYPT}(X \oplus T^{(j)}, \mathrm{KE}(Z), 0, \mathrm{UE}(U)) \oplus T^{(j)}$, is therefore impossible.
(2) For efficient random access, applications may opt to use non-flat spaces of the block index j. For example, an application that encrypts relational databases may define the index in the format $j = j_4\, j_3\, j_2\, j_1\, j_0$, where j_4 is database number, j_3 is table number within the database, j_2 is row number within the table, j_1 is field number within the row and j_0 is block number within the field.
(3) Tweaking must be disabled when the cipher is used as a permutation, i. e. to generate a sequence of unique numbers.
(4) Tweaking should be enabled in all other modes of operation. For example, a non-tweakable block cipher can generate a sequence of independent numbers by encrypting a counter block in Cipher Block Chaining (CBC) mode; NSABC can generate a similar sequence with virtually the same cycle length by encrypting a constant block in a "tweaked CBC" mode.

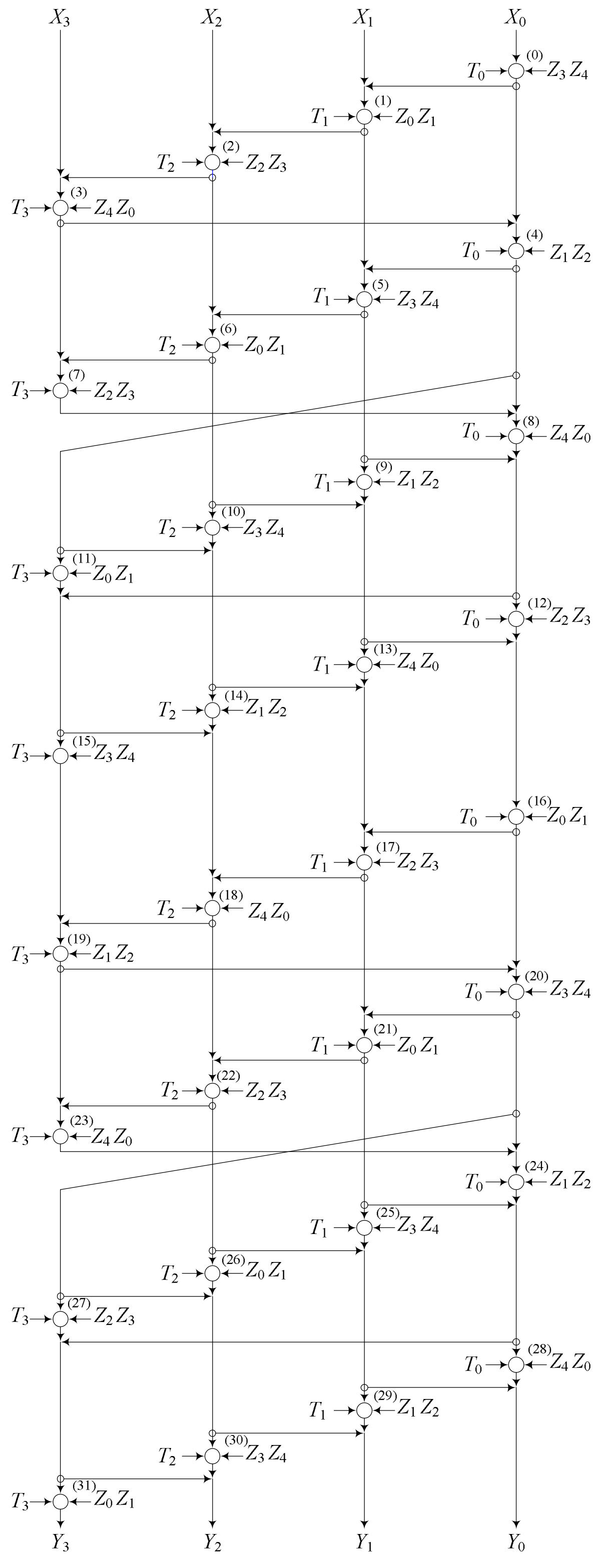

FIGURE 3.3. The full cipher, by "unrolling".

4. Example

An encipherment in NSABC/16 with

$$X = \texttt{0x0123456789ABCDEF}$$
$$Z = \texttt{0x88880777006600050000}$$
$$T = \texttt{0x0001002203334444}$$
$$U = \texttt{0x1998}$$

results in

$$Y = \texttt{0x88B14E700F51921E}$$

Table 1 lists states of the [conceptual] processor during the encipherment, i.e. the contents of all registers at the start of round k for $k = 0, 1, 2, ..., 64$ for key schedule and unit schedule, and $k = 0, 1, 2, ..., 32$ for tweak schedule and text encryption. The start of round 64 (32) conveniently means the end of round 63 (31), which is that of the entire algorithm.

5. Notes on implementation

This section provides methods for efficient software implementation for two types of environment: memory-constrained, such as embedded computers, and memory-abundant, such as servers and personal computers.

5.1. **Memory-constrained environment.** The function ENCRYPT can be implemented without using any writeable memory on a processor with at least 16 word registers:

- 4 for (x_0, x_1, x_2, x_3) – text register
- 5 for $(z_0, z_1, z_2, z_3, z_4)$ – key register
- 4 for (t_0, t_1, t_2, t_3) – tweak register
- 1 for u – unit register,
- 1 for the constant value $2U \boxplus 1$, and
- 1 for k – round index.

Indeed, the schedules K, L and C can vanish because every word of them, once produced, can be consumed immediately, provided that the functions KE, TE, UE and CRYPT are programmed to run in parallel and synchronized with each increment of k. Source code of this implementation is given in Appendix A.

Unlike ENCRYPT, DECRYPT needs memory for the key schedule because on-the-fly modular multiplicative inversion is too slow to be practical. In this environment, modes of operation that avoid DECRYPT (i.e. ones using ENCRYPT to decrypt) are thus preferable.

5.2. **Memory-abundant environment.** The quasi-group operation $\underset{e}{\boxdot}$ can be evaluated by only one multiplication and one addition. Indeed,

$$x \underset{e}{\boxdot} z = mx \boxplus n$$

where x is a text word, z key word, and

$$m = 2(z - e) \boxplus 1$$

$$n = (2e - 1) \boxtimes (z - e)$$

So, instead of using the (z, e) pairs, one may pre-compute the (m, n) pairs once and use them many times.

Table 1. Processor states during an encipherment by NSABC/16

```
=== ==== ==================== === ================ ================
    Unit         Key register       Tweak register    Text register
  k    u   z4  z3  z2  z1  z0   k   t3  t2  t1  t0   x3  x2  x1  x0
=== ==== ==================== === ================ ================
  0 1998 88880777006600050000   0 0001002203334444 0123456789ABCDEF
  1 4CC9 00008888077700660005
  2 7FFA 00050000888807770066   1 4444000100220333 388401234567B12F
  3 B32B 00660005000088880777
  4 E65C 07770066000500008888   2 0333444400010022 1E90388401235BF7
  5 198D 88880777006600050000
  6 4CBE 00008888077700660005   3 0022033344440001 60AC1E903884618F
  7 7FEF 00050000888807770066
  8 B320 00660005000088880777   4 0001002203334444 499160AC1E907115
  9 E651 07770066000500008888
 10 1982 88880777006600050000   5 4444000100220333 C2D7499160ACDC47
 11 4CB3 00008888077700660005
 12 7FE4 00050000888807770066   6 0333444400010022 F1EFC2D749919143
 13 B315 00660005000088880777
 14 E646 07770066000500008888   7 0022033344440001 03D2F1EFC2D74A43
 15 1977 88880777006600050000
 16 4CA8 00008888077700660005   8 0001002203334444 273D03D2F1EFE5EA
 17 7FD9 00050000888807770066
 18 B30A 00660005000088880777   9 4444000100220333 1615C2D703D2F1EF
 19 E63B 07770066000500008888
 20 196C 88880777006600050000  10 0333444400010022 A9B6E7FAC2D703D2
 21 4C9D 00008888077700660005
 22 7FCE 00050000888807770066  11 0022033344440001 18C0AA64E7FAC2D7
 23 B2FF 00660005000088880777
 24 E630 07770066000500008888  12 0001002203334444 B049DA17AA64E7FA
 25 1961 88880777006600050000
 26 4C92 00008888077700660005  13 4444000100220333 851857B3DA17AA64
 27 7FC3 00050000888807770066
 28 B2F4 00660005000088880777  14 0333444400010022 71F82F7C57B3DA17
 29 E625 07770066000500008888
 30 1956 88880777006600050000  15 0022033344440001 D5F0ABEF2F7C57B3
 31 4C87 00008888077700660005
 32 7FB8 00050000888807770066  16 0001002203334444 E5118243ABEF2F7C
 33 B2E9 00660005000088880777
 34 E61A 07770066000500008888  17 4444000100220333 94FAE51182433F15
 35 194B 88880777006600050000
 36 4C7C 00008888077700660005  18 0333444400010022 7BB394FAE511F9F0
 37 7FAD 00050000888807770066
 38 B2DE 00660005000088880777  19 0022033344440001 11747BB394FAF465
 39 E60F 07770066000500008888
 40 1940 88880777006600050000  20 0001002203334444 D14F11747BB345B5
 41 4C71 00008888077700660005
 42 7FA2 00050000888807770066  21 4444000100220333 0385D14F11747836
 43 B2D3 00660005000088880777
 44 E604 07770066000500008888  22 0333444400010022 873B0385D14F964F
 45 1935 88880777006600050000
 46 4C66 00008888077700660005  23 0022033344440001 CB9B873B03851AD4
 47 7F97 00050000888807770066
 48 B2C8 00660005000088880777  24 0001002203334444 D6FCCB9B873BD579
 49 E5F9 07770066000500008888
 50 192A 88880777006600050000  25 4444000100220333 D4CF0385CB9B873B
 51 4C5B 00008888077700660005
 52 7F8C 00050000888807770066  26 0333444400010022 779F53F40385CB9B
 53 B2BD 00660005000088880777
 54 E5EE 07770066000500008888  27 0022033344440001 8CECBC0453F40385
 55 191F 88880777006600050000
 56 4C50 00008888077700660005  28 0001002203334444 C93A8F69BC0453F4
 57 7F81 00050000888807770066
 58 B2B2 00660005000088880777  29 4444000100220333 2E1A9ACE8F69BC04
 59 E5E3 07770066000500008888
 60 1914 88880777006600050000  30 0333444400010022 8038921E9ACE8F69
 61 4C45 00008888077700660005
 62 7F76 00050000888807770066  31 0022033344440001 D4BE0F51921E9ACE
 63 B2A7 00660005000088880777
 64 E5D8 07770066000500008888  32 0001002203334444 88B14E700F51921E
```

The cipher is parallelizable. The following procedure executes all 32 rounds in 20 steps, of which half performing two or three parallel evaluations of G. Recall that $g^{(k)}$ is the result of G in round k.

(1) Compute $g^{(0)}$
(2) Compute $g^{(1)}$
(3) Compute $g^{(2)}$
(4) Compute $g^{(3)}$
(5) Compute $g^{(4)}$
(6) Compute $g^{(5)}$, $g^{(11)}$ in parallel
(7) Compute $g^{(6)}$, $g^{(9)}$ in parallel
(8) Compute $g^{(7)}$, $g^{(10)}$, $g^{(13)}$ in parallel
(9) Compute $g^{(8)}$, $g^{(14)}$ in parallel
(10) Compute $g^{(12)}$, $g^{(15)}$ in parallel
(11) Compute $g^{(16)}$
(12) Compute $g^{(17)}$
(13) Compute $g^{(18)}$
(14) Compute $g^{(19)}$
(15) Compute $g^{(20)}$
(16) Compute $g^{(21)}$, $g^{(27)}$ in parallel
(17) Compute $g^{(22)}$, $g^{(25)}$ in parallel
(18) Compute $g^{(23)}$, $g^{(26)}$, $g^{(29)}$ in parallel
(19) Compute $g^{(24)}$, $g^{(30)}$ in parallel
(20) Compute $g^{(28)}$, $g^{(31)}$ in parallel

The procedure becomes evident by examining the dataflow graph of the cipher, shown in Figure 5.1, which is obtained by "unrolling" the one in Figure 3.3. Here "unrolling" means introducing a rotation so that the G-boxes with congruent round indices (mod 3) lay on a straight line.

On a x86-64 processor in 32-bit mode ($w = 32$), the procedure takes about 256 clock cycles, i.e. $256/16 = 16$ clock cycles per byte encrypted. (The source code of this implementation is given in Appendix B.) In 64-bit mode ($w = 64$), it takes about 384 clock cycles, i.e. $384/32 = 12$ clock cycles per byte.

The procedure may be also coded twice, i.e. it may be run in two instances in parallel on a single core of the processor, with the second instance delayed by a few steps after the first, to encrypt two blocks possibly under different tweaks and/or keys. This method has shown to be effective for x86-64 processors in 64-bit mode, resulting in about 9 clock cycles per byte.

6. Conclusion

We defined NSABC, a block cipher utilizing a group operation that is essentially modular multiplication of machine words, a powerful operation available on many processors.

NSABC was meant to be elegant. It uses no S-boxes or "magic" constants. It uses only machine word-oriented algebraic operations. It makes use of the simple and regular structure of Skipjack which has become publicly known for over a decade — sufficient time to be truly understood. It is elegant to be easily memorizable, realizable and analyzable.

NSABC bases on some valuable design of a well-reputed agency in the branch. We therefore believe that it is worth analysis and it can withstand rigorous analysis. If this happens to be true, then we may have a practical cipher with 256-bit blocks, allowing to encrypt enormous amount of data under the same key, and with 320-bit keys, allowing to protect data over every imaginable time.

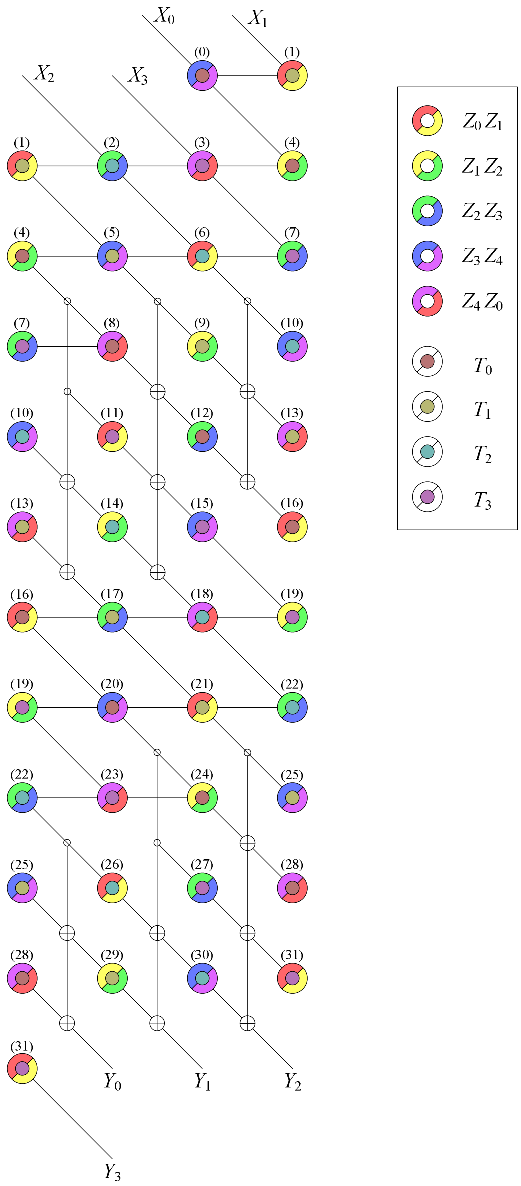

FIGURE 5.1. The full cipher, by another "unrolling".

In cipher design there is always a trade-off between security and efficiency, and designers always have to ask: "What do we want, a very strong and fairly fast cipher, or fairly strong but very fast?"

NSABC reflects the authors' view on the dilemma. If Skipjack is regarded as very strong and just fairly fast, then NSABC may be regarded as a design emphasizing the second aspect — make it very fast, abeit just fairly strong. For w-bit word length, NSABC key length is $5w$ bits, optionally plus $5w$ bits more, whilst the true level of security is yet to be determined. On the other hand, on a modern 64-bit processor it takes only 9 clock cycles to encrypt a byte.

NSABC is thus fast to be comparable to every modern block cipher.

References

[IBM98] Burwick C., Coppersmith D., D'Avignon E., Gennaro G., Halevi S., Jutla C., Matyas S. M. Jr., O'Connor L., Peyravian M., Safford D., Zunic N.: MARS — a candidate cipher for AES. 1998.

[LM91] Lai X., Massey J. L.: A proposal for a new block encryption standard. 1991.

[LMM91] Lai X., Massey J. L., Murphy S.: Markov ciphers and differential cryptanalysis. 1991.

[LRW02] Liskov M., Rivest R. L., Wagner D.: Tweakable block ciphers. 2002.

[Mey97] Meyers J. H.: Modifying IDEA. Discussion on Usenet. 1997.

[NSA98] National Security Agency: Skipjack and KEA specification. 1998.

[Riv99] Rivest R. L.: Permutation polynomials modulo 2^w. 1999.

[Sha49] Shannon C. E.: Communication theory of secrecy systems. 1949.

APPENDIX A. A reference implementation of NSABC/32 — ENCRYPT only

```
typedef uint32_t word;
//--------------------------------------------------------------------
static word o(word x, word y, word e)
{
    return 2*x*y + (1 - 2*e)*(x - y + e);
}
//--------------------------------------------------------------------
static word G(word x, word z0, word z1, word u0, word u1, word t)
{
    x = o(x,z0,u0);
    x = _rotl(x,16);
    x ^= t;
    x = o(x,z1,u1);
    x = _rotl(x,16);
    return x;
}
//--------------------------------------------------------------------
void encrypt( word Y[4], word const X[4], word const Z[5],
              word const T[4], word U )
{
    word
        x0 = X[0], x1 = X[1], x2 = X[2], x3 = X[3],
        z0 = Z[0], z1 = Z[1], z2 = Z[2], z3 = Z[3], z4 = Z[4],
        t0 = T[0], t1 = T[1], t2 = T[2], t3 = T[3],
        u0 = U,
        u1 = u0 + 2*U + 1;
    for(int k = 0; k < 32; k++)
    {
        if(k & 8)           // B-round
        {
            x3 ^= x0;
            x0 = G(x0, z3, z4, u0, u1, t0);
        }
        else                // A-round
        {
            x0 = G(x0, z3, z4, u0, u1, t0);
            x1 ^= x0;
        }
        word x = x0; x0 = x1; x1 = x2; x2 = x3; x3 = x;
        word z = z0; z0 = z2; z2 = z4; z4 = z1; z1 = z3; z3 = z;
        word t = t0; t0 = t1; t1 = t2; t2 = t3; t3 = t;
        u0 = u1 + 2*U + 1;
        u1 = u0 + 2*U + 1;
    }
    Y[0] = x0; Y[1] = x1; Y[2] = x2; Y[3] = x3;
}
```

APPENDIX B. An optimized implementation of NSABC/32

```
void expandkey   (word M[64], word N[64], word const Z[5], word U)
{
    word z0=Z[0], z1=Z[1], z2=Z[2], z3=Z[3], z4=Z[4];
    word u = U;
```

```
    for( int k=0; k<64; k++ )
    {
        M[k] = 2*(z3 - u) + 1;
        N[k] = (2*u - 1)*(z3 - u);
        u += 2*U + 1;
        word z=z0; z0=z1; z1=z2; z2=z3; z3=z4; z4=z;
    }
}
//---------------------------------------------------------------------
static inline
word G( word x, word t, word m0, word m1, word n0, word n1 )
{
    x *= m0;
    x += n0;
    x = _rotl(x,16);
    x ^= t;
    x *= m1;
    x += n1;
    x = _rotl(x,16);
    return x;
}
//---------------------------------------------------------------------
void crypt( word Y[4], word const X[4], word const T[4],
            word const M[64], word const N[64])
{
    // Step 1
    word const g0 = G(X[0],              T[0],  M[0], M[1], N[0], N[1]);
    // Step 2
    word const g1 = G(X[1]^g0,           T[1],  M[2], M[3], N[2], N[3]);
    // Step 3
    word const g2 = G(X[2]^g1,           T[2],  M[4], M[5], N[4], N[5]);
    // Step 4
    word const g3 = G(X[3]^g2,           T[3],  M[6], M[7], N[6], N[7]);
    // Step 5
    word const g4 = G(g0^g3,             T[0],  M[8], M[9], N[8], N[9]);
    // Step 6
    word const g5 = G(g1^g4,             T[1], M[10],M[11],N[10],N[11]);
    word const g11= G(g4,                T[3], M[22],M[23],N[22],N[23]);
    // Step 7
    word const g6 = G(g2^g5,             T[2], M[12],M[13],N[12],N[13]);
    word const g9 = G(g5,                T[1], M[18],M[19],N[18],N[19]);
    // Step 8
    word const g7 = G(g3^g6,             T[3], M[14],M[15],N[14],N[15]);
    word const g10= G(g6,                T[2], M[20],M[21],N[20],N[21]);
    word const g13= G(g6^g9,             T[1], M[26],M[27],N[26],N[27]);
    // Step 9
    word const g8 = G(g4^g7,             T[0], M[16],M[17],N[16],N[17]);
    word const g14= G(g4^g10,            T[2], M[28],M[29],N[28],N[29]);
    // Step 10
    word const g12= G(g5^g8,             T[0], M[24],M[25],N[24],N[25]);
    word const g15= G(g5^g8^g11,         T[3], M[30],M[31],N[30],N[31]);
    // Step 11
    word const g16= G(g6^g9^g12,         T[0], M[32],M[33],N[32],N[33]);
    // Step 12
    word const g17= G(g4^g10^g13^g16,    T[1], M[34],M[35],N[34],N[35]);
    // Step 13
    word const g18= G(g5^g8^g11^g14^g17, T[2], M[36],M[37],N[36],N[37]);
    // Step 14
    word const g19= G(g15^g18,           T[3], M[38],M[39],N[38],N[39]);
    // Step 15
    word const g20= G(g16^g19,           T[0], M[40],M[41],N[40],N[41]);
    // Step 16
    word const g21= G(g17^g20,           T[1], M[42],M[43],N[42],N[43]);
    word const g27= G(g20,               T[3], M[54],M[55],N[54],N[55]);
    // Step 17
    word const g22= G(g18^g21,           T[2], M[44],M[45],N[44],N[45]);
    word const g25= G(g21,               T[1], M[50],M[51],N[50],N[51]);
    // Step 18
    word const g23= G(g19^g22,           T[3], M[46],M[47],N[46],N[47]);
    word const g26= G(g22,               T[2], M[52],M[53],N[52],N[53]);
```

```
    word const g29= G(g22^g25,           T[1], M[58],M[59],N[58],N[59]);
    // Step 19
    word const g24= G(g20^g23,           T[0], M[48],M[49],N[48],N[49]);
    word const g30= G(g20^g26,           T[2], M[60],M[61],N[60],N[61]);
    Y[1]   = g20^g26^g29;
    // Step 20
    word const g28= G(g21^g24,           T[0], M[56],M[57],N[56],N[57]);
    word const g31= G(g21^g24^g27,       T[3], M[62],M[63],N[62],N[63]);
    Y[2]   = g21^g24^g27^g30;
    // Step 21
    Y[0]   = g22^g25^g28;
    Y[3]   = g31;
}
//---------------------------------------------------------------------
// Multiplicative inverse of x (mod 2**32), x odd.
// Source code by Thomas Pornin, Usenet 2009.
word inverse(word x)
{
    word y = 2 - x; // xy == 1 mod 4
    y *= 2 - x*y;   // xy == 1 mod 16
    y *= 2 - x*y;   // xy == 1 mod 256
    y *= 2 - x*y;   // xy == 1 mod 65536
    y *= 2 - x*y;   // xy == 1 mod 4294967296
    return y;
}
//---------------------------------------------------------------------
void invertkey( word      iM[64], word      iN[64],
                word const M[64], word const N[64] )
{
    // M, N, iM, iN must not overlap!
    for(int k=0; k < 64; k++)
    {
        iM[k] = inverse( M[63-k] );
        iN[k] = - N[63-k] * iM[k];
    }
}
//---------------------------------------------------------------------
void icrypt( word X[4], word const Y[4], word const T[4],
             word const iM[64], word const iN[64] )
{
    word Xrs[4], Trs[4];
    Trs[0] = _rotl(T[3],16);
    Trs[1] = _rotl(T[2],16);
    Trs[2] = _rotl(T[1],16);
    Trs[3] = _rotl(T[0],16);
    Xrs[0] = _rotl(Y[3],16);
    Xrs[1] = _rotl(Y[2],16);
    Xrs[2] = _rotl(Y[1],16);
    Xrs[3] = _rotl(Y[0],16);
    crypt( Xrs, Xrs, Trs, iM, iN );
    X[0] = _rotl(Xrs[3],16);
    X[1] = _rotl(Xrs[2],16);
    X[2] = _rotl(Xrs[1],16);
    X[3] = _rotl(Xrs[0],16);
}
//---------------------------------------------------------------------
// Testing against the reference implementation
void test()
{
    int const nTimes = 10000;
    int const nRep   = 100;
    word X[4], Y[4], T[4], Z[5], M[64], N[64], iM[64], iN[64];
    for(int i=0; i<5; i++)
        Z[i] = random_word();
    for(int i=0; i<4; i++)
        T[i] = random_word();
    word U = random_word();
    // correctness of the optimized implementation
    expandkey( M, N, Z, U );
    for(int n=nTimes; n; n--)
```

```
    {
        for(int i=0; i<4; i++)
            X[i] = random_word();
        memcpy( Y, X, sizeof(X) );
        for(int m=nRep; m; m--)
        {
            encrypt( X,  X,  Z, T, U );
            crypt( Y, Y, T, M, N );
        }
        if( memcmp(Y,X,sizeof(X)) !=0 )
            cout << "crypt: incorrect encryption!" << endl;
    }
    // invertibility of the optimized implementation
    invertkey( iM, iN, M, N );
    for(int n=nTimes; n; n--)
    {
        for(int i=0; i<4; i++)
            X[i] = random_word();
        memcpy( Y, X, sizeof(X) );
        for(int m=nRep; m; m--)
            crypt( Y, Y, T, M, N );
        for(int m=nRep; m; m--)
            icrypt( Y, Y, T, iM,iN );
        if( memcmp(Y, X, sizeof(X)) !=0 )
            cout << "icrypt: incorrect decryption!" << endl;
    }
}
///////////////////////////////////////////////////////////////////////
```