The WG Stream Cipher

Yassir Nawaz and Guang Gong

Department of Electrical and Computer Engineering
University of Waterloo
Waterloo, ON, N2L 3G1, CANADA
ynawaz@engmail.uwaterloo.ca, G.Gong@ece.uwaterloo.ca

Abstract. In this paper we propose a new synchronous stream cipher, called WG cipher. The cipher is based on WG (Welch-Gong) transformations. The WG cipher has been designed to produce keystream with guaranteed randomness properties, i.e., balance, long period, large and exact linear complexity, 3-level additive autocorrelation, and ideal 2-level multiplicative autocorrelation. It is resistant to Time/Memory/Data tradeoff attacks, algebraic attacks and correlation attacks. The cipher can be implemented with a small amount of hardware.

1 Introduction

A synchronous stream cipher consists of a keystream generator which produces a sequence of binary digits. This sequence is called the *running key* or simply the keystream. The keystream is added (XORed) to the plaintext digits to produce the ciphertext. A secret key K is used to initialize the keystream generator and each secret key corresponds to a generator output sequence. Since the secret key is shared between the sender and the receiver, an identical keystream can be generated at the receiving end. The addition of this keystream with the ciphertext recovers the original plaintext.

Stream ciphers can be divided into two major categories: bit-oriented stream ciphers and word-oriented stream ciphers. The bit-oriented stream ciphers are usually based on binary linear feedback shift registors (LFSRs) (regularly clocked or irregularly clocked) together with filter or combiner functions. They can be implemented in hardware very efficiently. However due to their bit-oriented nature their software implementations are very slow. To address this drawback recently many word oriented stream ciphers have been proposed [1–7]. Most of these stream ciphers are also based on LFSRs but they operate on words instead of bits. This results in a very high throughput in software. While these word oriented stream ciphers are fast in software and can also provide high security, they usually do not provide guaranteed keystream properties such as exact measure of linear complexity and ideal two-level autocorrelation etc.. These properties might be desirable in certain communications applications.

The desirable keystream properties are provided in bit-oriented cipher design by well developed theory of sequence design and boolean functions. Many bit oriented stream ciphers such as A5 [10], E_0 [9], and LILI-128 [8] exist that are fast and can be implemented with a small amount of hardware. However they all suffer from various cryptanalytic attacks and do not provide properties such as exact period, linear

complexity and good statistical properties, such as ideal two-level (multiplicative) autocorrelation.

In this paper we propose a new stream cipher, WG, which generates a keystream with guaranteed keystream properties and also offers high level of security. The cipher is based on WG (Welch-Gong) transformations which have proven cryptographic properties [11]. The cipher has been designed to produce a keystream which has all the cryptographic properties of WG transformation sequences, is resistant to Time/Memory/Data tradeoff attacks, algebraic attacks and correlation attacks, and can also be implemented in hardware efficiently.

The rest of the paper is organized as follows. In Sections 2 and 3 we describe the WG cipher. In Section 4 we describe the key initialization process and operation of the cipher. The security of the cipher including the randomness properties of the keystream are discussed in Section 5. In Section 6 we give the rationale behind the chosen design parameters, and the implementation aspects are discussed in Section 7.

2 Preliminaries

In this section we define and explain the terms and notations that will be used in this document to describe the WG cipher and its operation.

- $F_2 = GF(2)$, finite field with 2 elements: 0 and 1.
- $F_{2^{29}} = GF(2^{29})$, extension field of $GF(2)$ with 2^{29} elements. Each element in this field is represented as a 29 bit binary vector.
- $Tr(x) = x + x^2 + x^{2^2} + \cdots + x^{2^{28}}$, the trace function from $F_{2^{29}} \rightarrow F_2$
  $Tr_{29}^{11\times 29}(x) = x + x^{2^{29}} + \cdot\cdot + x^{2^{10\times 29}}$, the trace function from $F_{2^{11\times 29}} \rightarrow F_{2^{29}}$.
- Polynomial basis of $F_{2^{29}}$: Let α be the root of the primitive polynomial that generates $F_{2^{29}}$. Then $\{1, \alpha, \alpha^2, \cdots, \alpha^{28}\}$ is the polynomial basis of $F_{2^{29}}$ over F_2.
- Normal basis of $F_{2^{29}}$: Let γ be an element of $F_{2^{29}}$ such that $\{\gamma, \gamma^{2^1}, \gamma^{2^2}, \cdots, \gamma^{2^{28}}\}$ is a basis of $F_{2^{29}}$ over F_2. Then $\{\gamma, \gamma^{2^1}, \gamma^{2^2}, \cdots, \gamma^{2^{28}}\}$ is a normal basis of $F_{2^{29}}$ over F_2.

3 WG Cipher

In this section we provide a detailed description of the WG cipher. The WG cipher can be used with keys of length 80, 96, 112 and 128 bits. An initial vector (IV) of size 32 or 64 bits can be used with any of the above key lengths. To increase security, IVs of the same length as the secret key can also be used. WG cipher is a synchronous stream cipher which consists of a WG keystream generator. A simple block diagram of the WG keystream generator is shown in Figure 1. The keystream produced by the generator is added bitwise to the plaintext to produce the ciphertext. We now describe the WG keystream generator. As shown in figure 1 the keystream generator consists of a 11 stage linear feedback shift register(LFSR) over $F_{2^{29}}$. The feedback polynomial of the LFSR is primitive over $F_{2^{29}}$ and produces a maximal length sequence (m-sequence) over $F_{2^{29}}$. This m-sequence is filtered by

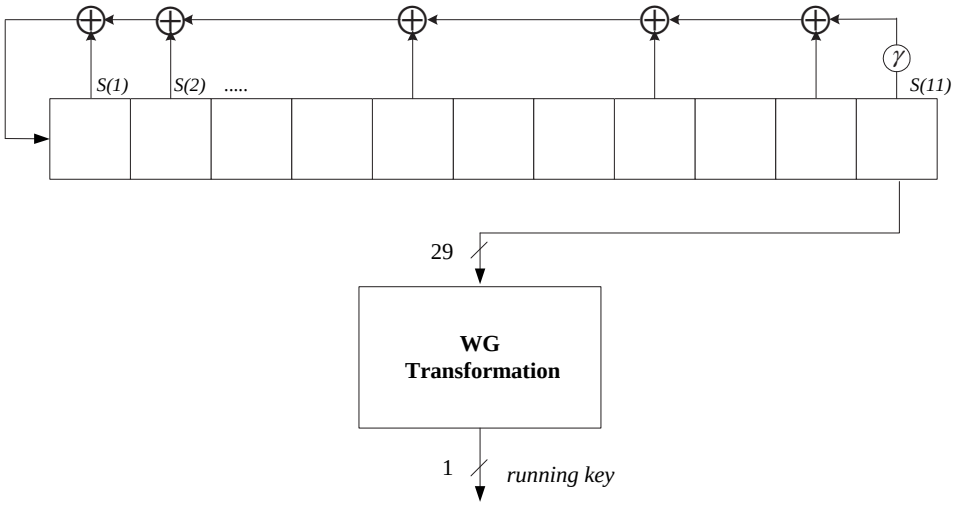

Fig. 1. Block diagram of WG Keystream generator

a nonlinear WG transformation, $F_{2^{29}} \rightarrow F_2$, to produce the keystream. All the elements of $F_{2^{29}}$ are represented in normal basis and all the finite field computations are in normal basis as well. The feedback polynomial of the LFSR is given by

$$p(x) = x^{11} + x^{10} + x^9 + x^6 + x^3 + x + \gamma \tag{1}$$

where $F_{2^{29}}$ is generated by the primitive polynomial

$$\begin{aligned} g(x) = x^{29} + x^{28} + x^{24} + x^{21} + x^{20} + x^{19} + x^{18} + x^{17} + \\ x^{14} + x^{12} + x^{11} + x^{10} + x^7 + x^6 + x^4 + x + 1. \end{aligned} \tag{2}$$

over F_2, and $\gamma = \beta^{464730077}$ where β is a root of $g(x)$. We define $S(1), S(2), S(3), .., S(11) \in F_{2^{29}}$ to be the state of the LFSR and the internal state of the WG cipher. We denote the output of the LFSR as $b_i = S(11-i), i = 0, 1, .., 10$. Then for $i \geq 11$, we have

$$b_i = b_{i-1} + b_{i-2} + b_{i-5} + b_{i-8} + b_{i-10} + \gamma b_{i-11}, i \geq 11 \tag{3}$$

where the above computations are in $F_{2^{29}}$. We now give a mathematical description of the WG transformation:

3.1 Mathematical Description of WG Transformation

Consider $F_{2^{29}}$ generated by the primitive polynomial $g(x)$ with β as its root. Let

$$t(x) = x + x^{2^{10}+1} + x^{2^{19}+2^9+1} + x^{2^{19}-2^9+1} + x^{2^{19}+2^{10}-1}, x \in F_{2^{29}}. \tag{4}$$

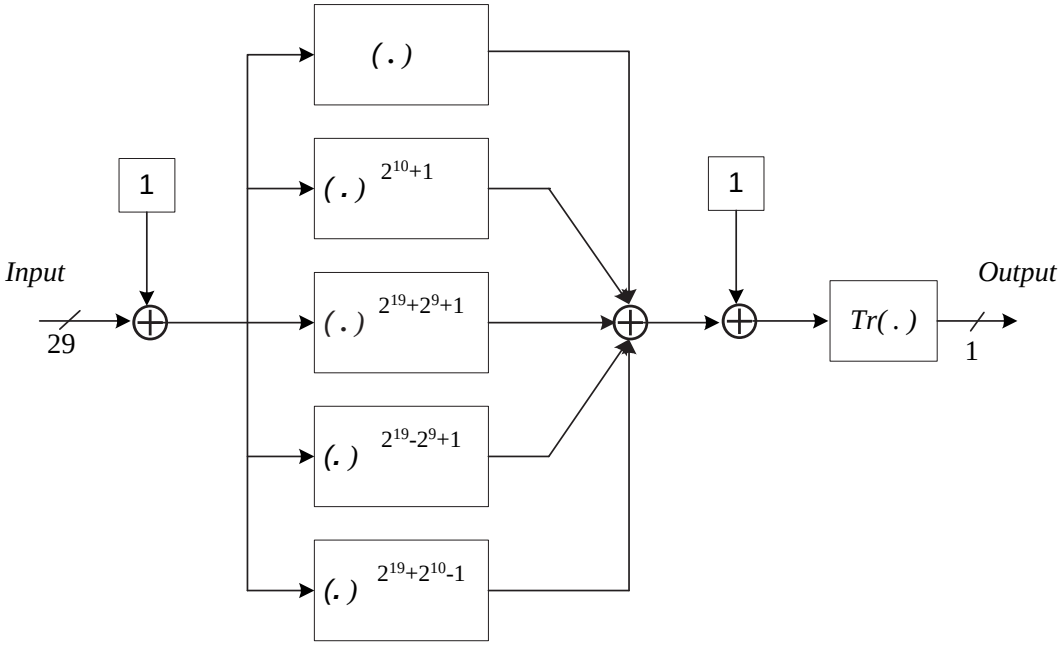

Fig. 2. Block diagram of WG transformation: $F_{2^{29}} \rightarrow F_2$

Then the function defined by

$$f(x) = \mathrm{Tr}(t(x+1)+1), x \in F_{2^{29}} \tag{5}$$

is the WG transfromation from $F_{2^{29}} \rightarrow F_2$. Figure 2 illustrates the above transformation. The 29 bit input to the WG tranformation function is regarded as an element of $F_{2^{29}}$ represented in normal basis. All the computations shown in Figure 1 and 2 (addition, exponentiation, multiplication and inversion) are normal basis computations. This normal basis of $F_{2^{29}}$, which is defined by $g(x)$, is generated by the element $\gamma \in F_{2^{29}}$ and γ is given above by $\gamma = \beta^{464730077}$. In polynomial form γ is represented as

$$\begin{aligned}\gamma = {} & \beta^1 + \beta^2 + \beta^3 + \beta^4 + \beta^5 + \beta^6 + \beta^7 + \beta^{10} + \beta^{11} + \beta^{12} + \\ & \beta^{13} + \beta^{14} + \beta^{15} + \beta^{16} + \beta^{17} + \beta^{20} + \beta^{23} + \beta^{24} + \beta^{26} + \beta^{27}.\end{aligned} \tag{6}$$

The normal basis can now be defined as $\{\gamma, \gamma^{2^0}, \gamma^{2^2}, \cdots, \gamma^{2^{28}}\}$.

We now state a few facts about computation in $F_{2^{29}}$ when field elements are represented in normal basis. We will use these facts to give a simple and more specific description of WG transformation in the next section.

– If the field elements are represented in a normal basis, exponentiation can be done by right cyclic shift, i.e., if $x \in F_{2^{29}}$ is represented as a 29 bit vector, then x^{2^i} can be obtained by simply shifting the bits of x cyclically to right by i steps.

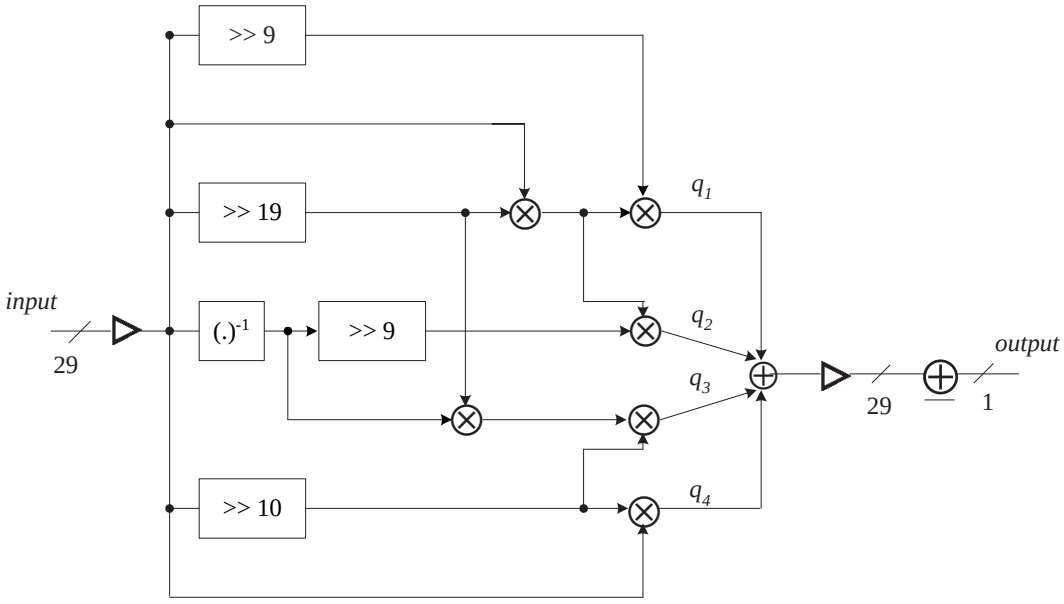

Fig. 3. Block diagram of the implementation of WG transformation: $F_{2^{29}} \rightarrow F_2$

- If γ is the generator of the normal basis then $1 = \sum_{i=0}^{28} \gamma^{2^i}$, i.e., the 29 bit vector representing 1 is all one vector. Therefore addition of a field element, in normal basis representation, with 1 can be done by simply inverting the bits of that element.
- The trace of all the basis elements is one, i.e., $\mathrm{Tr}(\gamma^{2^i}) = 1$ where $0 \leq i \leq 28$. Therefore the trace of any field element represented as a 29 bit vector can be obtained by adding all the bits of the elements over F_2 (i.e., XOR).

3.2 Block Diagram Description of WG Transformation

To facilitate the implementation of the WG transformation in hardware and software we provide a more specific description in Figure 3. From the figure the output of the WG transformation can be written as

$$output = \underline{\bigoplus}(\triangleright(q_1 \oplus (q_2 \oplus (q_3 \oplus q_4)))) \tag{7}$$

where

$$q_1 = (I \gg 9) \otimes ((I \gg 19) \otimes I)$$
$$q_2 = (I^{-1} \gg 9) \otimes ((I \gg 19) \otimes I)$$
$$q_3 = (I^{-1} \otimes (I \gg 19)) \otimes (I \gg 10)$$
$$q_4 = (I \gg 10) \otimes I$$
$$I = \triangleright(input).$$

The notation $x \otimes y$ means normal basis multiplication of x and y in $\mathrm{F}_{2^{29}}$ defined by $g(x)$. Similarly $(x)^{-1}$ means the normal basis inversion of x in $\mathrm{F}_{2^{29}}$ defined by $g(x)$. $x \oplus y$ represents the bitwise addition (XOR) of words x and y, and $x \gg c$ represents the cyclic shift of x, c stages to the right where c is a positive integer. The symbol $\triangleright(x)$ means all the 29 bits of x are complemented and $\underline{\bigoplus}(x)$ means the addition of the 29 bits of x over F_2 (XOR) i.e., for $x = (x_0, .., x_{28})$, $\underline{\bigoplus}(x) = \sum_{i=0}^{28} x_i \bmod 2$.

4 Key Initialization and Operation

In this section we describe the key initialization process and operation of the WG cipher. Note that according to the ECRYPT NoE requirements for the Proposals in profile 2, an Initial Vector (IV) length of 32 bits or 64 bits must be accommodated [12]. However due to the recent developments regarding Time/Memory/Data tradeoff attacks by Hong and Sarcar [13] and subsequent comments by Canniere, Lano and Preneel [14], it is highly recommended that IV's which have the same length as the secret key itself are used. Therefore we provide two key initialization mechanisms: One for 32-bit and 64-bit long IVs and the second for the IVs that have the same length as the secret key.

4.1 Key Initialization (32-bit and 64-bit IVs)

The recommended key sizes for WG cipher are 80, 96, 112 and 128 bits. An Initialization Vector (IV) of size 32 or 64 bits can be used with any of the above key sizes. To initialize the cipher the key bits and IV bits are loaded into the LFSR. Now we describe how to load the key and IV bits into the LFSR. The state of the LFSR is represented as $S(1), S(2), S(3), .., S(11) \in F_{29}$. We represent each stage $S(i) \in \mathrm{F}_{29}$, as $S_{1,..,29}(i)$, where $1 \leq i \leq 11$. Similarly we represent the key bits as $k_{1,..,j}, 1 \leq j \leq 128$ and IV bits as $IV_{1,..,m}, 1 \leq m \leq 64$.

The key bits are divided into blocks of 16 bits and each block is loaded into the LFSR as follows:

- 80 bits key is loaded as

$$
\begin{array}{lll}
S_{1,..16}(1) = k_{1,..,16} & S_{1,..16}(2) = k_{17,..,32} & S_{1,..16}(3) = k_{33,..,48} \\
S_{1,..16}(4) = k_{49,..,64} & S_{1,..16}(5) = k_{65,..,80} & S_{1,..,16}(9) = k_{1,..,16} \\
S_{1,..,16}(10) = k_{17,..32} \oplus 1 & S_{1,..,16}(11) = k_{33,..,48} &
\end{array}
$$

- 96 bits key is loaded as

$$
\begin{array}{lll}
S_{1,..16}(1) = k_{1,..,16} & S_{1,..16}(2) = k_{17,..,32} & S_{1,..16}(3) = k_{33,..,48} \\
S_{1,..16}(4) = k_{49,..,64} & S_{1,..16}(5) = k_{65,..,80} & S_{1,..16}(6) = k_{81,..,96} \\
S_{1,..,16}(9) = k_{1,..,16} & S_{1,..,16}(10) = k_{17,..32} \oplus 1 & S_{1,..,16}(11) = k_{33,..,48}
\end{array}
$$

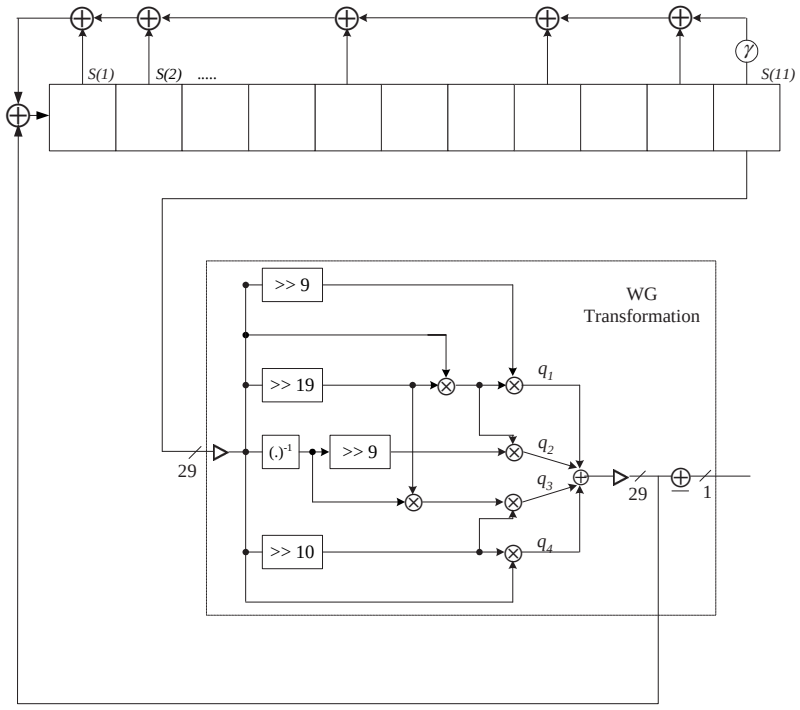

Fig. 4. Key initialization phase of WG cipher

– 112 bits key is loaded as

$S_{1,..16}(1) = k_{1,..,16}$ $S_{1,..16}(2) = k_{17,..,32}$ $S_{1,..16}(3) = k_{33,..,48}$
$S_{1,..16}(4) = k_{49,..,64}$ $S_{1,..16}(5) = k_{65,..,80}$ $S_{1,..16}(6) = k_{81,..,96}$
$S_{1,..16}(7) = k_{97,..,112}$ $S_{1,..,16}(9) = k_{1,..,16}$ $S_{1,..,16}(10) = k_{17,..32} \oplus 1$
$S_{1,..,16}(11) = k_{33,..,48}$

– 128 bits key is loaded as

$S_{1,..16}(1) = k_{1,..,16}$ $S_{1,..16}(2) = k_{17,..,32}$ $S_{1,..16}(3) = k_{33,..,48}$
$S_{1,..16}(4) = k_{49,..,64}$ $S_{1,..16}(5) = k_{65,..,80}$ $S_{1,..16}(6) = k_{81,..,96}$
$S_{1,..16}(7) = k_{97,..,112}$ $S_{1,..16}(8) = k_{113,..,128}$ $S_{1,..,16}(9) = k_{1,..,16}$
$S_{1,..,16}(10) = k_{17,..32} \oplus 1$ $S_{1,..,16}(11) = k_{33,..,48}$

The IV bits are divided into blocks of 8 bits and each block is loaded into the LFSR as follows:

– 32 bits IV is loaded as

$S_{17,..24}(1) = IV_{1,..,8}$ $S_{17,..24}(2) = IV_{9,..,16}$ $S_{17,..24}(3) = IV_{17,..,24}$
$S_{17,..24}(4) = IV_{25,..,32}$

– 64 bits IV is loaded as

$S_{17,..24}(1) = IV_{1,..,8}$ $S_{17,..24}(2) = IV_{9,..,16}$ $S_{17,..24}(3) = IV_{17,..,24}$
$S_{17,..24}(4) = IV_{25,..,32}$ $S_{17,..24}(5) = IV_{33,..,40}$ $S_{17,..24}(6) = IV_{41,..,48}$
$S_{17,..24}(7) = IV_{49,..,56}$ $S_{17,..24}(8) = IV_{57,..,64}$

All the remaining bits of the LFSR are set to zero. Once LFSR has been loaded with the key and IV, the keystream generator is run for 22 clock cycles. This is the key initialization phase of the cipher operation. During this phase the 29 bit vector, given by

$$keyinitvec = \triangleright(q_1 \oplus (q_2 \oplus (q_3 \oplus q_4))) \tag{8}$$

in Figure 3, is added to the feedback of the LFSR which is then used to update the LFSR. The key initialization process is shown in Figure 4. Once the key has been initialized the LFSR is clocked once and the 1 bit output of the WG transformation gives the first bit of the running keystream. Since the linear complexity of the keystream is slightly more than 2^{45} (see Sec 5.1) the maximum length of the keystream allowed to be generated with a single key and IV is 2^{45}. After this the cipher must be reinitialized with a new IV or a new key or both.

4.2 Key Initialization (Same Length Keys and IVs)

We keep the notation introduced in Sec 4.1, i.e., $S(1), S(2), S(3), .., S(11) \in \mathrm{F}_{2^{29}}$; Each stage $S(i) \in \mathrm{F}_{2^{29}}$, is represented as $S_{1,..,29}(i)$ where $1 \leq i \leq 11$. Similarly we represent the key bits as $k_{1,..,j}, 1 \leq j \leq 128$ and IV bits as $IV_{1,..,m}, 1 \leq m \leq 128$.

– 80 bits key and IV are loaded as

$S_{1,..16}(1) = k_{1,..,16}$ $S_{17,..24}(1) = IV_{1,..,8}$ $S_{1,..8}(2) = k_{17,..,24}$
$S_{9,..24}(2) = IV_{9,..,24}$ $S_{1,..16}(3) = k_{25,..,40}$ $S_{17,..24}(3) = IV_{25,..,32}$
$S_{1,..8}(4) = k_{41,..,48}$ $S_{9,..24}(4) = IV_{33,..,48}$ $S_{1,..16}(5) = k_{49,..,64}$
$S_{17,..24}(5) = IV_{49,..,56}$ $S_{1,..8}(6) = k_{65,..,72}$ $S_{9,..24}(6) = IV_{57,..,72}$
$S_{1,..8}(7) = k_{73,..,80}$ $S_{17,..24}(7) = IV_{73,..,80}$

– 96 bits key and IV are loaded as

$S_{1,..16}(1) = k_{1,..,16}$ $S_{17,..24}(1) = IV_{1,..,8}$ $S_{1,..8}(2) = k_{17,..,24}$
$S_{9,..24}(2) = IV_{9,..,24}$ $S_{1,..16}(3) = k_{25,..,40}$ $S_{17,..24}(3) = IV_{25,..,32}$
$S_{1,..8}(4) = k_{41,..,48}$ $S_{9,..24}(4) = IV_{33,..,48}$ $S_{1,..16}(5) = k_{49,..,64}$
$S_{17,..24}(5) = IV_{49,..,56}$ $S_{1,..8}(6) = k_{65,..,72}$ $S_{9,..24}(6) = IV_{57,..,72}$
$S_{1,..16}(7) = k_{73,..,88}$ $S_{17,..24}(7) = IV_{73,..,80}$ $S_{1,..8}(8) = k_{89,..,96}$
$S_{9,..24}(8) = IV_{81,..,96}$

– 112 bits key and IV are loaded as

$S_{1,..16}(1) = k_{1,..,16}$ $S_{17,..24}(1) = IV_{1,..,8}$ $S_{1,..8}(2) = k_{17,..,24}$
$S_{9,..24}(2) = IV_{9,..,24}$ $S_{1,..16}(3) = k_{25,..,40}$ $S_{17,..24}(3) = IV_{25,..,32}$
$S_{1,..8}(4) = k_{41,..,48}$ $S_{9,..24}(4) = IV_{33,..,48}$ $S_{1,..16}(5) = k_{49,..,64}$
$S_{17,..24}(5) = IV_{49,..,56}$ $S_{1,..8}(6) = k_{65,..,72}$ $S_{9,..24}(6) = IV_{57,..,72}$
$S_{1,..16}(7) = k_{73,..,88}$ $S_{17,..24}(7) = IV_{73,..,80}$ $S_{1,..8}(8) = k_{89,..,96}$
$S_{9,..24}(8) = IV_{81,..,96}$ $S_{1,..16}(9) = k_{97,..,112}$ $S_{17,..24}(9) = IV_{97,..,104}$
$S_{9,..16}(10) = IV_{105,..,112}$

– 128 bits key and IV are loaded as

$S_{1,..16}(1) = k_{1,..,16}$ $S_{17,..24}(1) = IV_{1,..,8}$ $S_{1,..8}(2) = k_{17,..,24}$
$S_{9,..24}(2) = IV_{9,..,24}$ $S_{1,..16}(3) = k_{25,..,40}$ $S_{17,..24}(3) = IV_{25,..,32}$
$S_{1,..8}(4) = k_{41,..,48}$ $S_{9,..24}(4) = IV_{33,..,48}$ $S_{1,..16}(5) = k_{49,..,64}$
$S_{17,..24}(5) = IV_{49,..,56}$ $S_{1,..8}(6) = k_{65,..,72}$ $S_{9,..24}(6) = IV_{57,..,72}$
$S_{1,..16}(7) = k_{73,..,88}$ $S_{17,..24}(7) = IV_{73,..,80}$ $S_{1,..8}(8) = k_{89,..,96}$
$S_{9,..24}(8) = IV_{81,..,96}$ $S_{1,..16}(9) = k_{97,..,112}$ $S_{17,..24}(9) = IV_{97,..,104}$
$S_{1,..8}(10) = k_{113,..,120}$ $S_{9,..24}(10) = IV_{105,..,120}$ $S_{1,..8}(11) = k_{121,..,128}$
$S_{17,..24}(11) = IV_{121,..,128}$

The remaining bits in the LFSR are all set to zero. Once the key and the IV have been loaded, the keystream generator is run for 22 clock cycles. This initialization process is identical to the one described in Section 4.1.

4.3 Operation of the Cipher

Once the cipher has been initialized, the contents of the LFSR constitute the internal state of the cipher. To produce the running keystream LFSR is clocked once and the contents of the stage $S(11)$ are fed into the WG transformation block which produces a single bit of the runnig keystream. The LFSR is clocked again and the updated contents of $S(11)$ are again fed to WG transformation block thus producing the next bit of the keystream and so on. This running keystream is added bitwise (XORed) to the plaintext to produce the ciphertext.

5 Security of the Cipher

5.1 Randomness Properties of the Keystream

The WG keystream $\{a_i\}$is generated by filtering a maximal length sequence over $\mathrm{F}_{2^{29}}$ by a WG transformation. Therefore we can represent the output of the generator as

$$a_i = f(b_i), i = 0, 1, \cdots$$
$$u(x) = f \circ \mathrm{Tr}_{29}^{319}(\mathrm{x}), \tag{9}$$

i.e., $u(x)$ is the composition of $\mathrm{Tr}_{29}^{319}(x)$ and f where $\{b_i\}$ is the m-sequence generated by the LFSR over $\mathrm{F}_{2^{29}}$ with the trace representation $\mathrm{Tr}_{29}^{319}(x)$ and f is the

WG transformation. The sequence corresponding to $u(x)$ is also referred to as the generalized GMW sequence in the literature [15]. We will use the properties of f and (9) to derive the properties of the keystream generated by the WG keystream generator.

- **Period**: The WG keystream generator has an 11 stage LFSR over $F_{2^{29}}$ with a primitive feedback polynomial which generates a maximal length sequence of period $2^{11\times 29}-1$ over $F_{2^{29}}$. Therefore the period of the keystream generated by the cipher is $2^{319}-1$.
- **Balance**: Since the m-sequence $\{b_i\}$ over $F_{2^{29}}$ is balanced and the WG is a balanced boolean function from $F_{2^{29}} \rightarrow F_2$, the keystream is also balanced. There are 2^{318} 1's and $2^{318}-1$ 0's in one period of the keystream.
- **Two-level autocorrelation**: The WG transformation is an orthogonal function and the corresponding WG transformation sequence has 2-level autocorrelation [11]. We now consider (9): It is proved in [15] that if f is an orthogonal function then the sequence that corresponds to u has 2-level autocorrelation. Therefore the keystream generated by the WG keystream generator has 2-level autocorrelation.
- **t-tuple distribution**: Since $\{b_i\}$ is an m-sequence over $F_{2^{29}}$ of degree 11 and f is a balanced boolean function from $F_{2^{29}} \rightarrow F_2$, then according to [15] the keystream $\{a_i\}$ has ideal t-tuple distribution where $1 \leq t \leq 11$.
- **Linear Complexity**: Due to (9) the linear complexity of the keystream can be determined exactly and is given by the following formula:

$$LS = 29 \times \sum_{i \in I} 11^{w(i)} \approx 2^{45.0415} \tag{10}$$

where $w(i)$ is the hamming weight of i and

$$I = I_1 \cup I_2,$$

where

$$I_1 = \{2^{19} + 2^9 + 2 + i \mid 0 \leq i \leq 2^9 - 3\},$$

$$I_2 = \{2^{20} + 3 + 2i \mid 0 \leq i \leq 2^9 - 2\}.$$

For more details on the calculation of linear complexity the reader is referred to [15].

5.2 WG Transformation

The WG transformation form $F_{2^{29}} \rightarrow F_2$ can be regarded as a boolean function in 29 variables. The exact boolean representation depends on the basis used for computation in $F_{2^{29}}$. The normal basis provided in previous selections has been selected in such a way so that the corresponding boolean representation of WG transformation is 1-order resilient. It has degree 11 and its nonlinearity is $2^{28}-2^{14} = 268419072$. For more detail on the selection of resilient bases see [11].

5.3 Security Against Attacks

In this section we analyze the security of the WG cipher against some well known attacks on stream ciphers. The types of attacks considered are Time/Memory/Data tradeoff attacks, algebraic attacks and correlation attacks.

Time/Memory/Data tradeoff attacks: Let's first consider the Time/Memory/Data tradeoff attack on stream ciphers. This kind of attack has two phases: During pre-computation phase the attacker exploits the structure of the stream cipher and summarizes his findings in large tables. During the attack phase, the attacker uses these tables and the observed data to determine the secret key or the internal state of the stream cipher. The size of the tables in the pre-computation stage, the required keystream, and the computational effort required to recover the secret key determine the feasibility of this attack. A tradeoff $TM^2D^2 = N^2$ for $D^2 \leq T \leq N$, was presented in [16] where T is the time required for the attack, M is the memory required to store the tables, D represents the realtime data or the keystream required, and N is the size of the search space. A simple way to provide security against this attack in stream ciphers is to increase the search space. This can be done by increasing the size of the internal state and using random IVs along with the secret key. In WG stream cipher the size of the inner state in 2^{319} which is more than twice the size of the largest possible keysize. If a random IV of the same length as the secret key is selected, the cipher is secure against Time/Memory/Data tradeoff attacks.

Algebraic attacks: Now we consider algebraic attacks that have been used recently to break many well known stream ciphers [17–20]. Courtois in [18] has shown that the complexity of these attack depends on the nonlinear filter and the number of outputs generated by the cipher. If the nonlinear filter can be approximated by a multivariate equation of low degree this complexity can be reduced significantly. The boolean function representation of WG has 29 inputs, one output and has degree 11. According to [18] a nonlinear filter with 29 inputs and 1 output must have a multivariate approximation of degree 14. However this degree is greater than 11, the degree of the WG transformation. The meaningful approximations to the WG transformation should have degree less than 11. Assuming that there are no approximations of WG with degree less than 11, the cipher can be reduced to a system of approximately $\binom{319}{11}$ linear equations. The complexity of solving such a system is approximately $7/64 \cdot \binom{319}{11}^{log_2 7} \approx 2^{182}$. If an approximation of WG with degree less than 11 is found the complexity of the attack will be reduced. According to the assertion in [19] and our experimental results on WG transformation we conjecture that the probability of the existence of such an approximation is negligible. We have verified that WG transformations in 11, 13 and 14 variables(which have degrees 5, 6 and 6 respectively) do not have approximations with degrees less than the degree of the WG transformations themselves. Since both the boolean and polynomial representations of WG transformation have large number of monomial terms with high degrees it is not possible to remove higher degree terms without significantly affecting the output of the transformation. The property that WG transformation

has a large number of monomial terms in its polynomial and boolean representation provides adequate security against algebraic attacks.

Correlation attacks: Another powerful attack against stream ciphers is the correlation attack. Several correlation attacks have been proposed in the literature [21–23]. These attacks exploit any correlation that may exist between the keystream and the output of the LFSR in the cipher. In these attacks the keystream is regarded as a distorted or noisy version of the the LFSR output. This reduces the problem of finding the internal state of the LFSR to a decoding problem where keystream is the received codeword and LFSR internal state, the original message. The WG transformation used in WG cipher is 1-order resilient, i.e., the output of the WG transformation or the keystream is not correlated to any single input bit of the LFSR output.This suggests that WG cipher is secure against correlation attacks. However we must consider the case where WG transformation is approximated by linear functions. These linear approximations can be used to derive a generator matrix of a linear code. The decoding can then be performed by a Maximum Likelihood (ML) decoding algorithm to recover the internal state of LFSR. We now use some of the theoretical bounds given in [23] to estimate the complexity of these attacks on the WG cipher. Let f' be the boolean function representation of WG transformation and l be a linear function with minimum hamming distance to f'. Then

$$P(f'(x) = l(x)) = \frac{2^{29} - (2^{28} - 2^{14})}{2^{29}} = 0.5000305. \tag{11}$$

Using the results given in [23] with parameter $t = 3$, the amount of keystream required for a successful attack is given by

$$N \approx (k \cdot 12 \cdot ln2)^{1/3} \cdot \epsilon^{-2} \cdot 2^{\frac{319-k}{3}} \tag{12}$$

and decoding complexity is given by

$$C_{dec} = 2^k \cdot k \cdot \frac{2ln2}{(2\epsilon)^6}, \tag{13}$$

where $\epsilon = P(f'(x) = l(x)) - 0.5 = 0.0000305$ and k is the number of LFSR internal state bits recovered. If we choose k to be very small, i.e., $k = 5$, the amount of keystream required for the attack is approximately 2^{133}, which is not a realistic amount to collect. Moreover the complexity of pre-computation phase is more than 2^{266}. Since the maximum keystream that can be produced with a single key and IV is 2^{45} we choose $k = 274$ to reduce N to this number. Now the complexity of the decoding phase is approximately 2^{366}. This analysis shows that the WG cipher is secure against this kind of correlation attacks.

6 Design Rationale

The WG keystream generator has been designed as an efficient and secure stream cipher with desired keystream properties i.e., balance, long period, large linear complexity, ideal t-tuple distribution $1 \leq t \leq 11$, 3-level additive autocorrelation [35]

and 2 level (multiplicative) autocorrelation. The cipher is intended for hardware applications however it can be used in software applications if the keystream with the above statistical properties is desired. To obtain the desired keystream properties, WG transformation has been chosen as the nonlinear filter since it has all the desired cryptographic properties. The WG transformation sequences have large linear complexities and ideal 2-level (multiplicative) autocorrelation. The size of the WG transforamation from $F_{2^{29}} \rightarrow F_2$ has been chosen to facilitate a practical hardware implementation.

The WG transformation $F_{2^n} \rightarrow F_2$ only exist for $n = 3k - 1$ and $n = 3k - 2$, where k is an integer. It involves finite field operations (exponentiation and inversion) over the extension field F_{2^n}. The exponentiation is an expensive operation if the elements of the field are represented in polynomial basis. However if the elements are represented in normal basis the exponentiations in WG tranformation can be achieved by simple cyclic shifts and a few multiplications. Since raising an element to the power of 2 is free and inversion can be implemented with multiplications, the complexity of the transformation depends mostly on the complexity of the normal basis multiplier. There are more than one normal basis in an extension filed and the complexity of a normal basis multiplier depends on the choice of the normal basis. The normal basis for which this complexity is minimum is known as optimal normal basis [24]. However optimal normal basis does not exist for every extension field F_{2^n}. For $F_{2^{29}}$ optimal normal basis as well as the WG tranformation exist. Therefore $F_{2^{29}}$ has been chosen to facilitate an efficient hardware implementation of WG tranformation. Moreover $F_{2^{29}}$ does not have any subfields and this elliminates the possibility of a subfield attack. The elements of $F_{2^{29}}$ can also be represented with a single word in 32 bit processors which leads to an efficient software implementation.

The LFSR has been chosen to generate the maximal length sequence over $F_{2^{29}}$ and the length of the LFSR has been chosen to obtain high linear complexity of the keystream and a large internal state of the cipher.

7 Implementation of WG Cipher

In this section we discuss the implementation aspects of WG cipher in hardware and software.

7.1 Hardware Implementation

The inversion and normal basis multiplication in $F_{2^{29}}$ are the two most expensive operations in the hardware implementation of the cipher. The WG transformation requires one inversion and six multiplications for each output. Depending on the type and number of multipliers used, a wide variety of area versus speed tradeoffs are possible. An implementation can range from using a single serial normal basis multiplier to multiple parallel normal basis multipliers. While the underlying platform and speed requirements will dictate an implementation, we suggest an obvious implementation which can achieve high speed with relatively less amount of hardware. The best normal basis inversion algorithm known to the authors requires 6

clock cycles to compute the inverse in $F_{2^{29}}$, provided a parallel normal basis multiplier is available [25]. A parallel normal basis multiplier multiplies two elements in normal basis and produces the result in one clock cycle. Therefore we suggest using 2 parallel normal basis multipliers. In Figure 3, one multiplier can be used to compute the inverse while the second can be used to compute q_1 and q_2. After the inverse has been calculated the remaining 3 multiplications can be done in 2 clock cycles by the 2 multipliers. Therefore it will take a total of 8 clock cycles to produce a keystream bit. Note that the cyclic shifts can be done by rearranging the connections to the registers or the inputs of multipliers.

Several normal basis multipliers have been proposed in the literature [26–32]. As stated earlier the hardware complexity of a normal basis multiplier depends on the basis used to represent the field elements. More precisely it depends on the multiplication matrix C_N of the chosen normal basis [24]. The complexity is directly proportional to the number of ones in the C_N matrix. To facilitate the hardware implementation of a normal basis multiplier for WG transformation we provide the C_N matrix that corresponds to the chosen optimal normal basis of $F_{2^{29}}$ in Appendix A.

To estimate the throughput that WG cipher can achieve we consider the delays of various operations in the cipher implementation. The operation with the maximum propagation delay is the normal basis multiplication. Most of the parallel normal basis multipliers in literature have propagation delay given by

$$delay = T_A + \lceil log_2 C_N \rceil T_X \tag{14}$$

where T_A is the delay of an AND gate, T_X is the delay of an XOR gate and C_N is number of ones in the multiplication matrix of the chosen normal basis. In case of optimal normal basis in F_{2^n}, $C_N = 2 \times n$. This means that for a parallel normal basis multiplier over $F_{2^{29}}$ the propagation delay is equal to 7 gate delays. The maximum clock frequency that can be used depends on the ASIC technology and the maximum propagation delay. For example with current ASIC technology clock frequency of 1 GHz can be used when the propagation delay equals 7 gate delays. This will result in the overall cipher throughput of 125 Mbps. The hardware complexity of the cipher depends on the type of multiplier chosen for implementation. For example a basic parallel normal basis multiplier given in [26] over $F_{2^{29}}$ requires 841 AND gates and 1653 XOR gates. Other implementations can provide even better hardware complexities.

Note that LFSR feedback has a normal basis multiplication with a constant γ. This can be done by either dedicated hardware in the feedback loop or by using one of the existing normal basis multiplier in WG transformation.

7.2 Software Implementation

The software implementation of the WG cipher is straight forward. Since the elements of $F_{2^{29}}$ can be represented within a single word of a 32 bit processor, each of the addition, negation and shift operations can be performed by a single instruction. The time consuming operations in software are the normal basis multiplications.

The exact implementation of the multiplier depends on the multiplication matrix. Several algorithms exist for performing normal basis multiplication in software efficiently [33, 34]. We provide a C implementation of the normal basis multiplication algorithm for the optimal normal basis over $F_{2^{29}}$ in Appendix D. In Appendix E we also provide the C implementation of the inversion algorithm given in [25] . The implementations are provided as examples and we do not claim them to be the most efficient. Any normal basis multiplication and inversion algorithm can be used for the software implementation of the cipher.

Since the cipher involves finite field operations over $F_{2^{29}}$, a software implementation for a general purpose processor is not very fast. Our simulations, which use an un-optimized C implementation on a 2.0 Ghz Pentium 4 PC using 512 Mbyte RAM result in a throughput of 0.22 Mbps.

8 Conclusions

We proposed a new stream cipher, WG cipher, suitable for hardware implementations. The cipher generates a keystream with guaranteed randomness properties and offers high level of security. The cipher can be implemented with relatively small amount of hardware while achieving speeds in excess of 100 Mbps. We believe that exhaustive key search is the most efficient way to recover the secret key or internal state of the cipher. We claim that we have not inserted any hidden weakness in the design of the WG cipher.

Acknowledgements: The authors would like to thank Dr. Kishan Gupta for his help in finding the primitive polynomials and Mr. NamYul Yu for generating WG sequences for algebraic testing.

References

1. R. Rivest, The RC4 Encryption Algorithm, *RSA Data Security, Inc.*, Mar. 1992.
2. P. Rogaway, and D. Coppersmith, A Software Optimized Encryption Algorithm, *Journal of Cryptology*, 11(4):273-287, 1998.
3. P. Hawkes, and G. Rose, Primitive Specification and Support Documentation for SOBER-t16 Submission to NESSIE, *Proceedings of First NESSIE Workshop*, Heverlee, Belgium, 2000.
4. P. Hawkes, and G. Rose, Primitive Specification and Support Documentation for SOBER-t32 Submission to NESSIE, *Proceedings of First NESSIE Workshop*, Heverlee, Belgium, 2000.
5. P. Ekdahl, and T. Johansson, SNOW-A New Stream Cipher, *Proceedings of First NESSIE Workshop*, Heverlee, Belgium, 2000.
6. P. Ekdahl, and T. Johansson, SNOW-A New Version of the Stream Cipher SNOW, *Selected Areas in Cryptography, 2002*, LNCS 2595, pp. 47-61, SPringer-Verlag 2003.
7. D. Watanabe, S. Furuya, H. Yoshida, and B. Preneel, A New Keystream Generator MUGI, *Fast Software Encryption 2002*, LNCS 2365, pp. 179-194, Springer-Verlag, 2002.
8. E. Dawson, A. Clark, J. Golic, W. Millan, L. Penna, and L. Simpson, The LILI-128 Keystream Generator, *Proceedings of First NESSIE Workshop*, Heverlee, Belgium, 2000.
9. Bluetooth Specification, version 1.1, Available at www.bluetooth.org/spec/.
10. M. Briceno, I. Goldberg, and D. Wagner, A Pedagogical Implementation of A5/1, http://www.scard.org, May 1999.
11. G. Gong, and A. Youssef, Cryptographic Properties of the Welch-Gong Transformation Sequence Generators, *IEEE Transactions on Information Theory*, vol. 48, No. 11, pp. 2837-2846, Nov. 2002.
12. ECRYPT: Call for Stream Cipher Primitives, https://www.cosic.esat.kuleuven.ac.be/ecrypt/stream/.
13. J. Hong, and P. Sarkar, Rediscovery of Time Memory Tradeoffs, *Cryptology ePrint Archive, Report 2005/090*, http://eprint.iacr.org/, 2005.
14. C. Canniere, J. Lano, and B. Preneel, Comments on the Rediscovery of Time Memory Data Tradeoffs, https://www.cosic.esat.kuleuven.ac.be/ecrypt/stream/TMD.pdf.
15. S. W. Golomb, and G. Gong, Signal Design for Good Correlation: For Wireless Communication, Cryptography, and Radar, *Cambridge University Press*, ISBN 0521821045, 2005.
16. A. Biryukov, and A. Shamir, Cryptanalytic Time/Memory/Data Tradeoffs for Stream Ciphers, *Asiacrypt 2000*, LNCS 1976, pp. 113, Springer-Verlag, 2000.
17. N. Courtois, Fast Algebraic Attacks on Stream Ciphers with Linear Feedback, *Advances in Cryptology-CRYPTO 2003*, LNCS 2729, pp. 176-194, Springer-Verlag, 2003.
18. N. Courtois, Algebraic Attacks on Combiners with Memory and Several Outputs, *Cryptology ePrint Archive, Report 2003/125*, http://eprint.iacr.org/, 2003.
19. W. Meier, E. Pasalic, and C. Carlet, Algebraic Attacks and Decomposition of Boolean Functions, *Advances in Cryptology EUROCRYPT-2004*, LNCS 3027, pp.474-491, Springer-Verlag, 2004.
20. Frederik Armknecht, On the Existence of Low-degree Equations for Algebraic Attacks, *Cryptology ePrint Archive, Report 2004/185*, http://eprint.iacr.org/, 2004.
21. T. Siegenthaler, Decrypting a Class of Stream Ciphers Using Ciphertext Only, *IEEE Transactions on Computers*, vol. C-34, pp. 81-85, 1985.
22. W. Meier, and O. Staffelbach, Fast Correlation Attacks on Certain Stream Ciphers, *Journal of Cryptology*, pp. 159-176, 1989.
23. V. Chepyzhov, T. Johansson, and B. Smeets, A Simple Algorithm for Fast Correlation Attacks on Stream Ciphers, *Fast Software Encryption 2000*, LNCS 1978, pp. 181-195, Springer-Verlag, 2001.
24. R. Mullin, I. Onyszchuk, and S. Vanstone, Optimal Normal Bases in GF(p^n), *Discrete Applied Mathematics*, vol. 22, pp. 149-161, 1989.
25. Gui-Liang Feng, A VLSI Architecture for Fast Inversion in GF(2^m), *IEEE Transactions on Computer*, vol. 38, No. 10, pp. 1383-1386, October 1989.
26. L. Massey, and J. Omura, Computational Method and Apparatus for Finite Field Arithmetic, *US Patent No. 4,587,627*, 1986.

27. C. Wang, T. Troung, H. Shao, L. Deutsch, J. Omura, and I. Reed, VLSI Architecture for Computing Multiplications and Inverses in GF(2^m), *IEEE Transactions on Computers*, vol. 34, No. 8, pp. 709-716, Aug. 1985.
28. B. Sunar, and C. Koc, An Efficient Optimal Normal Basis Type II Multiplier, *IEEE Transactions on Computers*, vol. 50, No. 1, pp. 83-88, Jan. 2001.
29. A. Reyhani-Masoleh, and A. Hassan, A New Construction of Massey-Omura Parallel Multiplier over GF(2^m), *IEEE Transactions on Computers*, vol. 51, No. 5, pp. 511-520, May 2002.
30. G. Agnew, R. Mullin, I. Onyszchuk, and S. Vanstone, An Implementation for a Fast Public-Key Cryptosystem, *Journal of Cryptology*, vol. 3, pp. 63-79, 1991.
31. A. Reyhani-Masoleh, and A. Hassan, Efficient Digit-Serial Normal Basis Multipliers over GF(2^m), *ACM Transactions on Embedded Computer Systems*, special issue on embedded systems and security, vol. 3, No. 3, pp. 575-592, Aug. 2004.
32. A. Reyhani-Masoleh, and A. Hassan, Low Complexity Word-Level Sequential Normal Basis Multiplier, *IEEE Transaction on Computers*, vol. 54, N0. 2, Feb. 2005.
33. P. Ning, and Y. Yin, Efficient Software Implementation for Finite Field Multiplication in Normal Basis, *ICICS 2001*, LNCS 2229, pp. 177-188, Springer-Verlag, 2001.
34. A. Reyhani-Masoleh, and A. Hassan, Fast Normal Basis Multiplication Using General Purpose Processors, *IEEE Transaction on Computers*, vol. 52, N0. 11, Nov. 2003.
35. G. Gong, and K. M. Khoo, Additive Autocorrelation of Resilient Boolean Functions, *Selected Areas in Cryptography 2003*, LNCS 3006, Springer-Verlag, 2004.

Appendix A
C_N Matrix for Normal Basis Multiplier in $\mathrm{F}_{2^{29}}$

$$C_N = \begin{pmatrix}
0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&1&0&0&0&0&0&0&0&0&0&0&0\\
0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&1&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&1&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0\\
0&0&0&0&1&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&1&0&0&0&0&0&0&0\\
0&0&0&1&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&1&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&1&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&1&0&0&0&0&0&0\\
0&0&0&0&0&1&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&1&0&0&0\\
0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1\\
0&1&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&1&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&1&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&1&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&1
\end{pmatrix}$$

Appendix B
Test Vectors for 32 bit and 64 bit IVs

Test vectors for WG cipher; Each key, IV and keystream is given in smallendian format (LSB $\cdots$ MSB) in hexadecimal. For example binary representation of key = 8000000000000000000 is 1000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 (least significant bit $\cdots$ most significant bit). Similarly binary representation of IV = 01234567 is 0000 0001 0010 0011 0100 0101 0110 0111 (least significant bit $\cdots$ most significant bit)

80 bit key and 32 bit IV

key = 80000000000000000000
IV = 01234567
keystream =9E4AA53B684E2B9AF3444BF556AE4944

key = ABCDABCDABCDABCDABCD
IV = 01234567
keystream = 059344F2E99F5581AD5601C85CBE5D9B

96 bit key and 32 bit IV

key = A00000000000000000000000
IV = 01234567
keystream = 1C620A7740E2F799636A0DF142EF8E6E

key = ABABABABABABABABABABABAB
IV = 01234567
keystream = 9E5191903DD3346B7182C6FDAE9A6CBF

112 bit key and 64 bit IV

key = B0000000000000000000000000000
IV = 0123456789ABCDEF
keystream = 796A1D6A52BC504DFB9BAE7841A75583

key = 12341234123412341234123412341234
IV = 0123456789ABCDEF
keystream = 767A61AC7556377CCBE743A72B9A5A9C

128 bit key and 64 bit IV

key = C00000000000000000000000000000000
IV = 0123456789ABCDEF
keystream = 8EF6E4190F0372D3CCBFE641563424BA

key = 56785678567856785678567856785678

IV = 0123456789ABCDEF
keystream = 63362C0BFD6A86B64AA4953C5466BE57

Appendix C
Test Vectors for Same Length Key and IV

Test vectors for WG cipher; Each key, IV and keystream is given in smallendian format (LSB $\cdots$ MSB) in hexadecimal. For example binary representation of key = 8000000000000000000, is 1000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 (least significant bit $\cdots$ most significant bit). Similarly binary representation of IV = 01234567, is 0000 0001 0010 0011 0100 0101 0110 0111 (least significant bit $\cdots$ most significant bit)

80 bit key and 80 bit IV

key = 80000000000000000000
IV = 01234501234501234501
keystream = C1C64550B84F37281B80961D927DA882

key = 56785678567856785678
IV = 01234501234501234501
keystream = B008DAA67C11BAD30E70E80AD4B007D7

96 bit key and 96 bit IV

key = B00000000000000000000000
IV = 012345012345012345012345
keystream = E27B2C072782ACE5DEC995559D090950

key = 567856785678567856785678
IV = 012345012345012345012345
keystream = 9A677F523776A789403F2ED6712DA566

112 bit key and 112 bit IV

key = A0000000000000000000000000000000
IV = 01234501234501234501234501234501
keystream = 695E126379DAB568AB5D962AB11EB72A

key = 56785678567856785678567856785678
IV = 01234501234501234501234501234501
keystream = 7C5CA96F9B9F604D792A55E175ECE33E

128 bit key and 128 bit IV

key = C0000000000000000000000000000000
IV = 0123456789ABCDEF0123456789ABCDEF
keystream = 1ECE4F577D0E71B2FA8580A81F676E12

key = 56785678567856785678567856785678
IV = 0123456789ABCDEF0123456789ABCDEF
keystream = 7AC9AB808B0836197E377BE48117AEE6

Appendix D
C Implementation of a Normal Basis Multiplier in $F_{2^{29}}$

Following is the C implementation (32 bit word size) of the normal basis multiplier for the normal basis defined by γ in Section 3.1. The implementation is based on the algorithm given in [33].

```
#define ROTL29(v, n)
(unsigned)(((v) ≪ (n)) | ((v) ≫ (29 - (n))))& 0xFFFFFFF8

void mult(unsigned int a, unsigned int b, unsigned int* c){

    unsigned int A[29], B[29];

    /*precomputation*/
    A[0]=a & 0xFFFFFFF8; B[0]=b & 0xFFFFFFF8;
    A[1]=ROTL29(A[0], 1); B[1]=ROTL29(B[0],1);
    A[2]=ROTL29(A[0], 2); B[2]=ROTL29(B[0],2);
    A[3]=ROTL29(A[0], 3); B[3]=ROTL29(B[0],3);
    A[4]=ROTL29(A[0], 4); B[4]=ROTL29(B[0],4);
    A[5]=ROTL29(A[0], 5); B[5]=ROTL29(B[0],5);
    A[6]=ROTL29(A[0], 6); B[6]=ROTL29(B[0],6);
    A[7]=ROTL29(A[0], 7); B[7]=ROTL29(B[0],7);
    A[8]=ROTL29(A[0], 8); B[8]=ROTL29(B[0],8);
    A[9]=ROTL29(A[0], 9); B[9]=ROTL29(B[0],9);
    A[10]=ROTL29(A[0], 10); B[10]=ROTL29(B[0],10);
    A[11]=ROTL29(A[0], 11); B[11]=ROTL29(B[0],11);
    A[12]=ROTL29(A[0], 12); B[12]=ROTL29(B[0],12);
    A[13]=ROTL29(A[0], 13); B[13]=ROTL29(B[0],13);
    A[14]=ROTL29(A[0], 14); B[14]=ROTL29(B[0],14);
    A[15]=ROTL29(A[0], 15); B[15]=ROTL29(B[0],15);
    A[16]=ROTL29(A[0], 16); B[16]=ROTL29(B[0],16);
    A[17]=ROTL29(A[0], 17); B[17]=ROTL29(B[0],17);
    A[18]=ROTL29(A[0], 18); B[18]=ROTL29(B[0],18);
    A[19]=ROTL29(A[0], 19); B[19]=ROTL29(B[0],19);
```

```
    A[20]=ROTL29(A[0], 20); B[20]=ROTL29(B[0],20);
    A[21]=ROTL29(A[0], 21); B[21]=ROTL29(B[0],21);
    A[22]=ROTL29(A[0], 22); B[22]=ROTL29(B[0],22);
    A[23]=ROTL29(A[0], 23); B[23]=ROTL29(B[0],23);
    A[24]=ROTL29(A[0], 24); B[24]=ROTL29(B[0],24);
    A[25]=ROTL29(A[0], 25); B[25]=ROTL29(B[0],25);
    A[26]=ROTL29(A[0], 26); B[26]=ROTL29(B[0],26);
    A[27]=ROTL29(A[0], 27); B[27]=ROTL29(B[0],27);
    A[28]=ROTL29(A[0], 28); B[28]=ROTL29(B[0],28);


    /* multiplication */
    *c=A[0] & B[1];
    *c^= A[1] & (B[0] ^B[21]);
    *c^= A[2] & (B[6] ^B[21]);
    *c^= A[3] & (B[13] ^B[18]);
    *c^= A[4] & (B[11] ^B[27]);
    *c^= A[5] & (B[17] ^B[20]);
    *c^= A[6] & (B[2] ^B[22]);
    *c^= A[7] & (B[13] ^B[25]);
    *c^= A[8] & (B[9] ^B[10]);
    *c^= A[9] & (B[8] ^B[14]);
    *c^= A[10] & (B[8] ^B[26]);
    *c^= A[11] & (B[4] ^B[14]);
    *c^= A[12] & (B[17] ^B[24]);
    *c^= A[13] & (B[3] ^B[7]);
    *c^= A[14] & (B[9] ^B[11]);
    *c^= A[15] & (B[24] ^B[26]);
    *c^= A[16] & (B[19] ^B[23]);
    *c^= A[17] & (B[5] ^B[12]);
    *c^= A[18] & (B[3] ^B[22]);
    *c^= A[19] & (B[16] ^B[27]);
    *c^= A[20] & (B[5] ^B[28]);
    *c^= A[21] & (B[1] ^B[2]);
    *c^= A[22] & (B[6] ^B[18]);
    *c^= A[23] & (B[16] ^B[25]);
    *c^= A[24] & (B[12] ^B[15]);
    *c^= A[25] & (B[7] ^B[23]);
    *c^= A[26] & (B[10] ^B[15]);
    *c^= A[27] & (B[4] ^B[19]);
    *c^= A[28] & (B[20] ^B[28]);
}
```

Appendix E
C Implementation of a Normal Basis Inverter in $F_{2^{29}}$

The following C implementation (32 bit word size) uses the normal basis multiplication to find inverse of an element in $F_{2^{29}}$. The following implementation is based on the algorithm given in [25].

```
#define ROTL29(v, n)
(unsigned)(((v) ≪ (n)) | ((v) ≫ (29 - (n))))& 0xFFFFFFF8
#define ROTR29(v, n) ROTL29(v, 29 - (n))


void inverse(unsigned int a, unsigned int* b){
    b=a
    /*4*/
    b=ROTL29(b, 16);
    mult(b, ROTR29(b, 8), b);
    mult(b, a, b);
    /*3*/
    b=ROTL29(b, 8);
    mult(b, ROTR29(b, 4), b);
    mult(b, a, b);
    /*2*/
    b=ROTL29(b, 4);
    mult(b, ROTR29(b, 2), b);
    /*1*/
    mult(b, ROTR29(b, 1), b);
}
```