Word-Based FCSRs with Fast Software Implementations

Dong Hoon Lee and Sangwoo Park

Abstract: **Feedback with carry shift registers (FCSRs) over 2-adic number would be suitable in hardware implementation, but they are not efficient in software implementation since their basic unit (the size of register cells) is 1-bit. In order to improve the efficiency we consider FCSRs over 2^ℓ-adic number (i.e., FCSRs with register cells of size ℓ-bit) that produce ℓ bits at every clocking where ℓ will be taken as the size of normal words in modern CPUs (e.g., $\ell = 32$). But, it is difficult to deal with the carry that happens when the size of summation results exceeds that of normal words. We may use long variables (declared with '`unsigned __int64`' or '`unsigned long long`') or conditional operators (such as '`if`' statement) to handle the carry, but both the arithmetic operators over long variables and the conditional operators are not efficient comparing with simple arithmetic operators (such as shifts, maskings, xors, modular additions, etc.) over variables of size ℓ-bit. In this paper, we propose some conditions for FCSRs over 2^ℓ-adic number which admit fast software implementations using only simple operators. Moreover, we give two implementation examples for the FCSRs. Our simulation result shows that the proposed methods are twice more efficient than usual methods using conditional operators.**

Index Terms: **Feedback with carry shift register (FCSR), software implementation, stream cipher.**

I. INTRODUCTION

In general the component of stream ciphers consists of two parts: One is to generate long sequences and the other is to produce a key stream from the sequences. At least one of the two components should have a nonlinear property to resist several attacks and cryptanalyses. Classical stream ciphers are based on linear feedback shift registers (LFSRs) as a function of generating long sequences. However, most LFSR-based stream ciphers are vulnerable to recent algebraic attacks [1]–[3]. Hence, there have been many efforts to apply nonlinear feedback shift registers (NFSRs) to stream ciphers.

Feedback with carry shift registers (FCSRs) were introduced by Klapper and Goresky in [4] and [5]. FCSRs can be regarded as LFSRs with ordinary addition over the integer ring with memory for storing the carry instead of addition over $\mathbb{F}_2$. The properties of FCSRs are also analogous to those of LFSRs, but based on 2-adic number rather than power series over $\mathbb{F}_2$. FCSRs over 2-adic number were extended to FCSRs over ramified extensions and finite fields, respectively [6], [7]. As noticed in [5], FCSRs over 2-adic integers can be straightforwardly generalized to FCSRs over p-adic integers where p is a prime number.

Unfortunately, the above FCSRs do not have efficient software implementations yet. For example, the original FCSRs and their variants update their register cells bit-by-bit and output one bit at every clocking since the size of the register cells is one bit. On the other hand, they may be suitable for hardware implementation.

In software implementation, we can extend the size of register cells from 1-bit to ℓ-bit where ℓ is the size of normal words in the given CPU (e.g., $\ell = 32$) to produce ℓ bits at every clocking. Such FCSRs may be regarded as FCSRs over b-adic number where $b = 2^\ell$. Here, b-adic number is denoted by a formal power series with base b. This concept was already introduced by Marsaglia and Zaman, and Goresky and Klapper to design random number generators in [8] and [9]. However, the implementation of such FCSRs is not trivial because of handling carry values. The simplest method is to use large variables of size 2ℓ-bit (declared with '`unsigned __int64`' or '`unsigned long long`' which depends on compilers). While almost all CPUs provide a large variable type, the efficiency of operations with large variable types would be very low. Another tricky method is to use conditional operators (such as '`if`' statement). But, the efficiency is also rather low.

In this paper, we propose some conditions for FCSRs over 2^ℓ-adic number which admit fast software implementations using only simple operators over normal ℓ-bit words (such as shifts, maskings, xors, and modular addition with modulus 2^ℓ). Moreover, we give two implementation examples for the FCSRs. Our simulation result shows that the proposed methods are twice more efficient than usual methods using conditional operators.

This paper is organized as follows. We briefly review some preliminaries about FCSRs over 2-adic number and b-adic number where $b = 2^\ell$ in the following section. In Section III, we propose some conditions on the connection integers to realize FCSRs with only simple operators and describe implementation methods. In Section IV, we validate our claims by giving simulation results. Finally, we conclude in Section V.

II. PRELIMINARIES

A. FCSRs over 2-Adic Number

Suppose that an FCSR has a r register cells. Let a_{n-i} be contents of each cell for $i = 1, \cdots, r$ and m_{n-1} be a value of the memory at $(n-1)$th clock. Then the contents of each cell and the memory are updated at nth clock as follows (see Fig. 1).

$$\Sigma_n = \sum_{i=1}^{r} q_i a_{n-i} + m_{n-1}, \tag{1}$$

$$a_n = \Sigma_n \mod 2, \tag{2}$$

$$m_n = \lfloor \Sigma_n / 2 \rfloor. \tag{3}$$

The sequence generated by an FCSR can be regarded as a 2-adic number $\alpha = (a_0 + 2a_1 + \cdots + 2^i a_i + \cdots)$. Since the

Manuscript received June 10, 2009; approved for publication by Huaxiong Wang, Division I Editor, August 13, 2010.

The authors are with the Electronics and Telecommunications Research Institute, Korea, email: {dlee, psw}@ensec.re.kr.

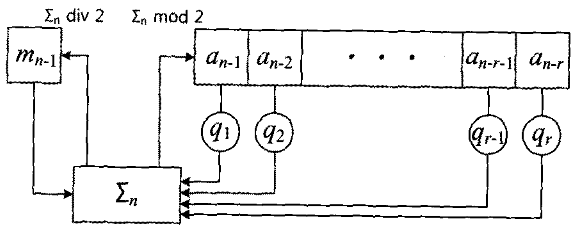


Fig. 1. Structure of FCSRs.

sequence is eventually periodic, α can be represented as a rational number of the form a/q where $q = \sum_{i=1}^{r} q_i 2^i - 1$ and q is called the connection integer of the FCSR. Conversely, for a rational integer a/q, we can construct an FCSR such that the output sequence of the FCSR is equal to the 2-adic representation of a/q. For more details, refer to [5].

B. FCSRs over 2^ℓ-Adic Number

FCSRs over 2-adic number can be extended to that over b-adic number where $b = 2^\ell$ [8]. In other words, the cell contents a_{n-i} and feedback coefficients q_i are ℓ-bit integers. Then, the updating operations are as similar as FCSRs over 2-adic number (cf. (1), (2), and (3)). The difference is that the modulus is b and the memory is updated by $m_n = \lfloor \Sigma_n / b \rfloor$. If $(a_{r-1}, a_{r-2}, \cdots, a_0)$ are initial contents of the register cells, we can also regard the output sequence $(a_0, a_1, \cdots)$ as a b-adic integer $\alpha = \sum_{i=0}^{\infty} a_i b^i$.

In the similar manner of the case of FCSRs over 2-adic number, α can be represented as a rational number a/q where $a = \sum_{k=0}^{r-1} \sum_{i=0}^{k} q_i a_{k-i} b^k - m_{r-1} b^r$, $q = q_1 b + q_2 b^2 + \cdots + q_r b^r - 1$, and $q_0 = -1$. Note that if a/q is the associated rational number of an FCSR over b-adic number, we can also construct an ordinary FCSR over 2-adic number whose output is same as a/q. Of course, the connection integer is also q, but the initial contents may be different from the given (ℓ-bit) initial contents of the FCSR over b-adic number. Conversely, if a connection integer q of an FCSR over 2-adic number satisfies that $b \mid (q+1)$ then we can construct an FCSR over b-adic number whose output is the same as that of the FCSR over 2-adic number.

III. SOFTWARE IMPLEMENTATION OF FCSR

The problem of implementing (original) FCSRs in software is that the size of register cells is only 1 bit while the least unit of a computer word is 8-bit (called byte). So we have to handle individual bits of each register and it forces to consume much cost. This is the reason why we are interested in word based FCSRs whose register cells consist of ℓ-bit integers. Throughout the paper, we assume that FCSRs are based on words of size 32-bit (usual computer word size).

However, we have another difficulties even with word based FCSRs since multiplications with 32-bit integers are also difficult to implement in software. We may avoid this problem by taking register coefficients q_i's with low weights. Then, we can implement a multiplication by q_i with several shifts and additions. It is not difficult to find FCSRs with low weight register coefficients.

We still have the problem of handling carry values which happens when the summation result is larger than 2^{32}. Since the addition provided by general compiler is usually modular addition (with modulus 2^{32}), the carry would disappear without special handling for carry.

A naive method for obtaining $c = a + b \mod 2^{32}$ and $m = (a+b)$ div 2^{32} for two 32-bit integers a and b uses 64-bit variables declared with 'unsigned __int64' as follows (C expression).

```
unsigned __int64    temp;
temp = (unsigned __int64)a
     + (unsigned __int64)b;
c = temp & 0xFFFFFFFF;
m = (temp >> 32).
```

The above method looks simple and is easy to understand. However, the efficiency is very low since the operators with 64-bit operands are inefficient. Another tricky method uses conditional operators as follows (C expression).

```
c = a + b;
m = 0;
if(c < a) m = 1; // carry happens.
```

Of course, we can use ternary conditional operator instead of 'if' operator: `m=((c < a) ? 1 : 0)`. Unfortunately, these conditional operators are also somewhat inefficient comparing with unary or binary operators such as compliments, shifts, modular additions, bitwise-ands (maskings), xors, etc. Thus, we want to implement FCSRs without conditional operators. In this section, we propose FCSRs which admit fast software implementation using only efficient operators.

A. FCSRs with Implementation Using Full-Size Words

Theorem 1: Let $q = \sum_{i=1}^{r} q_i (2^{32})^i - 1$ and w be the Hamming weight of $q+1$. Let k_i be the maximum number such that $2^{k_i} \mid q_i$. If the following conditions are satisfied, we can implement the FCSR associated with the connection integer q and the initial memory m_{r-1} without conditional operators.

1. $\min\{k_i\} \geq \lceil \log_2(w) \rceil$.
2. $\sum_{i=1}^{r} q_i - 1 < 2^{32}$.
3. $0 \leq m_{r-1} \leq \sum_{i=1}^{r} q_i - 1$.

Proof: Note that the contents are updated at nth clock as follows.

$$\Sigma_n = \sum_{i=1}^{r} q_i a_{n-i} + m_{n-1},$$
$$a_n = \Sigma_n \mod 2^{32},$$
$$m_n = \lfloor \Sigma_n / 2^{32} \rfloor.$$

By the third condition, the content of memory cell at every clocking satisfies the same inequality $0 \leq m_n \leq \sum_{i=1}^{r} q_i - 1$ for any $n \geq r-1$ by mathematical induction.

$$\begin{aligned} m_n &= \lfloor (\sum_{i=1}^{r} q_i a_{n-i} + m_{n-1}) / 2^{32} \rfloor \\ &\leq \lfloor (\sum_{i=1}^{r} q_i a_{n-i} + \sum_{i=1}^{r} q_i - 1) / 2^{32} \rfloor \\ &= \lfloor (\sum_{i=1}^{r} q_i (a_{n-i} + 1) - 1) / 2^{32} \rfloor \end{aligned}$$

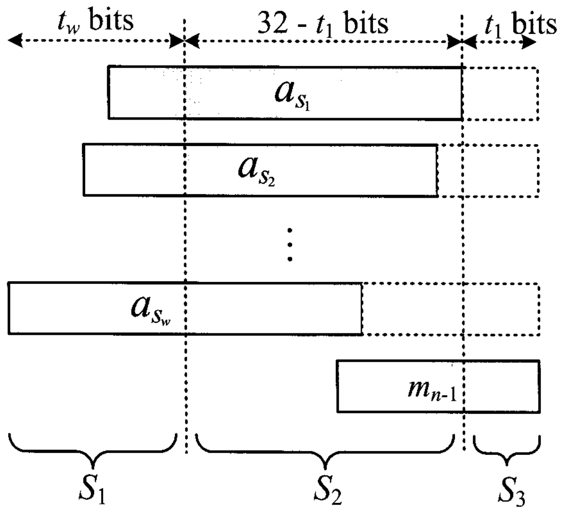


Fig. 2. Σ_n is split into 3 parts.

$$\le \sum_{i=1}^{r} q_i - \lfloor 1/2^{32} \rfloor = \sum_{i=1}^{r} q_i - 1.$$

Then, the second condition forces the content of the memory not to exceed a 32-bit integer, so we do not care about the carry in the memory.

Now, we describe how to update the registers using only normal operators. Let $q_i = \sum_j 2^{t_{i,j}}$ be the binary expression of q_i. Then, $\sum_i q_i a_{n-i} = \sum_{i,j} 2^{t_{i,j}} a_{n-i}$. So we may assume that $\Sigma_n = \sum_{j=1}^{w} 2^{t_j} a_{s_j} + m_{n-1}$ for $t_1 \le t_2 \le \cdots \le t_w$ by reordering indices $t_{i,j}$ since the Hamming weight of $(q+1)$ is w. We will split Σ_n into three parts (see Fig. 2).

Each part can be computed as follows.

$$S_1 = \sum_{j=1}^{w} \lfloor a_{s_j} / 2^{32-t_j} \rfloor,$$
$$S_2 = \sum_{j=1}^{w} (a_{s_j} \bmod 2^{32-t_j}) 2^{t_j - t_1} + \lfloor m_{n-1}/2^{t_1} \rfloor,$$
$$S_3 = m_{n-1} \bmod 2^{t_1}.$$

It is easy to show that $\Sigma_n = S_1 2^{32} + S_2 2^{t_1} + S_3$ since $a = \lfloor a/t \rfloor t + (a \bmod t)$.

Since S_1 is a part of the memory, it is less than 2^{32}. S_3 is also trivially less than 2^{32}. We will show that S_2 should be less than 2^{32}. By the definition of S_2, we have

$$S_2 \le \sum_{j=1}^{w} (2^{32-t_1} - 2^{t_j - t_1}) + \lfloor m_{n-1}/2^{t_1} \rfloor$$
$$\le 2^{32-(t_1 - \lceil \log_2(w) \rceil)} - \sum_{j=1}^{w} (2^{t_j - t_1}) + \lfloor m_{n-1}/2^{t_1} \rfloor.$$

We already described that $m_{n-1} \le \sum_{i=1}^{r} q_i - 1$ for $n \ge r$ in the first paragraph of this proof. Since $\sum_{i=1}^{r} q_i = \sum_{j=1}^{w} 2^{t_j}$, we have $m_{n-1} \le \sum_{j=1}^{w} 2^{t_j} - 1$. Hence,

$$\lfloor m_{n-1}/2^{t_1} \rfloor \le \sum_{j=1}^{w} 2^{t_j - t_1} - 1.$$

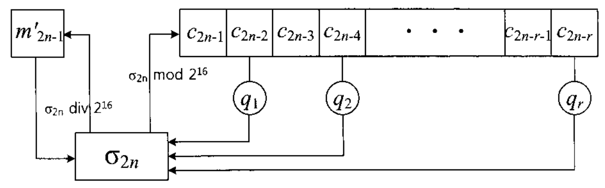


Fig. 3. Structure of the FCSR, $\mathcal{F}_2$.

Then, finally, we have

$$S_2 \le 2^{32-(t_1 - \lceil \log_2(w) \rceil)} - 1 < 2^{32}$$

since $t_1 \ge \lceil \log_2(w) \rceil$. Therefore, we can handle S_1, S_2, and S_3 without any overflow (exceeding 32-bit).

On the other hand, a_n and m_n can be computed as follows.

$$a_n = (S_2 \bmod 2^{32-t_1}) 2^{t_1} + S_3,$$
$$m_n = S_1 + \lfloor S_2 / 2^{32-t_1} \rfloor.$$

Note that all the operations used in computing S_1, S_2, S_3, a_n, and m_n are $\lfloor \cdot / 2^{t_j} \rfloor$, mod 2^{t_j}, multiplication by 2^{t_j}, and addition (whose result is less than 2^{32}), so they are implemented with only shifts, maskings, xors, and modular additions. Furthermore, conditional operators need not be used. □

B. FCSRs with Implementation Using Half-Size Words

In our second method, we fill the register cells (declared with 32-bit) with 16 bits. In other words, we regard the size of the register cells is 16-bit. Of course, the number of register cells should be double comparing with the given FCSR over 2^{32}-adic number. By this way, we can put the summation result within 32-bit, hence do not care about the carry disappearing due to overflow.

The problem is how to maintain the consistency between the given FCSR (over 2^{32}-adic number) and the new FCSR (over 2^{16}-adic number). By the following theorem, we can split each cells into two parts of the same size and implement naturally if the connection integer satisfies $\sum q_i - 1 < 2^{16}$.

Theorem 2: Let $q = \sum_{i=1}^{r} q_i (2^{32})^i - 1$ be the connection integer of the given FCSR, $\mathcal{F}_1$ with r register cells of 32-bit. Let $(a_{r-1}, \cdots, a_0)$ and m_{r-1} be the initial contents of register cells and the memory of $\mathcal{F}_1$, respectively. Let $\mathcal{F}_2$ be an FCSR with $2r$ register cells of 16-bit whose connection integer is q and $(c_{2r-1}, c_{2r-2}, \cdots, c_1, c_0)$ and m'_{2r-1} as the initial contents of $\mathcal{F}_2$ such that

$$a_i = c_{2i+1} 2^{16} + c_{2i} \text{ and } m_{r-1} = m'_{2r-1} \le \sum_{i=1}^{r} q_i - 1$$

for $i = 0, \cdots, r-1$. If $\sum q_i - 1 < 2^{16}$ and $q_i < 2^{16}$, then the output sequences of $\mathcal{F}_1$ and $\mathcal{F}_2$ are identical.

Proof: Since each q_i is less than 2^{16}, the FCSR, $\mathcal{F}_2$ associated with the connection integer q can be depicted in Fig. 3.

The updating processes of $\mathcal{F}_2$ at $2n$th and $2n+1$th clock are as follows.

$$\sigma_{2n} = c_{2n-2} q_1 + \cdots + c_{2n-2r} q_r + m'_{2n-1},$$

$$c_{2n} = \sigma_{2n} \bmod 2^{16},$$
$$m'_{2n} = \lfloor \sigma_{2n}/2^{16} \rfloor$$

and

$$\sigma_{2n+1} = c_{2n-1}q_1 + \cdots + c_{2n-2r+1}q_r + m'_{2n},$$
$$c_{2n+1} = \sigma_{2n+1} \bmod 2^{16},$$
$$m'_{2n+1} = \lfloor \sigma_{2n+1}/2^{16} \rfloor.$$

Note that $m'_{k-1} \leq \sum_{i=1}^{r} q_i - 1 < 2^{16}$ in the similar manner of the proof of the previous theorem. Then,

$$\sigma_k \leq (2^{16} - 1)\sum q_i + m'_{k-1}$$
$$\leq 2^{32} - 2^{16} + m'_{k-1} < 2^{32}$$

Thus, all the above processes can be implemented within single-word (32-bit) operations.

In order to show the consistency of the two FCSRs, it is enough to show that $a_n = c_{2n+1}2^{16} + c_{2n}$ and $m_n = m'_{2n+1}$. We will use the mathematical induction on n to prove it. Rewriting Σ_n (updating process of $\mathcal{F}_1$), we obtain the following equation.

$$\begin{aligned}\Sigma_n &= \sum_{i=1}^{r} a_{n-i}q_i + m_{n-1}\\ &= \sum_{i=1}^{r}(c_{2n-2i+1}2^{16} + c_{2n-2i})q_i + m'_{2n-1}\\ &= \sigma_{2n+1}2^{16} + \sigma_{2n} - m'_{2n}2^{16}\\ &= \sigma_{2n+1}2^{16} + (\sigma_{2n} \bmod 2^{16}).\end{aligned}$$

Thus, we have

$$\begin{aligned}a_n &= \Sigma_n \bmod 2^{32}\\ &= (\sigma_{2n+1} \bmod 2^{16})2^{16} + (\sigma_{2n} \bmod 2^{16})\\ &= (c_{2n+1}2^{16} + c_{2n})\end{aligned}$$

and $m_n = \lfloor \Sigma_n/2^{32} \rfloor = \lfloor \sigma_{2n+1}/2^{16} \rfloor = m'_{2n+1}$. □

IV. SIMULATION RESULTS

In this section, we implement an FCSR to validate the correctness of the claims in the previous section. Let $q = 8 \cdot 2^{32} + 4 \cdot 2^{96} + 8 \cdot 2^{160} - 1$ be a connection integer which satisfies all the conditions in Theorem 1 and Theorem 2. Of course, q is a prime number and $q - 1$ is factored into $q - 1 = 2 \cdot 3^3 \cdot 19p$ for a prime number p. The corresponding FCSR has 5 register cells $L[4], L[3], \cdots, L[0]$.

We implement the FCSR using different methods and compare the efficiency of them. The first method uses conditional operations such as `if`. The next uses 64-bit register provided computer CPU (declared with `unsigned __int64` or `unsigned long long`). In this case, the code looks very simple since the summation result can be represented only one variable. The last two methods are based on the previous section. We will describe the C expression of the main part in the `for` loop of each method. In the following description, `L[i]` will be shifted to `L[i-1]` and the last cell `L[4]` will be updated by the FCSR. 'm' is denoted by a previous memory which is updated at every step.

1. Use `if` statement:
```
sigma = 8*L[4] + m;
m = (L[4] >> 29) + (L[2] >> 30) + (L[0] >>29);
if(sigma < m)              m++;
sigma2 = 4*L[2] + 8*L[0];
if(sigma2 < 4*L[2])        m++;
L[0] = L[1];     L[1] = L[2];
L[2] = L[3];     L[3] = L[4];
L[4] = sigma + sigma2;          // update state
if(L[4] < sigma)           m++;
```
2. Use 64-bit variables declared with `unsigned __int64` or `unsigned long long` statement:
```
unsigned __int64     L[5], sigma;
sigma = (L[4]<<3) + (L[2]<<2) + (L[0]<<3) + m;
L[0] = L[1];     L[1] = L[2];
L[2] = L[3];     L[3] = L[4];
L[4] = sigma & 0xFFFFFFFF;        // update state
m = (sigma >> 32);                // update memory
```
3. Our method1 (using full-size word):
```
S1 = (L[4] >> 29) + (L[2] >> 30) + (L[0] >>29);
S2 = ((L[4] & 0x1FFFFFFF) << 1)
   + (L[2] & 0x3FFFFFFF)
   + ((L[0] & 0x1FFFFFFF) << 1) + (m>>2);
L[0] = L[1];     L[1] = L[2];
L[2] = L[3];     L[3] = L[4];
L[4] = (sigma << 2) + (m & 0x3);
m = S1 + (sigma >> 30);
```
4. Our method2 (using half-size word): In this method, all variables are declared with `unsigned int` (32-bit), but their contents are filled with only 16 bits. Thus, we need 10 register cells, $L[9], L[8], \cdots, L[0]$.
```
unsigned int L[10], sigma1, sigma2;
sigma1 = (L[8]<<3) + (L[4]<<2) + (L[0]<<3) + m;
m = (sigma1 >> 16);
sigma2 = (L[9]<<3) + (L[5]<<2) + (L[1]<<3) + m;
m = (sigma2 >> 16);
L[0] = L[1];     L[1] = L[2];
L[2] = L[3];     L[3] = L[4];
L[4] = L[5];     L[5] = L[6];
L[6] = L[7];     L[7] = L[8];
L[8] = sigma1 & 0xFFFF;
L[9] = sigma2 & 0xFFFF;
```

We perform the above C codes under three different environments (Intel Pentium 4 (@3.4 Ghz, 2 GB RAM), Intel Core2 T7200 (@2.0 Ghz, 2 GB ram), and TI DSP TMS320C64xx (by simulating program called Code Composer (CC) 3.3.38.2 in Pentium 4)). In Pentium 4, we compile the same C source code using different compilers, Microsoft Visual C++ 6.0 (MVC) and GNU compiler collection (GCC) 3.4.4 within Cygwin environment. In Core2, we use Microsoft Visual Studio .Net 2003 as a compiler. For the case of TMS320C64xx, we simulate the environment using CC at 200 Mhz clock speed.

In the above environment, we generate 2^{28} times of 32-bit and measure the performance by the duration time. The following table shows that our proposed methods are more efficient than other methods. In particular, ours are at least twice faster in Intel CPU environments.

V. CONCLUSION

In this paper, we dealt with FCSRs whose register cells are ℓ-bit integers in order to improve efficiency in software implementation (e.g., $\ell = 32$). We may use large variables of size

Table 1. Comparison of time (in sec.) generating 2^{28} 32-bit words.

	`if`	`__int64`	ours-full	ours-half
P4 (MVC)	3.86	7.35	1.10	1.43
P4 (GCC)	4.76	2.65	1.10	1.37
T7200 (.Net)	3.25	8.87	1.28	1.31
TMS (CC)	10.73	13.41	8.06	6.70

2ℓ-bit or conditional operators (such as '`if`' statement), but the efficiency is very low. So we have proposed 2^{32}-adic FCSRs which admit fast software implementation using only simple operators such as shifts, maskings, and modular additions without conditional operators. As a consequence our methods are more efficient than others.

REFERENCES

[1] F. Armknecht, "Improving fast algebraic attacks," in *Proc. FSE*, LNCS 3017, Springer-Verlag, 2004, pp. 65–82.

[2] F. Armknecht and M. Krause, "Algebraic attacks on combiners with memory," in *Proc. Advances in Cryptology–Crypto*, LNCS 2729, Springer-Verlag, 2003, pp. 162–175.

[3] N. Courtois and W. Meier, "Algebraic attacks on stream ciphers with linear feedback," in *Proc. Advances in Cryptology–Eurocrypt*, LNCS 2656, Springer-Verlag, 2003, pp. 345–359.

[4] A. Klapper and M. Goresky, "2-adic shift registers," in *Proc. FSE*, LNCS 809, Springer-Verlag, 1994, pp. 174–178.

[5] A. Klapper and M. Goresky, "Feedback shift registers, 2-adic span, and combiners with memory," *J. Cryptology*, vol. 10, pp. 111–147, 1997.

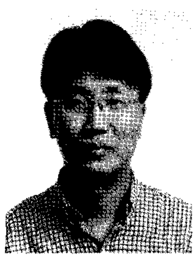
[6] A. Klapper and M. Goresky, "Feedback registers based on ramified extensions of the 2-adic numbers," in *Proc. Advances in Cryptology–Eurocrypt*, LNCS 950, Springer-Verlag, 1995, pp. 215–222.

[7] A. Klapper, "Feedback with carry shift registers over finite fields," in *Proc. FSE*, LNCS 1008, 1995, pp. 170–178.

[8] M. Goresky and A. Klapper, "Efficient multiply-with-carry random number generators with maximal period," *ACM Trans. Modeling and Computer Simulation*, vol. 13, pp. 310–321, Oct. 2003.

[9] G. Marsaglia and A. Zaman, "A new class of random number generators," *Annals Appli. Probability*, vol. 1, pp. 462–480, 1991.

Dong Hoon Lee received his the B.S. degree in Mathematical Education from Seoul National University in 1994. He also received the M.S. and the Ph.D. degrees in Mathematics from Korea Advanced Institute of Science and Technology (KAIST) in 1996 and 2000, respectively. From 2000 to 2002, he had worked at Cryptography and Network Security Center of Future Systems Inc. Since 2002, he has been with Electronics and Telecommunications Research Institute (ETRI) as a Research Member. His interests include computational number theory and design and analysis of cryptographic algorithms.



Sangwoo Park received his the B.S. degree in Mathematical Education from Korea University in 1989. He also received the M.S. and the Ph.D. degrees in Mathematics from Korea University in 1991 and 2003, respectively. From August 2003 to August 2004, he worked for NIST as a Guest Researcher. He is a Principal Member of Research Staff at Electronics and Telecommunications Research Institute (ETRI). His major interests are cyber security and design and analysis of cryptographic algorithms.