A Time and Storage Efficient Solution to Remote File Integrity Check

Sarad AV, Sankar K, Vipin M
AU-KBC Research Centre,
Anna University, India
{*avsarad, sankar, vipintm*}*@au-kbc.org*

Amitabh Saxena
Department of Information and
Communication Technology
University of Trento, Italy
amitabh@dit.unitn.it

Abstract

Checking the integrity of a file on a remote untrusted or compromised server is to be achieved with minimal computational and storage requirements on part of a healthy verifier. Existing solutions are time and storage intensive. A fast protocol comprising of maximum period linear congruence generators and linear feedback shift registers with compact storage requirements is proposed.

keywords: Data integrity and privacy, Security verification, Remote file integrity check, Cryptography.

1. Introduction

Validating the integrity of a file on a remote untrusted server assumes significance since its integrity may be compromised by malicious crackers, leaving a backdoor or virus to modify the contents on the server to his advantage [1]. Let *P* be a prover, whose task is to validate the correctness of the file on the remote untrusted server. The remote untrusted server (the verifier *V*) responds to a set of challenges from the prover *P*. Let *F* be the file whose integrity is under question. It is practically taxing to copy the contents of large files (order of a few gigabytes) from the remote server onto a machine with a clean copy of the file and verify its integrity due to bandwidth restrictions. It is also of no advantage to compute a one way hash of the file on the prover side since the verifier would simply replay the correct message digest to a prover *P*. Protocols that validate remote file integrity find applications in grid computing [1], trusted computing frameworks [2], peer to peer networks [3], intercept file system calls and checking file integrity before allowing access [4]. Practically, *P* should be able to verify *V* by storing minimal file information on *F*. Further, the challenge response between *P* and *V* should be secure. i.e. a malicious *V* should not be able to cheat. The other considerations include keeping the communication bandwidth and computational complexity of the protocol minimal. A protocol satisfying the above conditions is proposed. It is also of advantage to run the verification an unlimited number of times and such a scheme is proposed in [5]. This however comes at a high storage requirement very close to the actual size of the file *F* and the storage requirement is proportional to the size of the file *F*. The high computational complexity involved in their scheme is mainly due to the multiplication and binary exponentiation modulo *m*. Each multiplication consumes $O(n^2)$ time and computing (b^n mod m) requires $O((\log_2 n).(\log_2 m)^2)$ time [6]. We propose a protocol whose storage requirement remains small and fixed per challenge irrespective of the number of bits of the file being verified. The operations involved also have a low time complexity which amounts to clocking linear congruence generators, linear feedback shift registers and the use of a fast and secure stream cipher. The proposed protocol however cannot verify an unlimited number of times but in practice is shown to be more efficient than the protocol in [5] in terms of time and storage complexity when the size of *F* is relatively large. Section 2 discusses the mathematical background behind the protocol and section 3 describes the various steps in the protocol. Section 4 discusses experimental results and the main results are concluded in section 5.

2. Mathematical Background

This section discusses the primitives used in the protocol namely linear congruence generators, a cryptographically secure pseudo random number generator (SPRNG) as the keystream output of a fast and secure stream cipher namely HC-256 and a linear feedback shift register. The provably secure Blum Blum Shub quadratic residue generator is also briefly discussed.

2.1. Linear congruence generator

The Linear Congruence Generator (LCG) is given by the recurrence

$X_{n+1} = a \,.\, X_n + c \text{ (modulo } m\text{)}$ (1)

$n \geq 0$ where $0<m$ is the modulus, $0 \leq a<m$ the multiplier, $0 \leq c<m$ the increment, and $0 \leq X_0<m$ the seed. By Theorem A of Knuth [7], the LCG is of maximum period if and only if:

1. c is relatively prime to m;
2. $b = a-1$ is a multiple of p, for every p dividing m;
3. b is a multiple of 4, if m is a multiple of 4.

A closed form of the LCG in obtained by expanding the recurrence in Equation 1. The $n+1^{th}$ element of the generator is given by

$$X_{n+1} = (a^{n+1}.X_0 + c.\sum_{i=0}^{n} a^i) \text{(modulo } m\text{)}$$

$$= (a^{n+1}.X_0 + c.(\,(a^{n+1}-1)/(a-1)\,)\,) \text{ (modulo } m\text{)} \quad (2)$$

If $a = 2^k + 1$, $k>0$; then multiplication simplifies to a k-bit shift and an addition [8]. Furthermore if $m = 2^e$, $e>0$; computing X_{n+1} (modulo m) is equivalent to storing the e least significant bits of X_{n+1}. Hence, generating a new output for this LCG is linear in time and the speed of pseudo random number generation is greatly improved. Mersenne number, generalized Mersenne number and Crandall's primes are other attractive candidates for fast modular reduction due to efficient algorithms [9-11]. Further if $c=0$ and p is prime, $X_{n+1} = a^{n+1}.X_0$ (modulo p); by Equation 2. Let $a \in F_p^*$ be a generating element in the prime field, then $X_0 = a^j$, where j is an arbitrary constant and $a^j \in F_p^*$. Hence $X_{n+1} = a^{n+1}.\, a^j$ (modulo p) (3)

Therefore, finding a prime p where $a=2 \in F_p^*$ is a generator [12] results in the multiplication and exponentiation in equation 3 translating to a binary shift operation.

2.2. Cryptographically secure pseudo random number generation

Linear congruence generators are predictable since its multiplier, increment and modulus can be recovered given a sufficiently long subsequence of the generated output. The cryptanalysis of LCG's is extensively studied [13,14] and surveyed [15]. It is essential to thwart the attacks on the linear congruence generator and at the same time maintain a low computational complexity. To meet this requirement, a fast and cryptographically secure pseudo random (SPRNG) is obtained as the keystream output of the HC-256 [16] stream cipher. The HC-256 stream cipher encryption is very fast in software and is a candidate of the eSTREAM project[17].

2.3. BBS generator and LFSR

The Blum Blum Shub (BBS) [18] is a quadratic residue generator proven to be cryptographically secure and is an excellent candidate for secure pseudo random number generation. However it is slower than other Monte Carlo generators and is an excellent candidate for secure pseudo random number generation if provable security is a compelling factor. Next, a linear feedback shift register (LFSR) is briefly discussed. LFSR's in GF(2) has the advantage that it is extremely fast in hardware and software and hence used in the setup for remote file integrity check to generate pseudo random bits. Let $y_i \in \{0,1\}$, $1 \leq i \leq l$ be the taps of the LFSR and $s=\{s_1,s_2,\ldots,s_l\}$ be the initial loading of the LFSR, $s_i \in \{0,1\}$. Define $Z=\{0,1\}^l$. The feedback function $f: Z \rightarrow Z$ is

$$f(s_1,s_2,\ldots,s_l) = (\sum_{k=1}^{l} y_k.s_k \,, s_1,s_2,\ldots,s_{l-1})$$

When the LFSR is clocked, the contents of the tapped register cells are added modulo 2 and fed back. The bit in the right most register cell is the LFSR output. . For the output of the LFSR to be of maximum period 2^n-1, the polynomial of degree n corresponding to the taps should be primitive modulo 2. An extensive table of primitive polynomials modulo 2 is found in [12].

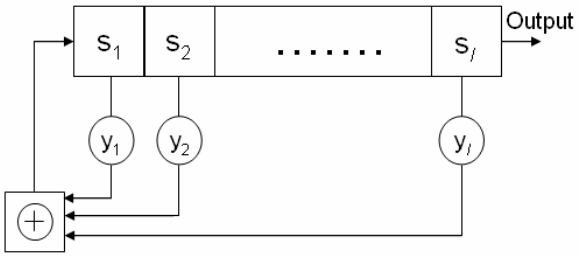

Figure 1. Fibonacci configured LFSR

Algorithms for finding primitive polynomials are available in [19-20]. A dense primitive polynomial of given degree n allow each output bit of the LFSR to be dependent on a larger number of input bits. The primitive polynomial modulo 2 is made public and multiple instances of the same LFSR with different initial seeding are used by both the prover P and the verifier V. The operations on a LFSR are illustrated in Figure 1.

3. The Challenge Response Protocol

Let F be the original untampered file and *fbits* be the total number of bits in it. Let M be the full length bit string in file F. The verifier carries out the operations described below with the help of LCG's, LFSR's and secure pseudo random number generators. The output is a hash like table T but much faster than a fixed length secure one way hash function. The generation of table T involves operations of low time complexity and can be parallelized. The collision probabilities are not as stringent as that of secure one way hash functions but are shown to be reasonable small for practical cases. The entries in T are used by the verifier V to challenge the prover P while checking the integrity of remote file F. Set n to the degree of the primitive polynomial. The challenge response protocol is carried out as follows:

3.1. The verifier pre-computation step

Step 1. Divide F into t equal sized blocks (M_1, M_2,..., M_t); each block of bit size *fblock*. The last block is suitably padded with 0 if its size is less than *fblock*. Let *verify_number* be the total number of block verifications per challenge to detect tampering on file F, *lfsr_pc* be the number of LFSR's used per challenge, *total_challenge* be the total number of challenges on file F and *total_lfsr*= *lfsr_pc*. *total_challenge*. Each of the *lfsr_pc* LFSR's used per challenge is set to maximum period by choosing a primitive polynomial modulo 2. The same primitive polynomial modulo 2 is used for all the LFSR's per challenge and across challenges.

Step 2. Generate two random secret keys ss_0 and ss_1. With ss_0 as key, the HC-256 stream cipher is set to produce a keystream output of *total_challenge* sequence of secure pseudo random numbers, each of size n bits. Let *Secure_seq0* ={$sseq0_1$, $sseq0_2$,..., $sseq0_{total\_challenge}$ }; where $sseq0_i$ ($1 \le i \le$ *total_challenge*) is the sequence output using HC-256 cipher and key ss_0. Similarly, with ss_1 as key, the HC-256 cipher is set to produce a keystream output of another sequence of secure pseudo random numbers. Let *Secure_seq1* ={$sseq1_1$, $sseq1_2$,..., $sseq1_{total\_lfsr}$ }; where $sseq1_i$ ($1 \le i \le$ *total_lfsr*) is the sequence output using HC-256 cipher with key ss_1.

Step 3: Set *counter*←1. For each challenge on file F; i←1 to *total_challenge*, do the following:

1. Use a LCG with publicly known parameters a, c and modulus m ($m \ge$ *verify_number*) Let the publicly known LCG parameters constitute the set *lcg_block*.

1.1. Using element in $Secure\_seq0_i$(modulo m) as seed, in the publicly known LCG with parameters from *lcg_block*

1.1.1. Generate *verify_number* output sequences. Place the generated sequences in array *block_pos*. The value of *verify_number* is chosen to match the number of blocks in F to be verified. i.e. *verify_number* $\le t$.

2. Set each of the *lfsr_pc* LFSR's used per challenge to be of maximum period S by using the same primitive polynomial modulo 2.

2.1. For each of the *lfsr_pc* LFSR's per challenge

2.1.1. Seed the LFSR with $Secure\_seq1_{counter}$. Place the generated element in array *lfsr_num*[*counter*].

2.1.2. *counter*←*counter*+1.

3. Invoke the routine to find the final clock count of the LFSR as described. Call routine: *clock_count* ← *Find_LFSR_Clock_Counts*(*F*,*block_pos*,*verify_number* , *lfsr_num, lfsr_pc*)

4. Retrieve *lfsr_pc* number of actual clock counts from the array *clock_count* and then compute array *actual_clock_count*[i][j]← *clock_count*[j]; for $1 \le j \le$ *lfsr_pc*.

4.1. for j←1 to *lfsr_pc* do the following

4.1.1. Compute *actual_clock_count*[i][j] ← *actual_clock_count*[i][j] modulo 2^z. Set j←j+1. The value of z is suitably chosen to minimize the storage requirement on the verifier side.

5. Set i←i+1.

Step 4: Verifier puts the truncated clock counts modulo 2^z in array *actual_clock_count* onto table T as an $i*j$ matrix; where i←*total_challenge* and j←*lfsr_pc*.

Step 5: T is retained and file F is discarded

3.2. The challenge response step

The task is to verify if the verifier side file contents F is the same as the prover side file F^*. The prover P performs the following steps:

Step 1: Set *counter*←1. For each challenge on file F; i←1 to *total_challenge*, do the following:

1. Verifier V gives prover P, $sseq0_i$(modulo m), $sseq1_j$ $\forall$ (*counter*$\le j \le$ *counter* + *lfsr_pc*-1). Prover places the latter sequence $sseq1_j$ in array *lfsr_num*.

2. Prover uses the publicly known LCG with parameters from *lcg_block* to generate *verify_number* sequences with seed $sseq0_i$(modulo m) and places the generated sequences in array *block_pos*.

3. Verifier invokes the routine to find the final clock count of the LFSR and place the value returned by the routine in *clock_count* as described.

Call routine:*clock_count*←*Find_LFSR_Clock_Counts* (*F**, *block_pos*, *verify_number*, *lfsr_num, lfsr_pc*)
4. Retrieve *lfsr_pc* number of actual clock counts from the array *clock_count* and compute array *actual_clock_count*[*i*][*j*]←*clock_count*[*j*];1≤*j*≤*lfsr_pc*.
4.1. for *j*←1 to *lfsr_pc* do the following
4.1.1. Compute *actual_clock_count*[*i*][*j*] ← *actual_clock_count*[*i*][*j*] modulo 2^z.
5. Set *counter*← *counter* + *lfsr_pc*; *i*←*i*+1.
Step 2: Prover *P* places the truncated clock counts modulo 2^y in array *actual_clock_count* onto table *U* as an *i***j* matrix; where *i*←*total_challenge* and *j*←*lfsr_pc*.

3.3. The verification step

Step 1. The Verifier *V* sends table *U* to prover *P*.
Step 2. For each verification challenge initiated by the verifier, Verifier *V* matches each row in table *U* with the corresponding row in its own stored table *T*. If they are similar, the integrity of the file *F* is declared as untampered; otherwise the integrity of file *F* is declared as compromised.

The parameters given by the verifier *V* to prover *P* in Step 1 of the challenge response step in Section 3.2 is exchanged per challenge initiated by *V*, (in a real scenario) so that the integrity check response cannot be precomputed and stored by *P*. The storage requirement at verifier *V* is the space for table *T*, secret keys ss_0 and ss_1. The other parameters are generated with the secret keys when the verifier *V* challenges the prover *P*.

Routine: *Find_LFSR_Clock_Counts*(*File*, *block_pos*, *verify_number*, *lfsr_num, lfsr_pc*)
Use array *clock_val* to hold the clock counts of each of the *lfsr_pc* LFSR's.
1. for *g*←1 to *lfsr_pc*
1.1. Given the *lfsr_pc* LFSR's, seed the g^{th} LFSR with *lfsr_num*[*g*], set *clock_val*[*g*]←0, *g*←*g*+1.
2. for *h*←1 to *verify*_number, do the following:
2.1. Find *block_content*← *Read*(*File*(*block_pos*[*h*])).
2.2. for *e*←1 to *lfsr_pc*
Begin delimiter
for *f*←0 to *fblock*-1 do
2.2.1.Compute *bitval* ← (*block_content* >>*f*) & 1; >> is the right bit shift operator, & is the bitwise AND operator. Set *f*←*f*+1.
2.2.2.*clock_val*[*e*]←*Find_clock*(*clock_val*, *bitval, e*).
End delimiter
Set *e*←*e*+1. End for loop section 2.2
Set *h*←*h*+1. End for loop section 2.
4. return (*clock_val*).

Routine: *Find_next_LFSR_bit*(*e*)
return(next bit output by LFSR *e*)

Routine: *Read*(*File*(*position*]))
return(block of bit size *fblock* from file whose position is specified by *position*)

Routine: *Find_clock*(*clock_val*, *bitval, e*)
1. if(*Find_next_LFSR_bit*(*e*) = *bitval*)
clock_val[*e*] ← *clock_val*[*e*] - 1
2. do
clock_val[*e*] ← *clock_val*[*e*] + 1
while(*Find_next_LFSR_bit*(*e*) ≠ bitval)
3. return(*clock_val*)

Routine *Find_clock* is used to find the integrity check parameter namely *clock_val* and is designed to detect errors introduced into the file stream intentionally or unintentionally; with a high probability. Due to the nature of the runs of the output bits of the LFSR, the intuition behind the routine is to widen the clock count gap from the actual clock count in the presence of errors.

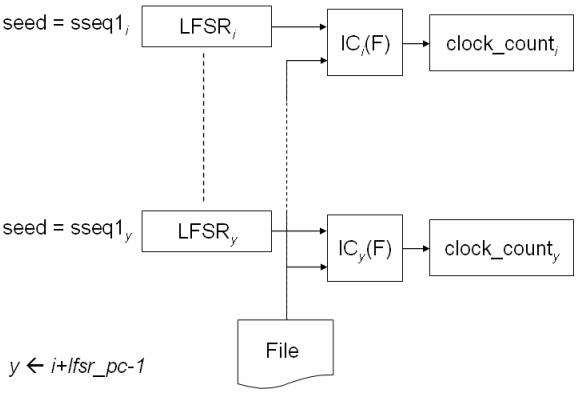

Figure 2. Integrity check computation per challenge

Figure 2 gives an overview of the integrity check value computation per challenge. IC(*F*) is the integrity check function over file *F* for computing *clock_count* corresponding to each LSFR seeding. Further *clock_count*(mod 2^z) is computed for each LFSR and is used for integrity verification per challenge. Function IC in Figure 2 corresponds to routine *Find_clock* run over the desired blocks in file *F*.

4. Experimental Results

The experiment is set by choosing a file *F* of size 1 MB. One thousand copies of *F* with consecutive random one bit error are introduced.

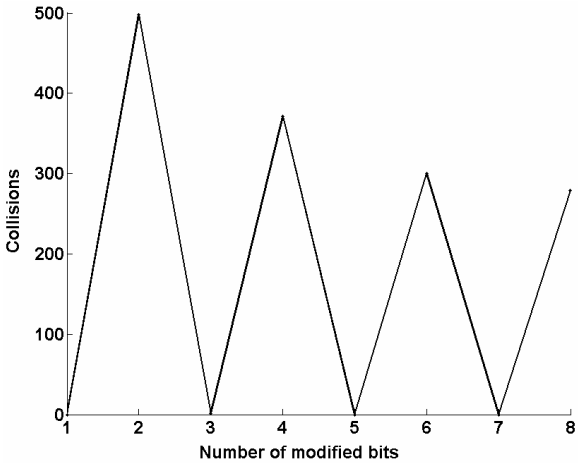

Figure 3. Collisions per 1000 file samples

Similarly, one thousand copies of *F*, each with consecutive random two bit, three bit and upto eight bit error is introduced. All the *t* blocks in F are checked for bit modifications. A collision is said to occur when the *clock_count* generated by a seeded LFSR on an untampered file *F* matches with that of the same file with modified bits or errors introduced into the file stream. The collision statistics per one thousand instances for random consecutive one bit, two bit and upto eight bit errors on file *F* is plotted in Figure 3; using a single LFSR instance. The experiment was repeated for a random bit string file, a text file and image file for file size 10 MB, 5MB and 10 KB respectively. The collisions were found to be consistent with the ratios in Figure 3. The discussion so far considers bit errors or bit modifications to be bit flips in the file stream. Further, random bit deletion and insertion experiments were carried out on file *F* and it was found to give fewer collisions than the bit flip case and hence not discussed further.

For a practical setup, choose *lfsr_pc* = 16 number of LFSR's per challenge. If we consider the worst case collision probability of a singe LFSR's *clock_count* to be 0.5, the probability of *lfsr_pc* number of *clock_count* colliding simultaneously is given as $P_{lfsr\_pc}(\text{colliding})=0.5^{lfsr\_pc}$. The value of *z* is to be chosen sufficiently large to minimize collisions as a result of modular reduction. If *lfsr_pc* = 16, then $P_{16}(\text{colliding})=0.5^{16}$. Choose $z=2^8$ and store *clock_count* (mod 2^8) for each of the 16 LFSR's; which requires a storage space of 16.8=128 bits per challenge. This storage requirement is independent of the size of the file whose integrity is verified.

Table 1. Collision percentage

Number of modified bits	Number of modified positions									
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
1	0	44	0	39	0	36	0	31	0	29
2	49	34	26	33	27	18	25	20	13	15
3	1	27	0	19	0	20	0	16	0	19
4	37	33	26	18	15	24	12	23	10	11
5	0	25	0	20	0	18	0	11	0	6
6	30	26	15	21	10	9	9	14	8	11
7	0	24	0	8	0	14	0	10	0	7
8	27	19	18	15	14	16	15	11	9	8

Critical applications may require verifying the integrity of a remote file every five minutes where as other applications may require upto two verifications per day. If a remote file for the above setup is verified once every five minute for 365 days, it amounts to 128.12.24.365 bits; which is roughly about 1.7 MB of table T that is to be stored on the verifier side. If the remote file verification is done twice a day for 365 days, it amounts to 128.2.365 bits; which is roughly about 12 KB of table T that the verifier needs to store.

For the same experimental setup on the 1 MB file, the entries in Table 1 gives the percentage of collisions with the row indicating the number of random consecutive bit flip introduced onto file *F* and the column indicating the number of instances of such errors introduced onto *F*. It is observed that all collisions in Table 1 are less than 50%; which is calculated using a single LFSR instance. Hence the collision probability using *lfsr_pc* LFSR's; $P_{lfsr\_pc}(\text{colliding})=0.5^{lfsr\_pc}$ is valid.

5. Conclusion

An efficient protocol to verify remote file integrity on an untrusted host with low error probabilities is proposed. It is found that the proposed protocol is time and storage efficient in comparison to existing remote file integrity check protocols and secure against malicious intent. Further, the storage requirement is independent of the number of blocks of the file being verified per challenge. The LFSR's per challenge and across challenges can be used in parallel without any loss of generality to enhance speed up.

6. References

[1] Paolo Falcarin, Riccardo Scandariato, Mario Baldi, Yoram Ofek, “Integrity checking in remote computation”, AICA, Udine, Italy, October 2005

[2] Trusted Computing Group Delivers Trusted Network Connect (TNC) Architecture to Ensure Endpoint Integrity and to Protect Networks from Attacks and Unauthorized Access(2005).

[3] Eu-Jin Goh, Hovav Shacham, Nagendra Modadugu, Dan Boneh: “SiRiUS: Securing Remote Untrusted Storage”. NDSS 2003

[4] Vinícius da Silveira Serafim, Raul Fernando Weber, "The SoFFIC Project".

[5] Francesc Sebé, Josep Domingo-Ferrer, Antoni Martínez-Ballesté, Yves Deswarte and Jean-Jacques Quisquater, ``Efficient remote data possession checking in critical information infrastructures", IEEE Transactions on Knowledge and Data Engineering, 2007(preprint).

[6] Neal Koblitz, *A Course in Number Theory and Cryptography*, Springer-Verlag, New York, 1987. Second edition, 1994.

[7] Donald E. Knuth. *Seminumerical Algorithms*. Addison-Wesley, New York, NY, third edition, 1997.

[8] Martin Greenberger, “Notes on a New Pseudo-Random Number Generator”, J.ACM, 163-167 (1961)

[9] Jerome A. Solinas,”Generalized Mersenne numbers”. Technical report, The centre for applied cryptographic research, University of Waterloo, 1999. CORR 99-39.

[10] Huapeng Wu(2000), “On Modular Reduction”.

[11] J. Guajardo, S. S. Kumar, C. Paar, J. Pelzl, "Efficient Software-Implementation of Finite Fields with Applications to Cryptography", Acta Applicandae Mathematicae: An International Survey Journal on Applying Mathematics and Mathematical Applications, Volume 93, Numbers 1-3, pp. 3-32, September 2006.

[12] Schneier, Bruce. *Applied Cryptography*, John Wiley & Sons, 1994.

[13] Marsaglia, G. 1968. “Random numbers fall mainly in the planes”. Proceedings of the National Academy of Sciences of the United States of America 60:25-28.

[14] Joan Boyar (1989). "Inferring sequences produced by pseudo-random number generators". Journal of the ACM 36 (1): 129 - 141

[15] Ernest Brickel and Andrew Odlyzko, “Cryptanalysis: A survey of recent results”.

[16] Hongjun Wu: “A New Stream Cipher HC-256”. FSE 2004: 226-244

[17] eSTREAM–“Update 1, ECRYPT Network of Excellence”, 2005.

[18] Lenore Blum, Manuel Blum, Michael Shub. "Comparison of two pseudo-random number generators", Advances in Cryptology: Proceedings of Crypto '82.

[19] P Hellekalek, “Study of algorithms for primitive polynomials”, Research Institute of Software Technology, University of Salzburg, April 1994.

[20] T.A. Gulliver, M. Serra and V.K. Bhargava, “The Generation of Primitive Polynomials in GF(q) with Independent Roots and their Applications for Power Residue Codes, VLSI Testing and Finite Field Multipliers Using Normal Basis”, Int. J. Electronics, Vol. 71, No. 4, pp. 559-576, Oct. 1991.