Using Linear Congruential Generators for Cryptographic Purposes

Chung-Chih Li
Computer Science Department
Lamar University
Beaumont, TX 77710

Bo Sun
Computer Science Department
Lamar University
Beaumont, TX 77710

Abstract

We try to provide an alternative attitude toward the use of a Linear Congruential Generator (LCG here after) in generating pseudo-random numbers for some cryptographic purpose. In particular, we choose email encryption as our cryptographic application. Our encryption will be considered secure if the attacker cannot infer the pseudo-random numbers without knowing the parameters of the LCG. We implement Plumstead's inference algorithm [2] for an unknown LCG and our experimental results show that simply increasing the size of the modulus of the LCG does not significantly increase the difficulty of breaking the system. The only way to circumvent the weakness of the LCG, as we conclude, is to hide the generated numbers from the attacker. We suggest a practical attack on the method proposed in [11] and then introduce a much stronger version to patch the loophole without compromising the simplicity of the LCG. Moreover, we speculate that our new version of using the LCG in email encryption may resist the known plaintext attack and, therefore, there is no need to distribute a new set of parameters for the LCG for each encryption.

Keywords: linear congruential generator, email encryption, noise-permutation cipher.

1 Introduction

It is not an overstatement to say that every cryptosystem needs a source of random numbers either in constructing the keys for the encryption algorithm or in generating enough amount of random noises for scrambling the sensitive information. The first case usually needs a small amount of random numbers, and thus some clever methods relying on external random source are practical and affordable. For example, the positions and interval of mouse clicks is a commonly used random source. As for the latter case, it is almost impossible not to use a Pseudo-Random Number Generator (PRNG here after), although, as John von Neumann put it, thinking of any algorithms for generating random numbers is in a state of sin. Ironically, cryptographers pay just little attention to PRNGs and there has been comparatively little research on the topic (see [4], chapter 10). Of course, how to seed a PRNG is a must-consider issue but it falls into the first case and will not be considered here; we refer the reader to [7] for a thorough discussion on the issue.

While the rigorous mathematical definition for true randomness remains unsettled, there have been quite a lot of PRNGs introduced for practical purposes. Park and Miller [12] suggested a "minimal standard" for the implementation of a PRNG, which is not as easy as it may seem. Most of the existent PRNGs are designed for simulation or statistic purposes. Literally all existent PRNGs are not suggested for any cryptosystem because they are all predictable, at least, in theory. Therefore, cryptographers prefer using another cipher that has been proven secure such as AES or SHA-256 as a PRNG in their cryptosystems. This is a legitimate choice to maintain the maximum security of the system as long as the underlying cipher (served as a PRNG) remains secure. However, the trade-off is the efficiency of the system. In many applications, using AES or SHA-256 as a PRNG to generate a great amount of random numbers is an overkill. Encryption for daily life emails, for example, needs not to be a heavyweight one. Instead, regular email encryption is meant to provide a basic protection to ordinary emails just as envelopes to regular postal mails, and the encryption algorithm should be efficient and easy to be intergraded in email softwares.

With these in mind, we started examining some of the widely used PRNGs listed in [3]. They are all proven or believed to be cryptographically insecure under the assumption that an enough amount of numbers in the sequence generated by a fixed PRNG is known to the attacker. Thus, the only way to circumvent this theoretical weakness is to make sure that the assumption is compromised in the real world. We come out with the idea that if we can find a way to appropriately mix together the information itself and the random sequence generated by a PRNG, then the two probably can protect each other. In order to pro-

vide sufficient camouflage for the random sequence, the information itself should have entropy (randomness) at certain level. English (therefore, email exchange) seems to be a good candidate to satisfy this property. For the PRNG, we use the LCG in its simplest form to produce pseudo-random numbers. We pick up the LCG not only because it is the simplest, most efficient, and a well studied pseudo-random numbers generator, but also because it seems to be the most vulnerable PRNG. Thus, a success of our proposal will provide a strong evidence that our approach can also be applied to other less vulnerable PRNGs.

2 Linear Congruential Generators

LCGs have been widely used for a large variety of applications. An LCG generates a sequence of pseudo-random numbers according to some recurrence congruence. The simplest form of the LCG uses the following equation:

$$X_{n+1} = aX_n + b \mod m, \tag{1}$$

where a is called the multiplier, b the increment, and m the modulus. The numbers will be generated in a sequence: $X_0, X_1, X_2, \ldots$, where X_0 should be given in advance called *seed*. We say that a, b, m, and X_0 are the parameters of an LCG. The quality of an LCG depends on the selection of its parameters. For example, in the special case $m = 2^n$, the maximum cycle $(m-1)$ can be obtained under the condition given by M. Greenberger in 1961. A year later, Hull and Dobell gave a well-known condition for the general case. (Details about these conditions and their proofs can be found in [8].) For other criteria of being a "good" PRNG, we refer the reader to more recent and complete surveys in [3, 6]. However, these researchers emphasized on engineering applications, mostly for simulations, and thus put aside the issue of predictableness. In fact, the implementations and their carefully chosen parameters of most commercial LCGs are disclosed for open discussion. Secrecy is not an issue to PRNG designers. In other words, these LCGs do not intend to be used for cryptographic purposes.

A series of investigations on the LCG in late 80's and early 90's have been done and raised a substantial doubt about using the LCG in any cryptosystem. In other words, the LCG is cryptographically insecure in the sense that the attacker can practically recover the entire sequence with a very limited observation on the sequence. As a result, the LCG has been overshadowed by this theoretical fault for more than a decade. However, if we have to take simplicity and efficiency into our primary consideration, we think that the LCG deserves a second thought.

2.1 Predictableness of LCGs

That all sequences generated by the LCG are predictable was first argued by Knuth in [9]. In [2, 1] Boyar picked up the problem and gave a rather complete treatment on the predictableness of some widely used LCGs. In [10] Krawczyk gave an inference algorithm that can predict any sequence generated by the LCG in its most general form. Krawczyk's investigation settled on a final theoretical viewpoint to the inquiry. Meanwhile, Ritter in his survey paper [13] again strongly warned that any attempt to use LCGs for cryptographical purposes is dangerous *unless* the sequence can be isolated from another generator. We thus think of the text of English (or any other natural languages) that may serve as a "random" number generator for such isolation. Although English text certainly can't be seen as a source of random numbers, we believe that the entropy of English is high enough for the purpose.

In order to properly arrange the use of numbers generated by an LCG in our cipher, we need experimental results about how many numbers are actually needed to successfully infer the sequence. We implement Plumstead's inference algorithm against the LCG in its easiest form as shown in equation (1). The theoretical analysis of Plumstead's algorithm is based on the worst case, but the worst case occurs rarely. Our experimental results show that, unfortunately, the algorithm is much more powerful than what the theoretical analysis has suggested. We first briefly describe Plumstead's algorithm in the following.

Plumstead's algorithm [2]: Assume the LCG is fixed to (1) with a, b, m, and X_0 unknown and no properties of them are assumed, except $m > \mathsf{max}(a, b, X_0)$. The algorithm will find a congruence, $X_{n+1} = \hat{a}X_n + \hat{b} \mod m$, possibly with different multiplier and increment but generating the same sequence as the fixed congruence does. The inference consists of two stages as follows. Let $Y_i = X_{i+1} - X_i$.

Stage I: In this stage, we will find $\hat{a}$ and $\hat{b}$ as follows.

1. Find the least t such that $d = \gcd(Y_0, Y_1, \ldots Y_t)$ and d divides Y_{t+1}.
2. For each i with $0 \le i \le t$, find u_i such that
$$\sum_{i=0}^{t} u_i Y_i = d.$$
3. Set $\hat{a} = \frac{1}{d} \sum_{i=0}^{t} u_i Y_{i+1}$, and $\hat{b} = X_1 - \hat{a}X_0$.

This stage will give $X_{i+1} = \hat{a}X_i + \hat{b} \mod m$ for all $i \ge 0$.

Table 1: Results of Plumstead's algorithm

$\|m\|$ (bytes)	μ	δ	min	max
1	5.438	0.939	5	12
2	5.617	1.221	5	17
4	5.554	1.082	5	15
8	5.586	1.114	5	16
16	5.802	1.764	5	31
32	6.105	3.149	5	57

Stage II: In this stage, we begin predicting X_{i+1} and, if necessary, modifying the modulus m. When a prediction X_i is made, the actual value will be available to the inference algorithm. Initially, set $i = 0$ and $m = \infty$ and assume X_0 and X_1 are available (we can reuse the numbers used in the previous stage). Repeat the following steps forever:

1. Set $i = i + 1$ and predict
$$X_{i+1} = \hat{a}X_i + \hat{b} \mod m.$$
2. If X_{i+1} is incorrect, set $m = \gcd(m, \hat{a}Y_{i-1} - Y_i)$.

It is easy to prove that X_i indeed can be inferred *in the limit*, but the argument for an upper bound of the number of incorrect predictions is highly nontrivial. We shall describe only the main results in the following and omit the detailed proofs that can be found in Plumstead's original paper [2].

Analysis of Plumstead's algorithm: It is clear that every step in both stages are polynomial-time computable in terms of the size of m. In [2] Plumstead proves that t in Stage I is bound by $t \leq \lceil \log_2 m \rceil$. Also, Plumstead argues that the number of incorrect predictions made in Stage II is bounded by $2 + \log_2 m$. Thus, the algorithm is optimal with sample complexity $O(\log_2 m)$ in the worst case.

Empirical results of Plumstead's algorithm: In our experiment we find that the number of samples needed in average is far fewer than in the worst case. We test moduli, m, from 1 byte and double the size up to 32 bytes. The parameters are selected according to our symmetric key scheduling that will be discussed later in Section 3.1. For moduli m bigger than two bytes, we use the Miller-Rabin Test (see [14], page 179) to select prime numbers with an error rate less than $(\frac{1}{2})^{\lceil \log_2 m \rceil}$. For each size of m, we select 1000 different sets of parameters to test.

The results of our experiment are shown in Table 1, where μ is the average number of samples needed to successfully infer the sequence while δ is the standard deviation. Also, the table contains the best case (min) and the worst case (max) for each size. The values of δ in the table indicate that the worst case occurs rarely. In our experiment we take the liberty to notify our inferring program to stop when it reaches the correct point. This notification does not overpower our program because, in our email encryption case, this can be done equivalently by checking the *index of coincidence* of the decrypted text. The indices of coincidence of typical English articles are rather consistent, and therefore the boundary in terms of the index of coincidences between a meaningful text and a random text is clear and can be detected automatically.

Based on the results above, it is clear that the size of m does not significantly prolong the inference process. This is because, from theoretical point of view, the size of m does not affect the number of internal states in the LCG [4, 6]. Thus, our remedy for the LCG is not to increase the size of m but to hide the numbers generated. The experimental results suggest that if we can find a way to prevent the attacker from retrieving five or more consecutive numbers from the sequence, our cipher will remain on the safe side.

3 Email Encryption

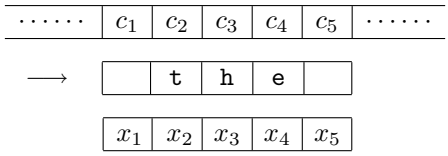
Although the market for email encryption softwares seemed to be promising ever since the Internet came to its age, its research and products were unreasonably few. In part this was because software companies and individuals had been discouraged to develop an email software with encryption ability due to the export control laws of the United States. Moreover, most public key ciphers were patented. For example, the creator of PGP, Phil Zimmermann, had experienced all these non-technique troubles [5]. Until recently the export controls are relaxed and the patents of many public-key ciphers are gradually expired. As a result, a new wave of developing fast, secure, inexpensive, and *legal* email encryption softwares is certainly expected.

We first outline the infrastructure of our system of email encryption in the following subsection.

3.1 Hybrid Cipher

We follow the structure of a hybrid cipher used in PGP. A standard RSA cipher will be used for the symmetric key exchange. The symmetric key is a set of LCG parameters. Our underlying symmetric cipher uses the LCG to generate random noise for scrambling the email message. Two parties, the sender and the receiver, should establish keys for the RSA cipher upon their first communication. We omit details about the RSA and how a hybrid cipher works, which are rather

Figure 1: Landing the word, '`the`'



standard (see [5, 14] for details). We use 512 bits (64 bytes) for the public keys of the RSA in our test cipher, which is big enough for us to encrypt the 4 parameters of our LCG (a 16-byte number for each parameter). In real implementation, we suggest to use 2048-bit RSA for maximum security. Moreover, a bigger key size of the RSA allows us to send bigger LCG parameters and increase the block size.

Key Scheduling: The sender randomly selects new LCG parameters. A fully isolated source of random numbers should be used for the selection. In our case, we need four 16-byte numbers: a, b, m, and X_0. We require the moduli, m, to be a 16-byte prime. We use the Miller-Rabin test to select prime numbers with error rate less than $2^{-\lceil \log_2^m \rceil}$ (see [14], page 179).

According to the Prime Number Theorem, the density of 16-byte primes is about $1/(\ln 2^{128}) = 0.0127$, and hence we can successfully find a prime within about 100 random trials. It is not a problem if we accidentally pick up a pseudo-prime for m since there is no particular reason to require that m must be a prime. Moreover, except some trivial values such as 0 or 2^k, we allow a, b, and X_0 to be any values less than m, since there is no concern about the size of the cycle in the sequence generated. A 128-bit prime as the modulus is very likely to generate un-repeated numbers within the length of a regular email.

4 Using an LCG as a Noise Generator

Our experimental results show that the algorithm can correctly predict the entire sequence with a few known numbers in the sequence. Thus, if we directly embed the numbers in our email text, some classical cryptanalysis such as the *dictionary-attack* can be used to provide enough information for inference. For example, "`the`" occurs in almost every English emails. Moreover, if we keep all delimiters of the plaintext for receiver's pleasure to read the email, a 3-letter word actually contributes 5 characters. Consider the diagram in Figure 1, where the segment $c_1c_2c_3c_4c_5$ in the ciphertext is obtained from " `the` ": After a few trials, " `the` " will be landed at a right position as above. When this happens, each x_i can be recovered by removing the corresponding characters in " `the` " from the cipher characters, c_i's.

According to the results shown in Table 1, it is very likely that five numbers are enough to successfully predict the entire sequence. Even if in some cases the random sequence cannot be broken with five known numbers, the attacker can search a dictionary for a longer word and try to find a match in the email message. As we mentioned earlier, the checking can be done automatically by examining the index of coincidence of the decrypted text. If the LCG is not the correct one, the index of coincidence will indicate that the decrypted text is close to a meaningless random text. With today's computer power, the attacker can search an entire dictionary and finish the computation within a few hours, which is very practical.

Our remedy proposed in [11] for this problem is to distribute a 16-byte random number into 16 text characters. This can be easily extended to 32-byte random numbers for better security. We describe the method as follows.

Encryption Method A: We assume that the plaintext is prepared in extended ASCII code (8-bits). Let

`Telecommunication in the state of the art` (2)

be the message to be encrypted. Thus, the first three characters in (2) are T = 84, e =101, and l = 108. Unlike PGP, we do not compress the plaintext in our test cipher for the simplicity reason.

Our proposed cipher is a block cipher, and each block consists of 16 bytes. Once the parameters of the LCG are determined, we generate $X_1, X_2, X_3, \ldots$ one by one until all characters in the message are encrypted. Starting from X_1, we embed each byte of the random number in a character of the email message. Since there are 41 characters in (2), we need three 16-byte random numbers. The embedding operation is straightforward, which is simply the addition modulo 256. A more complicate operation doesn't seem necessary. Suppose

$$X_1 = \texttt{10 5A FB 11 FC BB 00 11 22 33 44 55 66 77 88 99}_\texttt{h}$$

The values of the first three bytes are $\texttt{10}_\texttt{h} = 16, \texttt{5A}_\texttt{h} = 90$, and $\texttt{FB}_\texttt{h} = 251$. Thus, the values of the first three ciphertext characters encrypted from (2) are:

$$\begin{aligned} 84 + 16 \quad &\bmod 256 \ = 100, \\ 101 + 90 \quad &\bmod 256 \ = 191, \\ 108 + 251 \quad &\bmod 256 \ = 103. \end{aligned}$$

The message will be padded by space to make the length a multiple of 16. For decryption, the receiver should obtain the same parameters of the LCG via

some public-key cipher and be able to reproduce the same sequence. The reverse operation is straightforward. Using the dictionary-attack, a correct guess on a string of 16 characters reveals just one number in the sequence. If the attacker successfully guess 48 characters, which is very unlikely since this requires three consecutive long words to be correctly located in the ciphertext, three numbers generated by the LCG will be revealed. Even under this very unlikely situation, no prediction can be possibly made from three numbers, and hence the encryption method above can resist the dictionary-attack.

Attacks on Method A: It is reasonable to assume that a typical email message contains at least two or three sentences, say 100 characters. Thus, it is clear that the encryption method mentioned above can't resist the known plaintext attack, since a typical plaintext-ciphertext pair is likely to reveal at least 100 bytes, which are enough to provide six consecutive numbers of the secret sequence, and the six numbers has reached the minimum requirement for breaking the sequence. Therefore, in [11] we require that the parameters of the LCG should be changed for each new message. However, this does not completely solve the problem as describe below.

Consider typical business communication or official documents. They usually contain a certain amount of fixed texts for the purpose of formality or greeting. Such kind of texts repeatedly appear in every email from a specific organization. It is easy to collect these texts used by a concerned email sender. Once the attacker has these texts, the attacker can try to match them in the same way as the dictionary-attack and break the system. The database for such attack can be systematically constructed simply using some public archives. With today's computer power and capacity, it is very practical to search and update such database.

To overcome such attacks, we introduce a Noise-Permutation cipher in which the numbers generated by the LCG play an extra role in altering the original order of the characters in the message. The system relies on the fact that the number of permutations grows very fast. We describe the system in the following section.

4.1 A Noise-Permutation Cipher

Encryption Method B: As Method A, we use the LCG to generate numbers one by one until all characters in the message are encrypted. This newly proposed cipher is also a block cipher. Unlike the old version, the block size is 256 characters.

Let the sequence of the random numbers be X_1, X_2, X_3, ... and $\mathbb{Z}_{256} = \{0, 1, \ldots, 255\}$. Also, let m_i be the i^{th} character in the message. We pad the last block with characters $m_0, m_1, m_2, \ldots$ until the block has 256 characters in it. If there are not enough characters, repeat it from m_0. For each block, we need 32 pseudo-random numbers from the sequence. Our encryption includes three steps:

1. The first 16 pseudo-random numbers will be used as in Method A. Since each number has 16 bytes, we have $16 \times 16 = 256$ bytes for the block of 256 characters. Let $c_0, c_1, \ldots, c_{255}$ be the scrambled text of the block after the 16 random numbers are added.

2. The second half of the 32 pseudo-random numbers will be used to create a permutation function Π on $\mathbb{Z}_{256}$ as follows: Let $B_0, B_1, \ldots, B_{255}$ be the 256 bytes of the 16 pseudo-random numbers. The permutation function

$$\Pi = (\pi_0, \pi_1, \ldots, \pi_{255})$$

   is defined as follows:

   (a) $\pi_0 = B_0$, and

   (b) for i with $1 \leq i \leq 255$, $\pi_i = (n \mod 256)$, where n is the least integer such that $n \geq B_i$ and $\pi_i \notin \{\pi_0, \pi_1, \ldots, \pi_{i-1}\}$.

3. Finally, apply Π to the scrambled text and obtain the ciphertext of the block as

$$c_{\pi_0}, c_{\pi_1}, \ldots, c_{\pi_{255}}$$

The decryption method is straightforward once the receiver obtains the same parameters and generates the same random sequence.

4.2 Possible Attacks on Method B

At the moment, we speculate but are not sure that the cipher proposed above can resist the known plaintext attack. Our intuition is that, the difference between ciphertext and plaintext involves two factors: random noise and a random permutation. It doesn't seem easy to separate the two factors from observing the ciphertext, but the attacker needs an exact separation for the obvious reason.

A chosen plaintext attack can reveal the permutation function. For example, the position at which the ciphertext of a^{256} and the ciphertext of $a^i b a^{255-i}$ differ is the value of π_i in Π. However, according to the construction of the permutation function, the mapping from the possible values of the 16 pseudo-random numbers to the permutation functions on $\mathbb{Z}_{256}$ is many-one.

Roughly speaking, every permutation function corresponds to

$$\frac{256^{256}}{256!} \approx \frac{256^{256}}{(\sqrt{2 \times 256 \times \pi}) \times (256/e)^{256}} \approx 10^{109.6}$$

many values for the 16 pseudo-random numbers. Thus, even a permutation function is given, it is not feasible to sort out the values for 16 pseudo-random numbers. More precisely, according to the birthday attack, when

$$k \approx \sqrt{2n \ln(0.5)^{-1}} - 1 \approx 17.84,$$

the probability of $B_k \in \{\pi_0, \pi_1, \ldots, \pi_k\}$ is at least 0.5. In other words, starting from π_{17}, the value of π_i is not likely to be the value of B_i. Recall the minimum amount of numbers requirement for breaking an LCG, which is five. The chance to reveal five 16-byte numbers from a permutation function is the chance to have 80 1-byte integers without collisions, which is

$$\frac{256 \times 255 \times 254 \times \cdots \times 177}{256^{80}} \approx 1.27 \times 10^{-13}.$$

If we double the modulus size to to 32 bytes and keep the block size the same, the chance above will drop to 1.59×10^{-102}. However, we do not think this modification is necessary since revealing the parameters of the LCG in one out of 10^{12} emails from the same sender is not a problem.

Finally, we are also skeptical of the chosen plaintext attack with a subtle differential cryptanalysis that may lead to a crack of our system. Nevertheless, we still conservatively suggest changing the LCG parameters as often as possible.

5 Conclusion and Future Study

Since the LCG is such an efficient and widely known pseudo-random number generator, we don't want to completely remove it from our cryptographic design just because of its theoretical weakness, especially in some cases we can find an easy way to patch this fault. In general, this fault is not formidable if it is possible to block away the numbers from directly exposing to the attacker. Email encryption is a good example to achieve this, where the pseudo-random numbers and the text itself can hide each other. It is worthwhile to find other applications such as encryption on some wireless sensor network, where the communications between sensors are short and frequent, and the computational resource on each sensor is limited.

To conclude this investigation, we recommend that: if simplicity and efficiency are our major concerns and lightweight encryptions can meet our security requirement, the LCG should be reintroduced into our list of choices. What we need to do is to find an efficient way to hide the pseudo-random numbers generated by the LCG.

References

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

[2] Joan B. Plumstead (Boyar). Inferring a sequence generated by a linear congruence. *Proceedings of the 23rd Annual IEEE Symposium on the Foundations of Computer Science*, pages 153–159, 1982.

[3] Karl Entacher. A collection of selected pseudo-random number generators with linear structures. TR 97-1, University of Vienna, Austria, 1997.

[4] Niels Ferguson and Bruce Schneier. *Practical Cryptography*. Wiley Publishing, Inc, Indianapolis, Indiana, 2003.

[5] Simson Garfinkel. *PGP: Pretty Good Privacy*. O'Reilly & Associates, Inc, 1995.

[6] Joel Heinrich. Detecting a bad random number generator. Technical Report:CDF/MEMO/STATISTICS/PUBLIC/6850, University of Pennsylvania, 2004.

[7] John Kelsey, Bruce Schneier, David Wagner, and Chris Hall. Cryptanalytic attacks on pseudorandom number generators, 5^{th} *Fast Software Encryption International Workshop,* 1998.

[8] Donald E. Knuth. *The Art of Computer Programming*, volume 2:Seminumerical Algorithms. Addison-Wesley, 1969.

[9] Donald E. Knuth. Deciphering a linear congruential encryption. *IEEE Transactions on Information Theory*, IT-31(1):49–52, January 1985.

[10] Hugo Krawczyk. How to predict congruential generators. *Journal of Algorithms*, 13(4):527–545, 1992.

[11] Chung-Chih Li, Hema Sagar R. Kandati, and Bo Sun. Security evaluation of email encryption using random noise generated by LCGs. 15^{th} *CCSC:CS*, page (to appear), April 2005.

[12] Stephen K. Park and Keith W. Miller. Random number generators: Good ones are hard to find. *CACM*, 31(10):1192–1201, 1988.

[13] Terry Ritter. The efficient generation of cryptographic confusion sequences. *Cryptologia*, 15(2): 81–139, 1991.

[14] Douglas Stinson. *Cryptography: Theory and Practice*. Chapman & Hall, 2^{nd} edition, 2002.