1 Prime Shuffle: A New Type of Cryptographic Algorithm

Abstract

Linear congruential generators (LCGs) represent one of the oldest and best-known pseudorandom number generator algorithms. Prime Shuffle (PS) creates a bijective transformation vector T to shuffle bits of given data by using a subset of all LCGs builded out of prime numbers. The trick is to create a sequence using LCGs and change the used prime numbers in every step n depending on bytes of the key K which is created by a secret password PW, a random number RN and such many other keys K_i as you like. The algorithm is symmetric and has a variable block length of normally 1 000 000 bits (or more up to 10 000 000), the key K is not mixed with the original or crypted data, K is only part of the algorithm! So neither the key nor the password or hashes of it have to be saved on a permanent medium, they are only needed in RAM. Just the random number and the additional keys K_i have to be stored: The random number together with the data as crypted file, the additional keys anywhere.

Prime Shuffle is easy to implement and has high costs for performance and memory, but this was the design goal to hamper brute force attacs.

History

- 11.02.2007 update by variant submitted to
  Fast Software Encryption 2007, http://fse2007.uni.lu/
- 25.11.2006 update to fit implementation
- 15.03.2006 add more descriptions
- 07.03.2006 use LCG for pseudorandom number generator
- 05.03.2006 first published

Autor

Ralf Peine <ralf.peine@jupiter-programs.de>

Jupiter-Programs

Emil-von-Behring-Str. 16
46397 Bocholt
Germany

2 Description

Look at the following linear congruential generator (LCG, see [Wi]) like described in [Ba]:

$$a_n = (mul * a_{n-1} + add) \;\%\; mod \tag{1}$$

(% is used for modulo division like operator % in C or perl)

add, *mul* and *mod* are given as prime numbers (P_r) between $P_r min$ and $P_r max$ with

$$P_r min = 1\ 000\ 000 \text{ and } P_r max = 10\ 000\ 000 \tag{2}$$

(only bounded by processor architecture (32-bit) and programming language).

Now think about different values for add_n, mul_n and mod_n for every n:

$$a_n = (mul_n * a_{n-1} + add_n) \;\%\; mod_n \tag{3}$$

add_n, mul_n and mod_n can be calculated by a secret password PW, a random number RN and such many other keys K_i as you like with M_c (MaxChars) $\geq$ 1000 bytes.

2.1 Preparation

Longer PW by itself up to M_c chars. Combine PW, RN and K_e $\forall e$ to the key K by bitwise XOR. Fill up the data D up to 128 kByte by random values.

2.2 Basic Algorithm

Now use the integers I_{i_n}, I_{j_n} and I_{l_n} build out of K and add it to the indices i_n, j_n and l_n in the prime number list to get add_n, mul_n and mod_n for the next step.

2.2.1 Startup Values

are given as:

$$\begin{aligned}
i_0 &= 0 && (4)\\
j_0 &= 0 && (5)\\
l_0 &= 0 && (6)\\
k_{i_0} &= -5 && (7)\\
k_{j_0} &= -2 && (8)\\
k_{l_0} &= 1 && (9)\\
a_0 &= (K[0] * 256^3 + K[1] * 256^2 + K[2] * 256 + K[3]) \;\%\; length && (10)\\
a_{\delta_0} &= 0; && (11)
\end{aligned}$$

$K[i]$ means the byte number i of key K, starting with 0 as first index. Name these startup parameters as $\mathcal{I}_{DL_0}$ (Initial Data Loop 1)

2.2.2 Step from $n-1$ to n

$$\begin{aligned} k_{i_n} &= k_{i_{n-1}} + 9 && (12) \\ k_{j_n} &= k_{j_{n-1}} + 9 && (13) \\ k_{l_n} &= k_{l_{n-1}} + 9 && (14) \end{aligned}$$

$$\begin{aligned} I_{i_n} &= K[k_{i_n}] * 256^2 + K[k_{i_n}+1] * 256 + K[k_{i_n}+2] && (15) \\ I_{j_n} &= K[k_{j_n}] * 256^2 + K[k_{j_n}+1] * 256 + K[k_{j_n}+2] && (16) \\ I_{l_n} &= K[k_{l_n}] * 256^2 + K[k_{l_n}+1] * 256 + K[k_{l_n}+2] && (17) \end{aligned}$$

$$\begin{aligned} i_n &= i_{n-1} + I_{i_n} && (18) \\ j_n &= j_{n-1} + I_{j_n} && (19) \\ l_n &= l_{n-1} + I_{l_n} && (20) \end{aligned}$$

$$\begin{aligned} mul_n &= P_r[i_n] && (21) \\ mod_n &= P_r[j_n] && (22) \\ add_n &= P_r[l_n] && (23) \end{aligned}$$

$$a_n = ((mul_n * a_{n-1} + add_n) \,\%\, mod_n) \,\%\, length \qquad (24)$$

$P_r[i]$ means the prime number i of prime number list, starting with 0 as first index. *length* is the length (in bits!) of the data block D to crypt. Maximum for *length* is $P_r max$. If one of the indices k_* or i_* runs out of bounds, just use modulo operator (%) to make them fit into the vector:

$$\begin{aligned} k_{max} &= \text{max-index}(K) && (25) \\ k_* &= k_* \,\%\, (k_{max}+1) \qquad \text{if } k_* \geq k_{max} && (26) \end{aligned}$$

$$\begin{aligned} i_{max} &= \text{max-index}(P_r) && (27) \\ i_* &= i_* \,\%\, (i_{max}+1) \qquad \text{if } i_* \geq i_{max} && (28) \end{aligned}$$

Name the $k_{i_n}, k_{j_n}, k_{l_n}, i_n, j_n, l_n$ for the maximal n and the last calculated a_m (which is not set by a a_{δ_*}) as $\mathcal{I}_{DL_s}$ where s is the number of calls to $P_{sc}(...)$, which will be defined in the following.

2.2.3 Create Transformation Vector T:

Now build the Transformation Vector T and remember all used positions. If a position has been already used, search for one free up starting at $a_{\delta_{m-1}}$. Let the index found name with a_{δ_m}. Use a_{δ_m} instead of a_n to calculate a_{n+1} and increment m by one.

Next create T by:

$$T[a_i] = i - 1 \qquad \forall i \tag{29}$$

If $T[i]$ is initialized by $-1\ \forall i$, it is not needed to store information about already used a_n in an extra vector, because unused values for a_n can be identified as $T[a_n] = -1$.

2.2.4 Shuffle Data D:

Calc a new data array D_c as:

$$\text{Bit}(i) \text{ of } D_c \ = \ \text{Bit}(T[i]) \text{ of } D \qquad \forall i \tag{30}$$

Name the procedure to calc the value of D_c as

$$P_{sc}(\mathcal{I}_{DL_*}, K, D) := D_c \tag{31}$$

2.3 First Shuffle The Key

Create a vector T with length(T) = length(M_c) * 8 to shuffle the bits of the key K. Use the shuffled key K_{s-1} and $\mathcal{I}_{DL_{s-1}}$ to get K_s by

$$K_s = P_{sc}(\mathcal{I}_{DL_{s-1}}, K_{s-1}, K_{s-1}) \tag{32}$$

Repeat this at least 10 times to shuffle the bits of the key.

2.4 Next Shuffle The Data

Use $P_{sc}(\mathcal{I}_{DL_*}, K_s, D_*)$ to shuffle the Data D_0:

$$D_i := P_{sc}(\mathcal{I}_{DL_{s+i-1}}, K_s, D_{i-1}) \qquad \forall\ i \tag{33}$$

Do this at least 2 times.

2.5 Decryption

Just calculate the transformation vector T as described in the former sections. First shuffle the key, then calc $T_{i_{max}}$ for the data, but donot shuffle it, just calc the transformation vector.

$$T_i = P_{st}(\mathcal{I}_{DL_{s+i-1}}, K_s, T_{i-1}) \qquad \forall\ i \tag{34}$$

(named P_{st} to be different from P_{sc}) with

$$T_0[j] = j \qquad \forall\ j \tag{35}$$

Then simply move bit of position b of D_c back to position $T_{i_{max}}[b]$ in D.

2.5.1 Preconditions for Decryption

To be able to do the decryption, you need the following informations:

You have to know the key K, and therefore you need RN (out of the encrypted file), the password PW (user input at runtime) and all additional keys K_i. Also you need to know the *length* (data block length in bits), the prime number list, the number of calls to P_{sc} (encryption rounds) for the key and also the number of calls to P_{sc} for the data D_0. All these parameters except the password may be stored at harddisk and should be exchanged directly (by USB stick, disc, CD, ...). Only RN together with the encrypted data D_c needs to be transmitted by every (unsecure) connection (WWW, email, ...).

3 Safeness

As you can see the algorithm is a symmetric one and full scalable. The key K (and the password PW) is not combined with the data. There is no need to store K or PW, only RN has to be stored in the data, K_i elsewhere.

RN has to be changed for every saving/encryption, so that every encryption even of the same file will become complete different to the former ones, because every time a different key K is used.

3.1 Attacks

3.1.1 Testing The Password Or Key Byte By Byte

is not possible because the key K is shuffled at least 10 times, so that the bits of the first byte of the key are spread over the next (new) key. For every different startup key the bits of any key byte are shuffled to a new order, which gives a different key for the next of the 10 steps.

3.1.2 Abbreviate Decryption Calculation

is not possible, because a_n has to be calculated by a recursive sequence, and so $a_0..a_{n-1}$ have to be known to calc a_n. Also in the case of duplicate values for a_i and a_n, a_n is replaced by a_{δ_m} and so it is not calculated out of a_{n-1}.

3.1.3 Abbreviate Decryption Rounds

is also not possible, because every encryption round uses the $\mathcal{I}_{DL_{s-1}}$ from the previous round as initial values to start the next round s, so different primes will be used for calculation of a_i in every round and it will be different from the a_i of the previous round. Therefore it is not possible to calc one round faster out of the former one, every value has to be calculated as given in (4)-(33).

3.1.4 Recalculation Of The Key If Knowing The Plain Text

is not possible, because the key is not combined with the data and the transformation vector T is not recalculatable out of bits only. Knowing more pairs of plain/cipher texts doesn't help, every encryption uses a key K different to the others because RN is changed every time.

3.1.5 The Effort Of Brute Force Attacks

will be discussed in the following.

3.2 Everything Known But Password

All additional keys K_i are known, the number of key and data encryption rounds and the block length also.

In this case the strength of the encryption is only defined by the length of the password. A password of 20 chars with each of 100 possibilities gives

$$100^{20} = 10^{(2*20)} = 10^{40} \text{ combinations} \tag{36}$$

One day has 60*60*24 = 86400 seconds and one decryption needs 10 seconds at minimum, so at most 10000 decryptions could be done by a standard PC with 4.0 GHz per day, so a PC can calculate in a year

$$\begin{aligned} 10^5 \text{ (combinations per PC and day)}, && (37)\\ 365 \text{ days per year} && (38)\\ = 3.65 * 10^{(2+5)} = 3.65 * 10^7, && (39) \end{aligned}$$

which will be named "computer year" (like "man year").
With 1,000,000,000 PCs calculating one year

$$\begin{aligned} 3.65 * 10^7 \text{ (combinations per PC and year)}, && (40)\\ 10^9 \text{ PCs } (1,000,000,000) && (41)\\ = 3.65 * 10^{(7+9)} = 3.65 * 10^{16}, && (42) \end{aligned}$$

passwords may be checked. So these enormous amount of PCs has to calculate

$$10^{40}/(3.65 * 10^{16}) \tag{43}$$
$$= 10^{24}/3.65 \tag{44}$$
$$> 2.739 * 10^{23}\text{years} \tag{45}$$

for all combinations, that are

$$2.739 * 10^5 * 10^9 * 10^9 \text{ years} \tag{46}$$
$$= 2.739 * 100,000 * 1,000,000,000 * 1,000,000,000 \text{ years.} \tag{47}$$

When the PCs get faster, just do more rounds for encryption or use bigger files to keep the calculation time constant at 10 seconds.

3.3 Transmitted Data Only Known

The additional keys K_i are unknown, the number of key and data encryption rounds and the block length may be known, the transmitted data are known (D_c and RN). So the resulting key K is unknown and therefore the number of combinations to test for keylength 1 kByte are

$$2^{8000} = 1024^{800} \ > \ 10^{800*3} \ = \ 10^{2400} \text{ combinations} \tag{48}$$

AS you can see the effort if knowing the key is multiplied by 10^{2360} , and the additional effort, that would come from the unknown block lengths and the also unknown numbers of key and data encryption rounds isn't taken into account.

So you need at minimum

$$10^{2400}/(3.65 * 10^7) \ > 2.739 * 10^{2392} \quad \text{computer years} \tag{49}$$

to decrypt the data. So it is not needed to vary block length, prime numbers or shuffle rounds to be secure for this case.

3.4 Encryption Of Files Longer Than Block Length

To get faster encryption of following blocks, only the shuffle of the data D (bitwise transformation) can be done, so that the high costs of calculating the transformation vector T can be left out.

3.5 Fast Erasing of Encrypted Data

If a cypher text should be unreadable, it is enough to delete one of its keys K_e. So if every cypher text is created using the base key part K_B, just wipe out this key part from disc/memory, what should require less than a second. The decryption then costs as many effort as if you would know the transmitted data only as described in the former section.

3.6 Minimal Size for Encrypted File

How to calculate the minimal size for a encrypted file with the same strength of encryption for unknown password is an interesting (but open) question. Possibly it is much smaller than 1,000,000 bits.

4 Use Cases

This algorithm is **not** designed to be used in chipcards or to encrypt data streams. It is designed to secure very critical data like logins, passwords, PIN numbers or some text that may be stored in one file up to 10 MBits (1.25 MBytes) like the source forge tool "PasswordSafe" does.

5 Performance and Memory

You need powerful hardware and lots of memory to encrypt and decrypt, but this is the main feature to hamper using brute force methods.

The time needed to encrypt / decrypt will be 10 sec up to 2 min for a standard PC (with processor up to 4.0 GHz).

You need only 1 KB more than the original size to store the random number in the encrypted file. But for good security, the size of the file should not be less than 1 MBit (128 kByte), better are 10 MBit (1.25 MByte).

For an encryption of 1 MBit data memory needed for the encryption SW is betweeen 170 and 180 MByte (Perl at WinXP), 10 MBit needes ca. 700 MByte RAM. The C Implementation will be faster and needs smaller amounts of memory.

Implementations in Perl and Python are already available, C is in preparation.

References

[Ba] Ruedeger Baumann: CHIP Wissen: Programmieren mit PASCAL, 1. Auflage 1980, Vogel-Verlag, Wuerzburg

[Se] Gisbert W. Selke: Kryptographie, 1. Auflage, O'Reilly Verlag, 2000, ISBN 3-89721-155-6

[Wi] Wikipedia: Linear Congruential Generator, 2006/12/10-13:10, http://en.wikipedia.org/wiki/Linear_congruential_generator