Cryptographic Shuffling of Random and Pseudorandom Sequences

Markus Dichtl

Siemens Corporate Technology
Email: Markus.Dichtl@siemens.com

Abstract. This papers studies methods to improve the cryptographic quality of random or pseudorandom sequences by modifying the order of the original sequence. A new algorithm Cryshu is suggested, which produces its shuffled output data at the rate of the input data.

1 Cryptographic Aspects of Shuffling Random and Pseudorandom Sequences

A deck of cards is shuffled to arrange the cards in a random sequence. When we have random or pseudorandom numbers, we may try to rearrange their sequence to make them even "more random".

For streamciphers, the cryptanalyst tries to keep track of the state of the internal machinery which produces the pseudorandom output sequence. The only information he obtains about what is going on internally is from the output data. When the sequence of output data is shuffled, his task becomes more difficult, since subsequent output elements do not correspond to subsequent states of the internal machinery. Hence, it is considerably more difficult to draw conclusions from the output data to the internal state. This effect of making the cryptanalysis harder is only achieved, if it is impossible to determine the original sequence from the shuffled one. So one requirement for cryptographically useful shuffling algorithms is that it must be infeasible to reconstruct the original sequence when given only the shuffled one.

We suggest to use shuffling as a technique to improve the cryptographic strength of stream ciphers. A sequence of pseudorandom numbers is generated by some algorithm, then shufffling is applied to improve the cryptographic quality of the sequence. The algorithm used to generate the original sequence may be such, that succesful cryptanalytic attacks are possible. In this case, shuffling may be sufficient to thwart any attacks. Shuffling may also be useful for streamciphers for which no feasible attacks are known, to have an additional layer of security.

Physical random number generators tend to have correlations between bits generated subsequently. Here shuffling turns out to be helpful, since in order to exploit the correlation, an attacker must know which bits are correlated. For the shuffled sequence, the attacker does not have this information, since the correlated bits go to distant positions which he does not know. However, shuffling does not help against the most common problem of physical random

Dagstuhl Seminar Proceedings 07021
Symmetric Cryptography
http://drops.dagstuhl.de/opus/volltexte/2007/1014

number generators, bias, which means that the probability of a generated bit to be zero is not equal to 0.5 . Since the numbers of zeros and ones in the sequence remains the same when it is shuffled, the bias remains the same.

2 Known Methods to Shuffle Pseudorandom Sequences

Knuth [Knu81] describes two methods to shuffle sequences of pseudorandom numbers, which are called Algorithm M and Algorithm B. Algorithm M is due to MacLaren and Marsaglia, Algorithm B to Bays and Durham.

Algorithm M requires, in addition to the sequence Z1 of pseudo random numbers to be shuffled, another sequence Z2 of pseudo random numbers, which controls the shuffling of Z1. Algorithm M uses an array which is initially filled with the first numbers generated by Z1. When an element of the shuffled sequence resulting from Algorithm M is required, the next element of Z2 is generated and used to determine an index into the array. The entry stored at this position of the array is returned as the result, the entry of the array is replaced with the next element of Z1.

Algorithm M is suited well for cryptographic purposes, its disadvantage is that half of its random input is used up just for the shuffling.

Algorithm B does not require an additional sequence Z2 to control the shuffling, the sequence Z1 to be shuffled also controls its shuffling. This can be called self shuffling. Algorithm B also uses an array which is initially filled with the first elements of Z1. The auxiliary variable Y is initialised with the next element of Z1. When an element of the shuffled sequence is required, Y is used to determine an index into the array. The entry stored at this position of the array is returned as the result, and it is also used as the new value of Y. Then the entry of the array is replaced with the next value of Z1.

Algorithm B is cryptographically weak. Each number generated betrays from which entry of the array the next result will be taken. After a short period of observation, the attacker will know when this entry was changed last; with very little effort the cryptanalysis of Algorithm B is reduced to the cryptanalysis of Z1.

3 The New Algorithm Cryshu

We want to overcome the cryptographic weakness of Algorithm B while keeping its attractive property, that it produces output at the same rate as it reads its input sequence Z1. The new shuffling algorithm is called Cryshu (Crypto Shuffling). The aim of its design was to leak as little information about its internal state as possible.

Like Algorithm B, Cryshu also uses an array to shuffle the sequence Z1 of pseudo random numbers. Initially this array is filled with the first elements of Z1. The auxiliary variable Y is initialised with the next value of Z1. To determine an output element of Cryshu, Y is used to determine an index into the array.

The value stored at this entry of the array is used to determine a second index into the array. The number found at the second index in the array is returned as the result of Cryshu. The entry of the array at this position is replaced by the next value of Z1. This value is also used as another index into the array; the number stored at this entry of the array is the new value of Y.

This may sound somewhat complicated, but the next section, which gives Cryshu code in C, will show that it is not.

4 Example Software Implementation in C

As an example we give the software implementation of Cryshu in C. The array used has a length of 256 and contains bytes which can be used directly as indices into the array. Z1 also generates bytes.

```
static unsigned char a[256];
static unsigned char Y;

unsigned char Z1(void)
{
return rand();
}

void cryshuinit(void)
{
int i;
for(i=0;i<256;i++)
   a[i]=Z1();
Y=Z1();
}

unsigned char cryshu(void)
{
unsigned char x,res;
x=a[Y];
res=a[x];
a[x]=Z1();
Y=a[a[x]];
return res;
}
```

The function *cryshuinit()* is used to initialise the algorithm. Each call of *cryshu()* returns a byte of the shuffled sequence. As an example, the function *rand()* is used for Z1.

5 Implementation Considerations

Since Cryshu requires an array of considerable size, it is not suited too well for hardware implementation. This is true for shuffling in general, since it is necessarily connected with the storage of the elements to be shuffled.

But Cryshu is suited very well for software implementation. Care should be taken that the array used fits into the primary cache of the computer. It can run at considerable speed, since the computation of one output element requires only 4 table lookups. There are tradeoffs between the array size used, the bit length of the numbers shuffled, the data rate achieved, and the security. For example, the 8 bit words used in the example implementation will not give optimal data rates on 32 bit processors. However, it could be unwise to use 16 bit words, and to keep the length of the array at 256.

On an SNI Scenic Pro M6 with a 200 MHz Intel Pentium Pro processor, the program given in the previous section compiled with Watcom C 11.0 shuffled at a rate of 2.13 MBytes per second. This is only the time for shuffling, it does not include the time used for *rand()*.

6 Open Questions

Cryshu leads to many new questions: all broken streamciphers can be reexamined whether they can also be broken when the sequences they produce are shuffled by Cryshu. The most important candidates to consider are linear congruential generators, and the numerous broken variants and modifications of linear feedback shift registers.

References

[Knu81] D. E. Knuth, The Art of Computer Programming, Vol. 2, Seminumerical Algorithms, 2nd Edition, Addison-Wesley, Reading, Mass., 1981.