Provably Secure Pseudo-Random Generators

A Literary Study

Josefin Agerblad and Martin Andersen

Degree Project in Computer Science, First Level (DD143X)
School of Computer Science and Communication
The Royal Institute of Technology

Supervisor: Douglas Wikström

Abstract

This report is a literary study on provably secure pseudo-random generators. In the report we explain what provably secure pseudo-random generators are and what they are most commonly used for. We also discuss one-way functions which are closely related to our subject. Furthermore, two well-known generators are described and compared, one generator by Blum and Micali, and one by Blum, Blum and Shub. What we have concluded is that the x^2 mod N generator by Blum, Blum and Shub seems to be the better one concerning speed, security and application areas. You will also be able to read about how the Blum-Blum-Shub generator can be implemented and why we believe that implementation is suitable.

Sammanfattning

Den här rapporten är en litteraturstudie om bevisbart säkra pseudo-slumpmässiga generatorer. I rapporten förklarar vi vad bevisbart säkra pseudo-slumpmässiga generatorer är och vad de vanligtvis används till. Vi tar dessutom upp envägsfunktioner som är starkt kopplat till vårt ämne. Vidare beskrivs och jämförs två kända sådana generatorer, en generator av Blum och Micali och en generator av Blum, Blum och Shub. Den slutsats som vi kommit fram till är att x^2 mod N generatorn av Blum, Blum och Shub verkar vara den bättre utav dem vad gäller hastighet, säkerhet och applikationsområden. Ni kommer även kunna läsa om hur en sådan generator kan implementeras och vi förklarar varför den presenterade koden är bra.

Table of Contents

1.0 Introduction

This report is a literary study on the subject of provably secure pseudo-random generators, a tool primarily used in cryptography. The report is comprised of information on what provably secure pseudo-random generators are, how they work and what they are used for, and is the result of a couple of months research on these topics.
This report is a Bachelor’s Thesis in Computer Science, written as part of the course DD143X.

1.1 Problem Statement

Our intent with this report was to get a good basic knowledge of provably secure pseudo-random generators, and also to look at existing algorithms and analyze how they work and how they differ from each other. Furthermore, we wanted to find and describe a suitable method that could be used to implement a provably secure pseudo random generator.

1.2 Aim

Because of our previous limited knowledge of provably secure pseudo-random generators, the report’s main aim was for us to gain a deeper understanding of the subject. Naturally, sharing this newfound knowledge has also been the purpose of this report.

1.3 Definitions and explanations of technical terms

Term	Definition
Pseudo-random sequence	Sequence that appears to be random and passes all efficient statistical tests
True random sequence	Actual random sequence
Plaintext	Unencrypted data which one wants to send; used as input in an encryption algorithm
Ciphertext	Output from an encryption algorithm; encrypted data

2.0 Background

2.1 Background on provably secure pseudo-random generators

2.1.1 What are provably secure pseudo-random generators?

A pseudo-random generator is an algorithm commonly used in cryptography. Given a short random series of numbers, it calculates and returns a longer sequence of numbers. These numbers will appear to be random to any statistical test that compares them to a sequence of true random numbers. If the statistical test cannot distinguish these sequences from each other, the generated series of numbers is considered to be random. If a pseudo-random generator can produce a sequence which no statistical test can distinguish from a series of true random numbers, it is said to be a cryptographically secure pseudo-random generator[1].

To be able to define what a provably secure pseudo-random generator is, we must first define what provable security is. Provable security in cryptography means that cracking a certain cryptographic algorithm or scheme is equally hard as solving a specific well-known problem that is said to be difficult. This definition has faced some criticism because it is rather vague; it is for example hard to know if the chosen problem is "difficult" enough[2]. But for a pseudo-random generator to be provably secure, it must be equally hard to crack the algorithm of the generator as it is to solve a well-known and supposedly difficult problem[3].

Essentially, the term provably secure pseudo-random generator means: An algorithm that is equally hard to crack as a well-known, difficult problem and that converts a shorter random sequence of numbers into a longer one, which in specific aspects does not differ from a sequence of true random numbers[4].

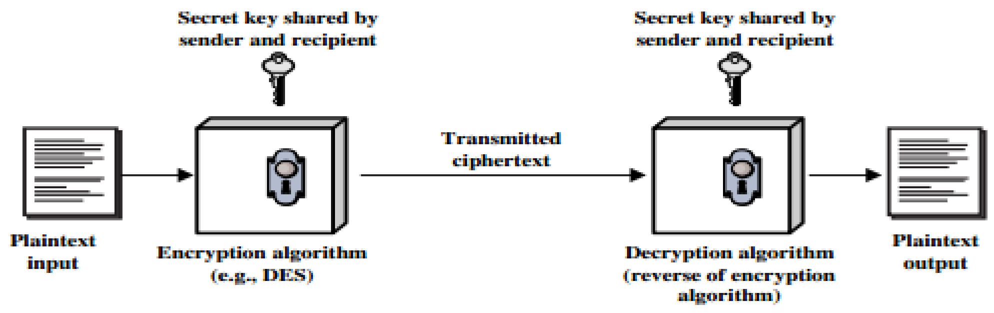
2.1.2 What are provably secure pseudo-random generators mainly used for?

Provably secure pseudo-random generators can be used for many different things. In this section, three of the main applications of provably secure pseudo-random generators are described.

2.1.2.1 Generating keys

One of the biggest fields of cryptography is the work on encryption. This section will explain how encryption works, how to ensure secrecy, and why one could benefit from using provably secure pseudo-random generators to assist with that.

[1] Sidorenko, A. p.4
[2] Lipton, R.J
[3] Sidorenko, A. p.6
[4] Sidorenko, A. pp.4 and 6



Secret key shared by sender and recipient
Secret key shared by sender and recipient
Transmitted ciphertext
Plaintext input
Encryption algorithm (e.g., DES)
Decryption algorithm (reverse of encryption algorithm)
Plaintext output

Figure 2.1[5]

Imagine Alice wanting to send a message, x, to Bob, but an eavesdropper prevents her from sending it as plaintext. To prevent the eavesdropper from being able to read the message, Alice will need to encrypt it into a ciphertext using an encryption algorithm E, before sending it. Bob will then use a decryption algorithm D to decrypt the message. For this to work, however, Alice and Bob will need a shared secret, a key, which they will use to encrypt and decrypt the message, see Figure 2.1. In conclusion, the algorithms will have the following relationship[6]:

$$D_k\left(E_k(x)\right) = x$$

A difficult task is to ensure that the encryption is secure enough. The goal is to reach perfect secrecy, which is attained if absolutely no information on the plaintext can be extracted from the ciphertext without the key. One of the few ways to reach perfect secrecy is to use one-time pads, an encryption technique that, if used correctly, is impossible to break. There are, however, a few, big disadvantages to using one-time pads[5].
Firstly, a one-time pad can only be used once, and trying to reuse them can have devastating consequences. As a matter of fact, in the 1940s, the KGB reused parts of their one-time pads and because of that the U.S. managed to decrypt some of their secret messages[7].
Another problem with one-time pads is that they require a key length that is as long as the message to be encrypted, which is not very practical as the message gets longer, nor is it secure to exchange a key of that size.
Thus, one must settle for an encryption technique which does not provide perfect secrecy, but uses smaller keys and is safe against eavesdroppers that run in polynomial-time[8].

Using a one-way function, a subject that will be described more in-depth later in this report, an encryption scheme which uses a key of length n for a message of length n^c (for some constant c), can be constructed. By then using a provably secure pseudo-random generator to stretch the key into a longer key, the encryption can be used as a one-time pad, not providing

[5] Krishnan, K.
[6] Arora, S. and Barak, B. p.153
[7] Mertens, S. and Moore, C. p.542
[8] Arora, S. and Barak, B. p.154

perfect secrecy, but still being safe against eavesdroppers that run in polynomial-time, and without the disadvantages of a one-time pad[9].

2.1.2.2 Stream ciphers

Provably secure pseudo-random generators can be used to generate a “keystream” for a stream cipher, and thereby serve as a “keystream generator”. To understand how this works, it is necessary to explain how a stream cipher works.

In a stream cipher, a number of bits from a plain text is combined with a number of bits from a keystream, a sequence of random or pseudo-random numbers, to encrypt the plain text-message into a ciphertext. This is usually done by an exclusive-or ($\oplus$) operation between the i:th bit of the plain text and the i:th bit in the keystream like this[10]:

Numbers of bits from a plain text: $p_1, p_2, \dots, p_k$
Numbers of bits from a keystream: $k_1, k_2, \dots, k_k$
Numbers of bits in the ciphertext: $c_1, c_2, \dots, c_k$

$$c_i = \sum_{i=1}^{n} (p_i \oplus k_i)$$

This will require that the keystream has equally many bits as the number of bits in the plain text, which can be impractical and it would be easier to be able to use a shorter keystream. This is where the provably secure pseudo-random generators come in, because it can be used as a keystream generator, as mentioned above. A shorter sequence of numbers can be given to the provably secure pseudo-random generator, and it will return the longer keystream required for the stream cipher[11].

2.1.2.3 Symmetric key cryptosystems

Symmetric key cryptosystems are used in encryption, this section will briefly describe how they work and how provably secure pseudo-random generators are used for symmetric key cryptosystems.

As mentioned above in the section on generating keys, the sender and the receiver require a key for the encryption and decryption of the message that the sender wants to send. The symmetric key cryptosystems differs from the asymmetric ones, by only requiring one secret key which has to be secretly shared between the sender and the receiver of the message in

[9] Arora, S. and Barak, B. pp.155-157
[10] Robshaw, M.J.B. p.2
[11] Pott, Kumar, Helleseth, Jungnickel. p.21

advance[12]. Asymmetric key systems, on the other hand, require the sender and the receiver to have one pair of keys each, one public and one private key[13].

Provably secure pseudo-random generators are often used in the encryption algorithm of the symmetric key cryptosystems. The encryption algorithm can also be based on a stream cipher, where provably secure pseudo-random generators can be used for shortening keystreams. This is discussed in the section on stream ciphers above[14].

2.2 One-way functions

The subject of one-way functions is an important one when discussing cryptography, and is especially significant to pseudo-random generators, which is why the subject is included in this report. This section will explain what one-way functions are and how they relate to provably secure pseudo-random generators.

A one-way function is a function that, for every input, can be easily computed, but is hard, or even impossible, to invert for any algorithm that runs in polynomial time. Furthermore, the function also has to be hard to invert by the average case and not by the worst case, which is highly unusual in cryptography[15].
However, there is, at this time, no actual proof that one-way functions even exist; their existence is in fact one of the famous unsolved problems of computer science. Nevertheless, they are widely believed to exist, and if they do, it would imply that a number of other assumptions in cryptography are accurate as well, among them the assumption that cryptographically secure pseudo-random generators exist[15].

The existence of one-way functions is also closely related to another big mystery in computer science; whether or not the complexity classes P and NP are equal to each other. In the complexity class P are all problems that can be solved in polynomial time. Then there are those problems that cannot be solved in polynomial time, but given an answer to the problem, that answer can be verified in polynomial time; those problems are in the complexity class NP[16]. The P versus NP problem is often discussed in the world of computer science, and as one might deduce from the facts given above, that the existence of one-way functions would imply that $P \neq NP$[17].

There are a few well-known functions that are believed to be one-way functions; that is, no one has yet been able to find a polynomial time algorithm that could invert these functions.

[12] Krishnan, K. p.22-24-1
[13] Roeder, T.
[14] Buttyán, L. p.4
[15] Katz, J. and Lindell, Y. p.182
[16] Kleinberg, J. and Tardos, É. pp.451-452
[17] Arora, S. and Barak, B. p.155

The integer factorization problem is one of them and is based on simple multiplication. The function takes two integers, x and y, as input and outputs the multiplication xy. Even with large numbers, this is a fairly easy calculation; however, inverting the function would mean having to find the prime factors of a large number, a feat no polynomial time algorithm has managed to do[18].

Another example of a possible one-way function is the discrete logarithm problem[19], which is utilized by the Blum-Micali generator. It will therefore be described more in-depth later in this report in the section concerning that generator.

2.2.1 Pseudo-random generator theorem

As mentioned earlier in this report, the existence of one-way functions would imply the existence of pseudo-random generators. This has been established through a few different proofs that together are called the pseudo-random generator theorem. The actual proofs are quite advanced and have therefore been excluded from this report.

2.3 Well-known provably secure pseudo-random generators

There are a few well-known provably secure pseudo-random generators; in this report the perhaps most famous ones, Blum-Blum-Shub and Blum-Micali, are described.

2.3.1 Blum-Blum-Shub

The provably secure pseudo-random generator proposed by Lenore Blum, Manuel Blum and Michael Shub is the x^2 mod N- generator. This section will describe how this generator works and also explain why it is provably secure.

The x^2 mod N-generator works as follows:

Let $N = P \times Q$, where P and Q are two primes. An integer x_0 is then chosen in the range $1 \leq x \leq N$. The integer x_0, which will be the seed for the generator, has to be chosen so that $gcd(x_0, N) = 1$. In other words, the greatest common divider of N and x is 1.
A sequence of numbers is then calculated by the formula $x_{i+1} = x_i^2 \ mod \ N$, and a sequence of bits (b_0, b_1, …) is calculated by the formula $b_i = parity(x_i)$. In other words, b_i will be 1 if x_i is an odd number and 0 if x_i is an even number. It is this sequence of bits that will be the output of the provably secure pseudo-random generator x^2 mod N[20].

The reason to why this pseudo-random generator is provably secure is that it can be proven to be equally hard to crack as the quadratic residuosity problem. This problem is defined in the following way:

[18] Katz, J. and Lindell, Y. p.185
[19] Katz, J. and Lindell, Y. p.186
[20] Blum, L., Blum, M., and Shub, M. p.63

Let $N = P \times Q$, where P and Q are two primes. The group $(Z_N)^*$ are then defined as the group of elements x in the range $1 \leq x \leq N$, where x has to have the property of $gcd(x, N) = 1$. This subgroup are then divided into two subgroups, $(Z_N)^*(+1)$ and $(Z_N)^*(-1)$. The quadratic residuosity problem consists of deciding whether the element x, $x \in (Z_N)^*(+1)$, is a quadratic residue or not. To be a quadratic residue, x has to have the property $y^2 \equiv x \; mod \; N$, for an integer y. To calculate this, an algorithm takes N and x as parameters, and returns 1 if x is a quadratic residue or 0 if it is not[21].

It can essentially be shown that the x^2 mod N-generator is a provably secure pseudo-random generator with the well-known supposedly difficult quadratic residuosity problem in its core. The proof of this can be read in the paper by Blum, Blum and Shub, *Comparison of two pseudo-random number generators*[22].

Because of the way the x^2 mod N-generator works, it is well suited for public-key encryption, where the value of N could serve as the public key and the values of P and Q could serve as the private keys of the receiver[23].

2.3.2 Blum-Micali

The Blum-Micali generator is a provably secure pseudo-random number generator and was created by Manuel Blum, who was also involved in the implementation of the Blum-Blum-Shub generator, and Silvio Micali. As mentioned earlier in this report, the generator uses a possible one-way function called the discrete logarithm problem, and its security is based on the difficulty of solving this problem. The discrete logarithm problem and how it is used by the Blum-Micali generator is described below.

Let p be a prime, then the set $[1, p-1]$is a cyclic group, $(Z_p)^*$, under multiplication modulo p. Let a be an element of $(Z_p)^*$, g the generator of the group and let k be any integer, this yields the function[24]:

$$g^k \equiv a \; mod \; p$$

If k is known then computing a can be done easily, this computation is called discrete exponentiation. However, inverting the function, namely, trying to compute k if a is known, is a far more difficult task, and this is what is known as the discrete logarithm problem[25].

[21] Blum, L., Blum, M. and Shub, M. p.63
[22] Blum, L., Blum, M. and Shub, M. pp.69-73
[23] Blum, L., Blum, M. and Shub, M. p.76
[24] Blum, M. and Micali, S. p.114
[25] Tang, Q. p.14

The Blum-Micali generator is based on the difficulty of computing the discrete logarithm. Let x_0 be a seed, the function $x_i = g^{x_i} \, mod \, p$ will output a sequence $(x_1, x_2, \ldots, x_k)$[26], which in turn will determine the final pseudo-random sequence in the following way[27]:

$$y_i = \begin{cases} 1, & x_i < \frac{(p-1)}{2} \\ 0, & otherwise \end{cases}$$

In the end, a pseudo-random sequence $Y = y_1, y_2, \ldots, y_k$ of k bits will have been generated. Even if one knew the previous bits, one would not be able to calculate the next bit, because of the difficulty of computing the discrete logarithm.

The Blum-Micali generator can be used in private-key cryptography, where it has the property to simulate a one-time pad[28].

2.4 How to implement a provably secure pseudo-random generator

This section will give a brief description of what to think about when implementing a provably secure pseudo-random generator. It will also give an example of how the Blum-Blum-Shub generator is implemented.

Generating a sequence of pseudo-random numbers can easily be done. A commonly used model based on the linear recurrence is:

$$x_{i+1} = ax_i + b \, mod \, n$$

Although a pseudo-random sequence produced by this model would pass many statistical tests, it is not cryptographically strong and would not be suited for use in cryptographic situations[29].
A cryptographically strong pseudo-random sequence must pass all polynomial time statistical tests; this means that no polynomial time algorithm should be able to tell the difference between the generated pseudo-random sequence and a true random sequence, with high probability. It must also pass the next-bit test, which means that even if the first k bits in the pseudo-random sequence are known, no polynomial time algorithm should be able to compute the next bit in the sequence, also with high probability[30]. To fulfill these requirements the generator's security must be based on the difficulty of computing certain functions, as for example the quadratic residuosity problem or the discrete logarithm problem. These have been discussed in detail in the sections on the Blum-Blum-Shub generator and Blum-Micali generator respectively. A more detailed example on how the Blum-Blum-Shub generator is implemented is given below.

[26] Tang, Q. p.15
[27] Blum, M. and Micali, S. p.115
[28] Blum, M. and Micali, S p.853
[29] Blum, M. and Micali, S. p.113
[30] Tang, Q. p.9

To implement the x^2 mod N-generator by Blum, Blum and Shub, the following will be needed (in Java code)[31]:

A function, getPrime(), that returns a random prime-number will be needed for calculating the value $N = P \times Q$. This function will create a bigInteger representing the random prime-number, given an integer for the bit-length and a random object.

A function, generateN(), for generating the value of N, which calls the getPrime() function twice to get P and Q, and then calculates and returns the value N as $N = P \times Q$.

Four different constructors. One for specifying the bits, one for generating the prime and seed, one for specifying the value of N and one for specifying both the value of N and the seed.

A function, setSeed(), for setting the seed x_0 for the generator.

A function, randomBits(), for calculating the random bits according to the formula specified in the section above about well-known provably secure pseudo-random generators. The function will take a number of bits as a parameter and return a sequence of random bits as a result of bit shifting to check whether the value of $x_{i+1} \equiv x_i^2 \; mod \; N$ is even or odd.

Lastly, the main()-function will be needed to call all the different methods used.

Nick Galbreath's implementation, in Java code, of the generator described can be viewed in Appendix A.

[31] Galbreath, N.

3.0 Method

This report is a literary study on provably secure pseudo-random generators and because of that, our main method for the report has been to gather information on the subject by finding approriate sources in the form of scientific articles and books.

Even though the subject of provably secure pseudo-random generators is quite specific and narrow, we have been able to find quite a few scientific papers and books with reliable authors that have been very helpful in our work.

We believe that our choice of method has suited us very well. Owing to the use of this method, we have successfully been able to accomplish our goals and aims of this literary study and we are satisfied with the results.

4.0 Discussion

4.1 Comparison of two provably secure pseudo-random generators

In this section we will discuss the differences between the two provably secure pseudo-random generators that have been discussed in this report, the Blum-Micali generator and the Blum-Blum-Shub generator.

The main difference between the two generators is that they have two different problems in their core, the Blum-Micali generator has the discrete logarithm problem and the Blum-Blum-Shub generator has the quadratic residuosity problem. This causes the generators to behave a little bit differently, and it would therefore be interesting to discuss which of them that is better suited when it comes to different aspects; including speed, security and applications.

4.1.1 Applications

As mentioned earlier in the report, the x^2 mod N-generator can be used for public-key encryption, and the Blum-Micali generator is more suited for private-key encryption, where it can be used to simulate one-time pads. We want to discuss witch one of these to that has the most advantages when it comes to applications in the field of cryptography.

One-time pads has the big advantage that they offer perfect security, but there are some major disadvantages with it that makes it a little bit unattractive when it comes to efficient cryptography, and public-key cryptosystems are more used because of that. Because of this, our assumption is that the x^2 mod N-generator by Blum, Blum and Shub has more advantages then the Blum-Micali generator when it comes to applications in cryptography.

4.1.2 Speed

The main thing that can be said about the speed of provably secure pseudo-random generators is that they are not very fast. This is natural because of how complex they have to be to be able to pass all statistical tests that check whether or not the generated sequence of numbers are distinguishable from a sequence of truly random numbers, as well as being as hard to crack as a known supposedly difficult problem. Another interesting aspect to discuss is which of the two provably secure pseudo-random generators that has been presented in this report, the Micali-Blum generator and the Blum-Blum-Shub generator, is the fastest.

This can be hard to determine since it depends on the speed of their respective hard problem that they are based on, the quadratic residue problem and the discrete logarithm problem. To be able to determine whether an element x is a quadratic residue or not, we have to determine if it satisfies the formula $y^2 \equiv x \ mod \ N$ for any integer y. Therefore, a basic algorithm would require a running time equally long as the size of N, where N is the size of the group $(Z_N)^*$.

For the discrete logarithm problem, on the other hand, the problem lies in trying to calculate the value of the integer k in the formula $g^k \equiv a \; mod \; p$, if a is known. A basic algorithm would, in this case, try to calculate the value by iterating over the size of p, which naturally would require a running time equal to the size of p, which is the size of the cyclic group $(Z_p)^*$.

As mentioned earlier, it is really hard to determine which of the generators that is the fastest, and by trying to determine it by analyzing their respective hard problem in their core, we can do no better than to say that they are equally fast.

4.1.3 Security

Determining the difference in security between the two provably secure pseudo-random generators is difficult as well, since they both contain two difficult problems, and the level of difficulty to solve these problems is connected to the speed of the algorithms that solves the problems. Thus, based on the analysis of the problems in the previous section on speed, we will say that they are equally secure, when it comes to breaking the generators by solving their respective problems.

4.2 Reasons to why the proposed Blum-Blum-Shub implementation is good

In the previous section of this report, on how to implement a provably secure pseudo-random generator, we presented Nick Galbreath's implementation of the algorithm for the x^2 mod N-generator by Blum, Blum and Shub. The reason to why we think this implementation is good is that it is an elegant piece of code, it only requires a few lines of code and it is quite easy to understand. Furthermore, the Blum-Blum-Shub generator is in general a good provably secure pseudo-random generator that provides good security and can be used for many applications.

4.3 The existence of one-way functions and the P versus NP problem

The very existence of the subject of this report, provably secure pseudo-random generators, is entirely dependent on the existence of one-way functions. In turn, the existence of one-way functions would imply that the complexity classes P and NP are not equal. This leads us to the discussions on whether or not one-way functions do exist and the P versus NP problem.

At this time, one-way functions are widely believed to exist and as a result it is also commonly believed that $P \neq NP$. On one hand, this seems reasonable since these assumptions have been discussed for quite some time and no one has yet been able to disprove them. On the other hand, there will always be people with new thoughts and ideas, so there is always the possibility that someone will one day be able to disprove them.

Presently, there are many applications in cryptography that are based on the assumptions that one-way functions exist and that $P \neq NP$. As of yet, this has not caused any problems, however, it is important to ask oneself what would happen if these assumptions turned out to be wrong.

5.0 Conclusion

In the discussion we talked about the differences between the Blum-Micali generator and the generator proposed by Blum, Blum and Shub. The conclusion that can be drawn from the discussion is that the Blum-Blum-Shub generator seems to be the generator that has the most applications in cryptography, and that no big differences can be found in the speed and security between the two. Therefore, it seems that the best generator when it comes to these three aspects is the x^2 mod N-generator by Blum, Blum and Shub.

The famous P versus NP problem and whether or not one-way functions exist were also discussed. It would be the height of hubris to claim that we know the answer to these much discussed questions; however, since many cryptographic applications that are commonly used in today’s society are based on the assumptions made concerning these questions one could conclude that those applications would certainly suffer if the assumptions turned out to be wrong.

6.0 References

Arora, S. and Barak, B. (2009) *Computational Complexity: A Modern Approach*, Cambridge University Press

Blum, L., Blum, M. and Shub, M. (1998) *Comparison of two pseudo-random number generators*

Blum, M. and Micali, S. (1984) *How To Generate Cryptographically Strong Sequences Of Pseudo Random Bits*

Buttyán, L. *Symmetric key cryptography*, Budapest University of Technology and Economics: Department of Networked Systems and Services

Galbreath, N. (2005) *BlumBlumShub.java* [Online], Available: http://javarng.googlecode.com/svn/trunk/com/modp/random/BlumBlumShub.java [24 Mar 2013]

Katz, J. and Lindell, Y. (2007) *Introduction to Modern Cryptography*, CRC Press

Kleinberg, J. and Tardos, É. (2006) *Algorithm Design*, Pearson Education

Krishnan, K. *Computer Networks and Computer Security*, North Carolina State University: Department of Mathematics

Lipton, R.J. (2010-03-13) *Breaking Provably Secure Systems*, Gödel's Lost Letter and P=NP, [Online], Available: http://rjlipton.wordpress.com/2010/03/13/breaking-provably-secure-systems/ [24 Mar 2013]

Mertens, S. and Moore, C. (2011) *The Nature of Computation*, Oxford University Press

Pott, Kumar, Helleseth, Jungnickel (1998) *Different sets, Sequences and their Correlation Properties*

Robshaw, M.J.B. (1995) *Stream ciphers*, RSA Laboratories, Redwood City

Roeder, T. *Asymmetric-Key Cryptography*, Cornell University: Computer Science Research

Sidorenko, A. (2008) *Design and Analysis of Provably Secure Pseudorandom Generators*, VDM Verlag Dr. Muller Aktiengesellschaft & Co. KG

Tang, Q. *Pseudo-random Number Generation*, Carleton University: School of Computer Science

Appendix A – Java code for the Blum-Blum-Shub generator

```
/*
* Copyright 2005, Nick Galbreath -- nickg [at] modp [dot] com
* All rights reserved.
*
* Redistribution and use in source and binary forms, with or without
* modification, are permitted provided that the following conditions
are
* met:
*
*   Redistributions of source code must retain the above copyright
*   notice, this list of conditions and the following disclaimer.
*
*   Redistributions in binary form must reproduce the above
copyright
*   notice, this list of conditions and the following disclaimer in
the
*   documentation and/or other materials provided with the
distribution.
*
*   Neither the name of the modp.com nor the names of its
*   contributors may be used to endorse or promote products derived
from
*   this software without specific prior written permission.
*
* THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND
CONTRIBUTORS
* "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
* LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
FOR
* A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
COPYRIGHT
* OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,
INCIDENTAL,
* SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
* LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF
USE,
* DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON
ANY
* THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR
TORT
* (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE
USE
* OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH
DAMAGE.
*
* This is the standard "new" BSD license:
* http://www.opensource.org/licenses/bsd-license.php
*/

package com.modp.random;
import java.util.Random;
import java.security.SecureRandom;
import java.math.BigInteger;
```

```
/**
* The Blum-Blum-Shub random number generator.
*
* <p>
* The Blum-Blum-Shub is a "cryptographically secure" random number
* generator.  It has been proven that predicting the ouput
* is equivalent to factoring <i>n</i>, a large integer generated
* from two prime numbers.
* </p>
*
* <p>
* The Algorithm:
* </p>
* <ol>
* <li>
*  (setup) generate two secret prime numbers <i>p</i>, <i>q</i> such
that
*   <i>p</i> &ne; <i>q</i>, <i>p</i> &equiv; 3 mod 4, <i>q</i>
&equiv; 3 mod 4.
* </li>
* <li> (setup) compute <i>n</i> = <i>pq</i>.  <i>n</i> can be re-
used, but
*   <i>p</i>, and <i>q</i> are secret and should be disposed
of.</li>
* <li> Generate a (secure) random seed <i>s</i> in the range [1,
<i>n</i> -1]
*   such that gcd(<i>s</i>, <i>n</i>) = 1.
* <li> Compute <i>x</i> = <i>s</i><sup>2</sup> mod <i>n</i></li>
* <li> Compute a single random bit with:
*   <ol>
*   <li> <i>x</i> = <i>x</i><sup>2</sup> mod <i>n</i></li>
*   <li> return Least-Significant-Bit(<i>x</i>) (i.e. <i>x</i> &
1)</li>
*   </ol>
* Repeat as necessary.
* </li>
* </ol>
*
* <p>
* The code originally appeared in <a
href="http://modp.com/cida/"><i>Cryptography for
* Internet and Database Applications </i>, Chapter 4, pages 174-
177</a>
* </p>
* <p>
* More details are in  the <a
href="http://www.cacr.math.uwaterloo.ca/hac/"><i>Handbook of Applied
Cryptography</i></a>,
* <a
href="http://www.cacr.math.uwaterloo.ca/hac/about/chap5.pdf">Section
5.5.2</a>
* </p>
*
* @author Nick Galbreath -- nickg [at] modp [dot] com
* @version 3 -- 06-Jul-2005
```

```
*
*/
public class BlumBlumShub implements RandomGenerator {

   // pre-compute a few values
   private static final BigInteger two = BigInteger.valueOf(2L);

   private static final BigInteger three = BigInteger.valueOf(3L);

   private static final BigInteger four = BigInteger.valueOf(4L);

   /**
    * main parameter
    */
   private BigInteger n;

   private BigInteger state;

   /**
    * Generate appropriate prime number for use in Blum-Blum-Shub.
    *
    * This generates the appropriate primes (p = 3 mod 4) needed to
compute the
    * "n-value" for Blum-Blum-Shub.
    *
    * @param bits Number of bits in prime
    * @param rand A source of randomness
    */
   private static BigInteger getPrime(int bits, Random rand) {
          BigInteger p;
          while (true) {
             p = new BigInteger(bits, 100, rand);
             if (p.mod(four).equals(three))
                    break;
          }
          return p;
   }

   /**
    * This generates the "n value" -- the multiplication of two
equally sized
    * random prime numbers -- for use in the Blum-Blum-Shub
algorithm.
    *
    * @param bits
    *            The number of bits of security
    * @param rand
    *            A random instance to aid in generating primes
    * @return A BigInteger, the <i>n</i>.
    */
   public static BigInteger generateN(int bits, Random rand) {
          BigInteger p = getPrime(bits/2, rand);
          BigInteger q = getPrime(bits/2, rand);

          // make sure p != q (almost always true, but just in case,
check)
```

```
        while (p.equals(q)) {
           q = getPrime(bits, rand);
        }
        return p.multiply(q);
   }

   /**
    * Constructor, specifing bits for <i>n</i>
    *
    * @param bits number of bits
    */
   public BlumBlumShub(int bits) {
        this(bits, new Random());
   }

   /**
    * Constructor, generates prime and seed
    *
    * @param bits
    * @param rand
    */
   public BlumBlumShub(int bits, Random rand) {
        this(generateN(bits, rand));
   }

   /**
    * A constructor to specify the "n-value" to the Blum-Blum-Shub
algorithm.
    * The inital seed is computed using Java's internal "true"
random number
    * generator.
    *
    * @param n
    *            The n-value.
    */
   public BlumBlumShub(BigInteger n) {
        this(n, SecureRandom.getSeed(n.bitLength() / 8));
   }

   /**
    * A constructor to specify both the n-value and the seed to the
    * Blum-Blum-Shub algorithm.
    *
    * @param n
    *            The n-value using a BigInteger
    * @param seed
    *            The seed value using a byte[] array.
    */
   public BlumBlumShub(BigInteger n, byte[] seed) {
        this.n = n;
        setSeed(seed);
   }

   /**
    * Sets or resets the seed value and internal state
    *
```

```
     * @param seedBytes
     *            The new seed.
     */
    public void setSeed(byte[] seedBytes) {
           // ADD: use hardwired default for n
           BigInteger seed = new BigInteger(1, seedBytes);
           state = seed.mod(n);
    }

    /**
     * Returns up to numBit random bits
     *
     * @return int
     */
    public int next(int numBits) {
           // TODO: find out how many LSB one can extract per cycle.
           //   it is more than one.
           int result = 0;
           for (int i = numBits; i != 0; --i) {
              state = state.modPow(two, n);
              result = (result << 1) | (state.testBit(0) == true ? 1
: 0);
           }
           return result;
    }

    /**
     * A quickie test application for BlumBlumShub.
     */
    public void main(String[] args) {
           // First use the internal, stock "true" random number
           // generator to get a "true random seed"
           SecureRandom r = new SecureRandom();
           System.out.println("Generating stock random seed");
           r.nextInt(); // need to do something for SR to be
triggered.

           // Use this seed to generate a n-value for Blum-Blum-Shub
           // This value can be re-used if desired.
           System.out.println("Generating N");
           int bitsize = 512;
           BigInteger nval = BlumBlumShub.generateN(bitsize, r);

           // now get a seed
           byte[] seed = new byte[bitsize/8];
           r.nextBytes(seed);

           // now create an instance of BlumBlumShub
           BlumBlumShub bbs = new BlumBlumShub(nval, seed);

           // and do something
           System.out.println("Generating 10 bytes");
           for (int i = 0; i < 10; ++i) {
              System.out.println(bbs.next(8));
           }
```

```
            // OR
            // do everything almost automatically
            BlumBlumShub bbs2 = new BlumBlumShub(bitsize /*,+ optional
random instance */);

            // reuse a nval (it's ok to do this)
            BlumBlumShub bbs3 = new BlumBlumShub(nval);
    }
}
```