Lehigh University
Lehigh Preserve

Theses and Dissertations

1997

# Cryptographically strong pseudo-random bit generators constructed using finite fields

Eric Pine
*Lehigh University*

Follow this and additional works at: http://preserve.lehigh.edu/etd

Recommended Citation
Pine, Eric, "Cryptographically strong pseudo-random bit generators constructed using finite fields" (1997). *Theses and Dissertations.* Paper 473.

This Thesis is brought to you for free and open access by Lehigh Preserve. It has been accepted for inclusion in Theses and Dissertations by an authorized administrator of Lehigh Preserve. For more information, please contact preserve@lehigh.edu.

Pine, Eric

# Cryptographically Strong Pseudo-Random Bit Generators Constructed Using Finite Fields

June 1, 1997

# Cryptographically Strong Pseudo-Random Bit Generators Constructed Using Finite Fields

by

Eric Pine

A Thesis

Presented to the Graduate and Research Committee

of Lehigh University

in Candidacy for the Degree of

Master of Science

in

Computer Science

Lehigh University

May, 1997

This thesis is accepted and approved in partial fulfillment of the requirements for the Master of Science.

May 1, 1997
Date

Thesis Advisor

Thesis Co-Advisor

Chairperson of Department

Acknowledgments

I would first like to thank Prof. Samuel Gulden for his departmental support and approval of my study of and thesis on cryptographically strong pseudo-random bit generators. This subject is one which is recognized as having great interest from both a mathematical and computer science point of view.

I am also indebted to Dr. Bruce Dodson, Professor of Mathematics at Lehigh University, my primary thesis advisor. He was invaluable in helping me to focus on the mathematical precision required, while keeping in mind the larger subject at hand. A seminar on elliptic curves which provided much of the background work on computationally difficult mathematical problems was led by Dr. Dodson, and was the springboard for this work. I extend a great thanks to both for helping me to study a subject in which I continue to have great interest.

Table of Contents

Abstract

Many cryptographic applications rely on random numbers at various stages of the algorithm to assure the level of security expected by the analysis. Since truly random numbers are difficult to acquire, a set of conditions is given which when satisfied by a sequence provide numbers possessing the qualities of randomness required for cryptographic applications. More specifically, bit generators are discussed, as they can be used to generate numbers of any size. Pseudo-random bit generators are the implementation of algorithms which generate bits satisfying the requirements for cryptographic use. The type of pseudo-random bit generators (or PRBG) which are discussed are called complexity theoretic. As the name implies, the strength of the algorithm, and thus the level of security afforded the generator is founded on the intractability of a mathematically difficult problem. Of the problems of this type, the three most often applied are: factoring, discrete logarithms, and quadratic residues. These three problems for the basis for the PRBGs discussed: BBS-generator, RSA-generator both of which rely upon factoring, the discrete logarithm generator over finite fields, and the discrete logarithm generator over elliptic curves. Any algorithm which is able to predict the next bit from any of these generators, is expected to be also able to solve the mathematically difficult problem upon which the generator is based and has been proven for the discrete logarithm cases. It is this implication which yields the cryptographic strength for any generators of the complexity theoretic type.

## Introduction

The subjects of prime numbers, discrete logarithms, factoring, and quadratic residues have been studied for many years, but it is only recently that they have become generally accepted as useful for data security. This paper focuses on cryptographic applications of the area of theoretically hard computing. Although there are many aspects of cryptography where the above mentioned areas of mathematics have broadened the horizons of those studying the subject, the purpose of this paper is to focus on a small subset of these applications, the pseudo-random bit generator (PRBG).

The first section provides motivation for this study and an extended introduction. Section two discusses the topics in mathematics required to study these generators. Section three discusses the mathematically difficult problems which form the basis of many PRBGs, as well as describing algorithms which solve these problems. Section four describes the basic requirements a function must satisfy in order to generate pseudo-random bits sufficiently random for use in cryptographic applications. Finally section five discusses several PRBGs which meet these requirements. Readers familiar with quadratic residues, elliptic curves, finite fields, and the notion of hard computations may wish to proceed directly to section four.

## 1. Motivation

Like many terms in the sciences and mathematics, the definition of the phrase "random number" varies depending upon the application. These numbers have found their use not only in areas of computing, but throughout the sciences. For instance, when testing software, it is often important to input data that may or may not be of the appropriate type or magnitude. While a truly random value would be sufficient, they are rarely used due to the difficulty of finding or generating them. In many cases, what is used is an arbitrary value, yet there are areas where such values are insufficient. One area where more stringent rules for random-like values are often required is the science of cryptology. Cryptology is the science of securing or hiding data, and is split up into two subjects: cryptography, the study of constructing codes, and cryptanalysis, the study of breaking or cracking these codes.

Before the mathematical revolution in cryptology, the two main techniques for securing data were, substitution and transposition. Substitutions are usually 1-to-1 or 1-to-many mappings; while transpositions simply re-arrange the order of the data. By combining these two techniques alone, strong ciphers can be produced. In fact, DES (the Data Encryption Standard) which was adopted for use in commercial and unclassified U.S. Government applications in 1977, is merely that, a sequence of transpositions and substitutions. It must be noted, that although this protocol for encryption (DES) has not been shown to have any major holes through which a

cryptanalyst (someone involved in the cryptanalysis side) may base an attack, a slight change in one of the substitutions or transpositions may be expected to yield a very breakable code. In particular, though the idea of a transposition or a substitution is trivial, combining these techniques in such a way as to provide a secure cryptographic algorithm is a non-trivial task.

Cryptologists refer to the message which is being sent as *Plaintext*. Whereas, *Ciphertext* is the encrypted message. Simple access to the ciphertext is no longer sufficient to understand the message. It should be noted that although the terms for both the original message and the result of the encryption have the word **text** in them, it is not usually necessary for the message to be an actual text-message, binary files could just as easily be encrypted by most modern algorithms.

Many modern algorithms do not rely on the secrecy of the algorithm itself to add to the security of the cipher. The way security is often achieved is through the use of *keys*; while the algorithm itself may be known, without the specific key used to transform the plaintext into ciphertext, the algorithm is of little use. Keys are used in ciphers much the way that passwords are used today to gain entrance into other software packages. To encrypt a message in a keyed security system, you must enter both the plaintext to be encrypted and the key to be used. When designing a keyed crypto-system, there are three features which may be included. First, from a theoretical standpoint, if a certain

length key is found to provide insufficient security for a particular application, one could ideally choose a longer key which would provide the added security desired (although in practice, many ciphers which use keys are not flexible enough to allow for keys of varying lengths). Second, from a practical standpoint, if a key is compromised (in this case we are speaking of a third party who gains unauthorized access to the key - rather than the key becoming unknowingly modified) the authorized users can simply change the key and continue to enjoy the security of their algorithm, without having to redesign an entire new system. A third area of importance is related to the first two, and that is the simple fact that it is much easier to transmit and keep secret a small key, rather than an entire encryption system. It is important to realize that although keyed systems have allowed crypto-systems to evolve into much easier to manage systems, the generation of these keys becomes an important topic; for since the algorithm is not secret, if someone can guess the key, the system is not only no longer secure, it is completely compromised.

One method used in discussing the strength of a crypto-system is brute-force, which is a measure of how long it would take to break a system if every key were to be tried, or more simply a count of the number of possible keys. One reason that this technique is apparently attractive as an attack against many cryptographic systems is that such a technique can require little knowledge of the system being used, while still being able to systematically test each of the keys for a system. This technique is usually only applied to block or stream algorithms such as DES mentioned above. The reason is that

the number of possible keys for these algorithms is often sufficiently small that testing each of the possible keys is plausible, or brute-force is the best known method for breaking the system. It should be noted that plausibility is a notion which changes over time. For instance, as recently as ten years ago, the DES encryption system was considered immune to a brute-force attack, while today, such an attack can be successful using numerous workstations and PCs. Recent brute-force attacks have successfully searched the keyspace for 40 and 48 bit DES, and are expected to complete the search on the standard 56 bit keyspace through a distributed effort. For algorithms such as the RSA encryption method, which bases its strength on the difficulty of factoring numbers, or those based upon the Diffie-Hellman method, a brute-force attempt could not be expected to find the correct values, even given more time than the age of the universe and computational resources greater than those available in the foreseeable future. Descriptions of RSA, Diffie-Hellman and other cryptographic algorithms are presented in [14] and [16].

For instance, if the key were 100 bits long, but the middle 80 were always known, an attacker would only have to try $2^{20}$ keys before it would be guaranteed to find the correct one. This problem is exactly that which was brought to the attention of much of the computing industry with the attack on Netscape's Navigator SSL or Secure Socket Layer. The basic idea of the crypto-system employed by Netscape for an encrypted method of passing information was not flawed, but required the generation of

a secret key that the two communicating systems would share. In fact, the basic algorithm for creating the random key was also not seriously flawed either, rather it was in the implementation that a seemingly secure system was shown to be completely devoid of security. To generate the key, the idea was to take a few pieces of information that only one user could know, and create a seed for a standard accepted mixing function (MD5 or Message Digest 5). The output for this mixing function would then be used to create the key for encrypting data. The problem in implementation was that the pieces of information chosen were: pid (process ID), ppid (parent process ID), and time. It then becomes immediately obvious that any attacker with an account on this computer could immediately find out both the pid and the ppid, while the time could be narrowed down to a tight window by watching when the initial packet was sent out. In fact, even attackers without an account on the system can mount a serious attack since both the pid and ppid can be gleaned from the system. But even if the attacker has no ability to check these, their structure, as well as the structure of the time on the system, allows for only 47 bits of randomness. A brute force attack on a key length of 47 bits is in fact not only possible but rather trivial with today's computing power. The result is that an algorithm expected to deliver a key consisting of 128 random bits, provides only 47. It is important to realize that this is not an example of inexperience, but rather a misunderstanding of the importance of each phase of a cryptographic-system. More detailed information on the Netscape key generation can be found in Goldberg and Wagner [4].

A similar mistake was made by a more widespread and more widely respected system, that of Kerberos [13]. Kerberos is the system, created at MIT, which allows networked computers to remain secure by controlling access to the system and system resources, described in [6] and [14]. The Kerberos system is incorporated into many products, for example, NFS (Network File System). Although the key generating problem was realized by the creators of Kerberos when the Netscape problem was announced, it was surprising to many, that a product developed at MIT, from whence many cryptographic techniques have started, possessed the same flaw as that of Netscape, a company with very limited cryptographic experience. If a cryptographically strong PRBG (Pseudo Random Bit Generator) were used for both of these applications, keys providing the security of 128 random bits could have been generated resulting in a key and a code unable to be broken with a brute-force method even in the foreseeable future.

Key generation, although an extremely important aspect of modern cryptography is not the only use for PRBGs. There is one cipher which uses no keys and no advanced mathematics which is in fact a perfect cipher; that is, given an unlimited amount of time and computing resources, an attacker could never break the code, it is called the *one time pad*. The idea of a one time pad is to encrypt every piece of information individually with a separate key, and while many examples show the use by encrypting an ASCII letter using another ASCII letter as the key, one can easily adapt the algorithm to

be used with bit-wise rather than character wise encryption. While the implementation can use many types of functions to use a bit from the key to encrypt the bit from the plaintext, the exclusive-or operation works quite well. The example below will help to illustrate the system:

| | |
|---|---|
| plaintext (binary) | : 1010001001001 |
| key1 (binary) | : 1100101100010 |
| ciphertext (binary) | : 0110100101011 |
| key2(binary) | : 1001011101011 |
| plaintext2 (binary) | : 1111111000000 |
| key3(binary) | : 0110100010100 |
| plaintext3 (binary) | : 0000000111111 |

Notice, to create the ciphertext, simply take each bit of the plaintext and combine the appropriate bit of the key using the exclusive-or function. It is easy to see, once can easily re-construct the original plaintext by combining the correct key (key 1) with the ciphertext using the very same exclusive-or function. The next two portions of the example illustrate that since to the attacker, the key is unknown, the sequences labeled plaintext2 and plaintext3 are just as likely if the keys were key2 and key3 respectively. Since the keys are generated randomly, not only are all of the above 3 keys equally likely, any sequence of binary digits of length 13 is equally likely to be the correct original plaintext message. This is the reason that this is called the only perfect cipher, any possible sequence of binary digits of the proper length is possibly the correct original plaintext from any ciphertext.

Two problems become obvious when looking at this system: first, since the key has to be the same length as the plaintext to afford this perfect security, long keys must be stored for use by both parties to be able to send longer messages or multiple short messages (note that re-use of the keys completely nullifies the strength of this algorithm). Second, since for the system to be secure, random bits must be generated for the keys, there is the problem of finding truly random sources. While the first problem of secure storage and distribution of the keys is an important one, it will not be addressed except to say that PRBGs help by reducing the amount of data required to be distributed or stored. It is the issue of more quickly generating cryptographically strong pseudo-random bits without a truly random source which will be the focus for much of the rest of this paper.

# 2. Introductory Mathematics

Much of the work in modern cryptography requires the use of higher mathematics. While modern number theory is often necessary to create new cryptographic codes or to cryptanalyze current codes, only the few important results from the areas of number theory and modern algebra will be presented which are required for understanding the PRBGs discussed in the following sections. There are two groups which most of our mathematics will be concerned with: finite field of p-elements, and elliptic curves. In the next section, we will find that the finite field of p-elements has more properties than those of a group. Yet for the purposes of constructing PRBGs, it is mainly the aspects of the cyclic multiplicative group which we will find most useful. The other features will be thoroughly discussed, as they are required in order to construct some of the methods for attacking the generators. Also, when we say that elliptic curves are the other group we are concerned with, we mean the points on an elliptic curve, when confining the curve to a finite field, and in our case, a specific type of finite field. Since both cyclic groups require an understanding of finite fields, we will begin by explaining the important parts of their structure.

## 2.1 Finite Field Mathematics

A finite field of $p$-elements where $p$ is an odd prime will be denoted using the symbol $F_p$, and for example, a finite field of 41 elements would be written as $F_{41}$. Every finite field can be associated with an odd prime $p$ or with the prime $p$=2. Of these two main types

of finite fields, the fields with $2^k$ elements are particularly well suited to machine computation on today's hardware. Yet for our purposes, we will focus on finite fields constructed with a prime number of elements as they are the basis for most of the theory and algorithms in our current study. The numbers in this finite field may be represented by the ordinary numbers, 0,1,2...p-1. One feature which defines a field is that both the multiplication and addition operations are well defined. For example, in $F_{11}$ with elements represented by 0,1,2,...,10, to perform addition, we may often perform ordinary integer addition, for example 2+3=5, 4+5=9. If we use integer addition on some elements we run into an apparent problem, for example, 7+8=15, yet 15 is not in our field of 11 elements. The way we account for this is to consider addition modulo the prime of the field. For example, 7+8=15=4(mod 11). In fact, the modulo operator will be required to define both addition and multiplication for our finite field. Thus, when we use the addition and multiplication operators on elements in a finite field, we will always consider the operations modulo the prime which is the basis for the specific field we are concerned with. In reality, the numbers we are considering as comprising our field are only representatives of equivalence classes. The following are examples of equivalence classes mod 11

...,-33,-22,-11,0,11,22,33,...
...,-32,-21,-10,1,12,23,34,...
⋮
...-23,-12,-1,10,21,32,43,...

To form the elements our finite field $F_{11}$, we must choose one element from each of the equivalence classes. For simplicity we consider the elements 0,1,...10 as representatives of their equivalence classes, and thus the elements in our finite field.

Another feature of the finite field that we must be concerned with is the order of the field. For the types of finite fields we are looking at, this is rather trivial, as the order is equivalent to the prime number we selected for the modulo function. When we are using multiplication, we actually use one fewer elements, we leave out the element 0. Not only can we exclude this element without incurring any inconsistencies, as no two elements when multiplied together can yield a multiple of our selected prime, we must exclude this element in order to ensure each element has a well defined multiplicative inverse. We also note that a finite field has the usual elements acting as the additive and multiplicative identities as the integers, 0 and 1 respectively. Since we have defined the multiplicative and additive identities, we would like to ensure the inverses for each element in the field under either operation, with the exception of 0 which is not considered for multiplication. The additive inverses are trivial to find, to find the additive inverse for $a$ simply negate the value as in case of ordinary integers, thus $-a$ becomes the inverse. But since if $a$ is in our field and is not equal to the additive identity element 0, $-a$ will not be in the field. We simply find the appropriate representative element from the equivalence class; or more easily simply add the prime order to the element, so $-a$=p-$a$. Determining the multiplicative inverses is a slightly more

complicated task. For the fields we are working with we would like to use an extension of Euclid's Algorithm for finding the greatest common divisors of two integers. Euclid's Algorithm requires that neither integer divides the other; for our use, this condition is satisfied, since one of the numbers for which we will be applying the algorithm on is the prime, and the other is the element whose inverse we wish to determine. The first step, is to write the prime as

$$p = aq_0 + r_0$$

with $0 \leq r_0 < a$. We then find values so as to rewrite $a$ as

$$a = r_0 q_1 + r_1$$

with $0 \leq r_1 < r_0$. We then find values so as to rewrite $r_0$ as

$$r_0 = r_1 q_2 + r_2$$

again with $0 \leq r_2 < r_1$. We continue this process until for some k, $r_k = 0$ so

$$\begin{aligned} r_{k-3} &= r_{k-2} q_{k-1} + r_{k-1} \\ r_{k-2} &= r_{k-1} q_k + r_k \\ r_{k-1} &= r_k q_{k+1} + 0. \end{aligned}$$

In Euclid's algorithm, the number $r_k$ is the greatest common divisor of p and a. But we are not specifically interested in the greatest common divisor since we have already noted that the value will always be 1, so we start to backtrack the algorithm and re-write the next to the last equation as

$$r_k = r_{k-2} - r_{k-1} q_k.$$

We then note that we can do the same with the equation above to find

$$r_{k-1} = r_{k-3} - r_{k-2}q_{k-1}.$$

We can then substitute this equation into the one we found before to be left with

$$r_k = r_{k-2} - (r_{k-3} - r_{k-2}q_{k-1})q_k$$

by combining like terms we are left with

$$r_k = r_{k-2}(1 + q_k q_{k-1}) - q_k r_{k-3}.$$

We continue to rewrite the equations from Euclid's Algorithm and substitute them into this equation in the order reverse from that of their generation. When the first equation has been rewritten and substituted in, the resulting equation will be of the form

$$r_k = a \cdot w + p \cdot v.$$

But since in our field, $p \equiv 0$ and since we have determined that $r_k = 1$,

$$1 \equiv a \cdot w$$

so

$$w = a^{-1}.$$

We will work through a small example to more clearly display the method. We wish to find the multiplicative inverse of 25 in $F_{211}$. We start by writing our first equation as

$$211 = 25 \cdot 8 + 11.$$

We then continue with the algorithm until we have a remainder 0

$$25 = 11 \cdot 2 + 3$$
$$11 = 3 \cdot 3 + 2$$
$$3 = 2 \cdot 1 + 1$$
$$2 = 1 \cdot 2 + 0$$

Now that we have a remainder of 0, we check the next to the last equation and see that the remainder is 1, thus the gcd(211,25) is 1; a fact we knew since 25 is smaller than the prime 211. We then work backwards to construct the inverse starting by rewriting the next to the last equation as

$$1 = 3 - 2 \cdot 1.$$

By substituting a re-written second equation we are left with

$$1 = 3 - (11 - 3 \cdot 3) \cdot 1 = 4 \cdot 3 - 11$$

continuing this process we calculate

$$1 = 4 \cdot (25 - 11 \cdot 2) - 11 = 4 \cdot 25 - 9 \cdot 11$$
$$1 = 4 \cdot 25 - 9 \cdot (211 - 25 \cdot 8)$$

which when rewritten becomes

$$1 = 76 \cdot 25 - 9 \cdot 211.$$

From this we can easily see that $76 \cdot 25 \equiv 1 (\mathrm{mod}\, 211)$, therefore

$$25^{-1} \equiv 76 (\mathrm{mod}\, 211).$$

There has been some work done in this area to improve this method, or to devise a different method for calculating the multiplicative inverses, but this technique is widely used, and is used specifically for many implementations of the finite field structure when implementing PRBGs.

A related idea is that of $\mathbf{Z}_n$ which is a ring of integers. A ring has fewer properties than a field, most notably, there need not be multiplicative inverses. In

general, rings need not have a multiplicative identity either, although the ring $\mathbf{Z}_n$ does in fact have the usual one. Addition and multiplication on $\mathbf{Z}_n$ is the same as that of the finite field above, noting that $n$ need not be a prime. The other important point to note is that over a ring, two non-zero elements could have a product whose value is zero. For example in $\mathbf{Z}_{12}$, $4 \cdot 6 = 0$. Such a ring also does not guarantee that each element have an inverse. If we exclude the elements which have prime factors in common with our value of $n$, we can avoid such problems. This new collection forms a multiplicative group and is called $\mathbf{Z}_n^*$.

Another important result with respect to the finite fields, is called the Chinese Remainder Theorem. This theorem is used to simplify the solving of a difficult congruence, by generating a system of congruences and solving them in such a way as to provide a solution to the general problem. The situation which will give rise to the use of this technique is the following, we would like to solve some equation for

$$x \equiv a \bmod n$$

but instead of attacking this problem directly, we instead factor $n$

$$n = \prod_{i=0}^{k} q_i^{c_i}$$

where the $q_i$ are the k distinct factors of $n$, and solve the problem for each of the powers of the factors of our composite number $n$. The Chinese Remainder Theorem then not only ensures us a solution, but provides a means of generating it. First we will describe

the algorithm for finding the solution, then we will work through an example to illustrate the technique. First, we must solve whatever problem we wish to solve using each of the powers of prime factors of our composite number as the modulus. This leaves us with a series of $k$ congruences

$$\begin{aligned} x &\equiv a_1 (\mathrm{mod}\, m_1) \\ x &\equiv a_2 (\mathrm{mod}\, m_2) \\ &\vdots \\ x &\equiv a_k (\mathrm{mod}\, m_k) \end{aligned}$$

where we could substitute $m_i = {q_i}^{c_i}$ to attack a problem in the manner described above.

For the Chinese Remainder Theorem to provide a unique solution, it requires only that each of the $m_i$ are relatively prime to each other, as we described our factoring method above as the impetus for our discussion, it is easy to see that each of the $m_i$ are relatively prime as they are powers of distinct primes. We notice that

$$n = \prod_{i=1}^{k} m_i \,.$$

The formula determining the solution mod n can be defined as

$$x = \sum_{i=1}^{k} a_i M_i y_i \bmod n$$

with the $a_i, M_i, y_i$ to be defined below. Obviously $a_i$ are those from the system of congruences above. Then we define

$$M_i = \frac{n}{m_i} \text{ and } y_i = M_i^{-1} \bmod m_i$$

and we are assured a solution for $y_i$ since $m_i$ and $M_i$ are relatively prime, and we can use the Euclidean algorithm described above to find it. A further description of the method, and a proof of the uniqueness of the solution can be found in, Stinson [16].

We will now work through an example to illustrate the algorithm. We will let $n$=60 so, for our use, $m_1 = 5, m_2 = 4, m_3 = 3$. We will also choose $a_1 = 2, a_2 = 3, a_3 = 2$. Thus, our system of equations is

$$x \equiv 2 (\text{mod } 5)$$
$$x \equiv 3 (\text{mod } 4)$$
$$x \equiv 2 (\text{mod } 3).$$

We next calculate

$$M_1 = 12, M_2 = 15, M_3 = 20$$

and

$$y_1 = 3, y_2 = 3, y_3 = 2.$$

We then calculate $x (\text{mod } n)$ using the Chinese Remainder Theorem formula

$$x \equiv (12 \cdot 3 \cdot 2) + (15 \cdot 3 \cdot 3) + (20 \cdot 2 \cdot 2) (\text{mod } 60) = 47.$$

We can quickly check that each of our congruences in our system of congruences is still valid

$$47 \equiv 2 (\text{mod } 5)$$
$$47 \equiv 3 (\text{mod } 4)$$
$$47 \equiv 2 (\text{mod } 3).$$

## 2.2 Fermat's Little Theorem

We will now mention one more theorem from elementary number theory before moving on to study elliptic curves, and that is Fermat's Little Theorem. The general form of this theorem can be stated as

$$a^{\varphi(m)} \equiv 1 (\mathrm{mod}\, m)$$

where $a$ and $m$ are integers, $a$ is relatively prime to $m$, and $\varphi(m)$ is the Euler phi function. The proof and further discussion can be found in [8]. For our purposes, we will be applying this theorem only in certain situations for which $m$ is always prime, therefore, any integer $a$ such that $0 \leq a < m$ will be relatively prime to $m$. We also will note that for any prime number $m$, $\varphi(m) = m-1$. We then are left with a re-statement of the above theorem for $m$ as a prime number

$$a^{m-1} \equiv 1 (\mathrm{mod}\, m).$$

## 2.3 Elliptic curve mathematics

We now will move on to a discussion of the arithmetic and structure of elliptic curves. We will be concerned only with elliptic curves when considered over a finite field, and more specifically over a finite field of the form $F_p$. An elliptic curve is defined as the set of points satisfying an equation of the type

$$y^2 = x^3 + ax + b$$

with a,b $\in F_p$ and p>3, and so that $4a^3 + 27b^2 \neq 0$. We note, that although the ordinary integer operators are used, we are considering the addition and multiplication operations as defined over a finite field, as discussed above. Another point must be added which is the additive identity element, in the case of elliptic curves, it is called the point at infinity and we will denoted it as $O$. The set of points (x,y) which satisfy this equation are all points in our group with $x,y \in F_p$ satisfying the above relation, and this point at infinity comprise the elements in the cyclic group of the elliptic curve, called $E$. The basic operation over these points is addition. As we noted above, there is an additive identity $O$ which means that if $P$ is any point in our group,

$$P + O = O + P = P.$$

Much like in our finite field of p-elements discussed above, each element in the finite field has an inverse and is defined as

$$-(x,y) = (x,-y),$$

where -y is the additive inverse in the finite field, therefore it is the element -y (mod p).

The next definition must be of the addition of elements in the group where neither of the elements are the point $O$. There are specific equations which define the addition algorithm which are presented below to add two arbitrary points on the curve $P$ and $Q$ with

$$P=(x_1,y_1) \quad \text{and} \quad Q=(x_2,y_2).$$

If the two points are inverses, that is, if $x_1=x_2$ and $y_1=-y_2$ then $P + Q = O$, otherwise $P+Q=(x_3,y_3)$ with

$$x_3 = \lambda^2 - x_1 - x_2$$
$$y_3 = \lambda(x_1 - x_3) - y_1$$

and

$$\lambda = \begin{cases} \dfrac{y_2 - y_1}{x_2 - x_1}, if(P \neq Q) \\ \dfrac{3{x_1}^2 + a}{2y_1}, if(P = Q) \end{cases}$$

As with our other cyclic group in $F_p$ we also need to study the order of the group. This is one calculation which is often much more difficult than that required for our previous finite fields. Since we are discussing elliptic curves over a finite field $F_p$ our elliptic curve will have approximately p elements. There are upper and lower bounds on this number by a result of Hasse, which states

$$p+1-2\sqrt{p} \leq |E| \leq p+1+2\sqrt{p}$$

where $|E|$ is the order, or number of elements [8]. Although this provides a bound for the order of the points on the curve, for many of the applications we will be discussing, the exact order is necessary rather than this rough approximation. An algorithm by Schoof computes just this. The only problem with this algorithm is the speed with which it calculates the order; the algorithm has a running time of $O((\log p)^8)$ [8]. This is considered an efficient algorithm since it runs in polynomial time in $\log p$, yet for large prime integers $p$, the algorithm may not be practical. Much work has been done on this

original algorithm to increase its efficiency, and a modified version of Schoof's original work still remains the most efficient deterministic algorithm for calculating the order of points on an elliptic curve over $F_p$. There are some curves for which the order is more simple to calculate, and we will note them as they are the types will be specifically used in the construction of our PRBG. For elliptic curves over finite fields of the form $F_p$ where $p$ is a prime greater than 3, elliptic curves of the form

$$y^2 = x^3 + b$$

have exactly $p$ solutions of the form $(x,y)$, as well as the point at infinity, in Bender and Castagnoli [1]. Curves of this form are part of the collection of curves called *supersingular*.

### 2.4 Quadratic Residues

The idea of quadratic residues over finite fields, is a specific form of multiplication over $\mathbf{Z}_n$. A number $a \in \mathbf{Z}_n$ is called a quadratic residue, if there exists some other number $b$ such that

$$b \in Z_n \text{ and } b^2 \equiv a (\text{mod } n).$$

To illustrate the point, the quadratic residues for $\mathbf{Z}_{15}$ are shown below with the elements whose square is equal to that residue listed to the right of each

1: 1, 4, 11, 14
4: 2, 7, 8, 13
6: 6, 9
9: 3, 12
10: 5, 10

We mentioned in the above section in our discussion of rings, for many applications we would like to exclude elements which share a factor with our modulus, in our case above 15, and also to exclude the zero element. The elements we would be interested in, when excluding these elements, would then be the set {1, 2, 4, 7, 8, 11, 13, 14}. Of these values, the quadratic residues are only {1, 4} and our list above is restricted to the first two lines. We will leave this topic for now, but will return to it in the next section as we describe its cryptographic usefulness.

### 2.5 Number Magnitudes

A final topic before describing the mathematically difficult problems which will form the computationally difficult basis for our PRBGs, is the idea of large numbers. We have already mentioned that one of the major advantages of *key*ed crypto-systems is the ability to increase security as necessary by increasing the size of the key. In much the same way, the usefulness of our generators can be increased by working with larger and larger numbers. Not only will this increase the security in the algorithms by markedly increasing the time required to solve the difficult mathematical problems, but it will allow us to generate a longer sequence of pseudo-random bits. We will further discuss the relationship between the size of the input and the length of the resulting output sequence of pseudo-random bits in the last two sections. As was mentioned above, for the factoring problem, 129 decimal digit numbers, approximately 430 bits, can be factored

using the current methods of elliptic curve and the two sieves discussed below. In fact larger numbers have been factored, but not numbers of the type which are most difficult to factor, composites whose factors are two large primes. Low security applications often use numbers whose sizes are approximately 512 bits, or 154 decimal digits. Commercial grade security employs the use of 1024 or 308 decimal digit numbers, as numbers of this size can be efficiently computed using specialized hardware currently available. Numbers of this size are not likely to be at risk of being factored in general even using the algorithms which are most effective and efficient currently. The size of the numbers currently being used for implementations of the discrete logarithm problem, discussed in the next section, are not required to be quite as large to afford the same level of security. Although computing power is continually increasing, and the ability to connect large numbers of powerful computers to attack a single problem is becoming more and more commonplace, it is in the study and improvement of the algorithms which is more likely to move the line between that which is difficult and therefore secure, and that which is feasible and no longer sufficient. A closer look at the mathematical problems and algorithms for their solutions is presented in the next section.

### 2.6 Random and Pseudo-Random Numbers

We briefly need to discuss the differences between random and pseudo-random numbers. For our purposes we will focus on random and pseudo-random bit sequences rather than numbers. Sequences of both types must pass statistical tests ensuring that there exists no

statistical test which is able to predict the next bit in the sequence, even given the entire previous collection of bits, with probability greater than one-half. If a such a test can be found, the sequence is most definitely not random. Yet the differences are most important to notice. The first is that pseudo-random sequences are generated by a deterministic algorithm. This is important so that we are able to use a computer, which at this point in time is solely a deterministic machine, to generate these pseudo-random sequences. The second difference, related to the first, is that random sequences can not be reliably reproduced. If we can create a process to generate our supposed random sequence given a certain input, it fails this important criteria and thus at best falls into the category of pseudo-random. A more complete discussion on random and pseudo-random numbers is given with additional references in Schneier [14] and Knuth [7].

# 3. Mathematically Hard Problems

There are various types of PRBGs, and many of them are currently suitable for cryptographic applications, yet the type presented in this paper will be a sub-type which can be described as complexity theoretic. This means that the algorithms to break (i.e. to predict the next bit of) these generators are constrained by a theoretically complex problem. While there are other possibilities, there are two such main problems which form the basis for the generators discussed in the next section. The first problem is the discrete logarithm and the second is that of factoring. A third problem we will discuss is that of quadratic residues; although not the main focus of any of our PRBGs, requiring a solution to this problem adds difficulty to attacking some of the generators.

## 3.1 The Discrete Logarithm Problem

The discrete logarithm is a wonderfully difficult problem in that the idea is very simple to state while the algorithms for solving the problem are ingenious and often subtle. The simple statement is that the problem is to determine $c$ in the equation $x=g^c$, i.e. $c=\log_g x$. We will first look at this problem in the finite field $F_p$ and later will show the differences when using a different group for the computations as those for the elliptic curve.

### 3.1.1 The Discrete Logarithm Problem over Finite Fields

Though it is not incorrect to state the problem as above, it is slightly deceiving, that is, under normal instances, solving for $c$ given $x$ and $g$ would not be a problem worth

consideration for cryptographic applications. Rather, since this problem is being considered over the finite field of p-elements, the calculations will be considered over the finite field of p elements thus using the *mod* function, that is, $x=g^c \bmod p$. This is when the calculations become worthy of study. The most obvious method for solving this problem is to start with the value *g* and if it is not equal to x, square g; if x isn't equal to $g^2$ then multiply by g again to obtain $g^3$. Simply continue this process until you find the appropriate value c such that it solves the equation $x=g^c$. It is important to point out that we will in fact come across a solution, that is, there aren't an infinite number of possibilities. This fact is the result of Fermat's Little Theorem discussed in the previous section, that is, since $g^p=g$, the resulting values of g raised to a power will in fact repeat after p-1 values. What this means, is that if after p-1 iterations a solution is not found, then there is no solution. Although this algorithm will in fact provide a solution if one exists, it is unacceptably slow with complexity $O(p)$. The applications of the discrete log problem in these PRBGs will be with a p on the order of at least several hundred digits possibly even several thousand. Even with special hardware to handle integers of this size, the sheer number of repetitions required to check every possibility in this brute force solution reduces this exclusively to a theoretical basis for comparison for the other algorithms.

The next algorithm for solving this DL problem is called Shank's Algorithm (or more informally baby-step/giant-step), Menezes [11] and Stinson [16]. The key to

understanding this algorithm is the fact that each value in the group can be expressed as $a \cdot m + b$, where $m = \lceil \sqrt{p} \rceil$ and 0<= a,b <= m. While this may not immediately seem like a relevant fact, by re-writing the DL problem as $x=g^{am+b} \bmod p$, it's use may become clear. Remembering that all calculations are mod p,

$$x=g^{am+b}$$

implies

$$xg^{-b}= g^{am}$$

Thus, if all of the values for $g^{am}$ are calculated and sorted (as these are only dependent on the size of the field p, and the base g) any value which is a discrete log can be found simply by calculating $xg^{-b}$ for $0 \leq b \leq m$ and searching for a match in the pre-computed values of $g^{am}$ above. When the match is found, it is simple to use the two equations above to work backwards to find the solution to the DL problem. This algorithm is a great improvement over the brute force method described above. The two tables require $O$(m) memory, and the time to find the solution by computing the $xg^{-b}$ and searching the table can be $O$(m). This provides a trade off where some memory is required, but much time is saved by doing so.

To help to bring the abstract into focus, the following DL problem over $F_p$ will be solved using each of the methods described. Let p=61, and the problem we would like to solve be $\log_2 17 \pmod p$. The first we must calculate

$$m = \lceil \sqrt{p} \rceil = \lceil \sqrt{61} \rceil = 8$$

so our table size is 8. Next we realize that ever member of this finite field can be described as $a \cdot 8 + b$, for $0 \le a, b \le 7$. Then we calculate the pairs: $(i, 2^{8i})$ for i=0,1,...7

(0,1) (1,12) (2,22) (3,20)
(4,57) (5,13) (6,34) (7,42).

We then sort this list according to the second coordinate

(0,1) (1,12) (5,13) (3,20)
(2,22) (6,34) (7,42) (4,57).

We then calculate $17 \cdot 2^{-i}$ again for i=0,1,...,7 until the value calculated matches the second coordinate of one of the above values

$$17 \cdot 2^{-0} = 17 (\mathrm{mod}\, 61)$$
$$17 \cdot 2^{-1} = 39 (\mathrm{mod}\, 61)$$
$$17 \cdot 2^{-2} = 50 (\mathrm{mod}\, 61)$$
$$17 \cdot 2^{-3} = 25 (\mathrm{mod}\, 61)$$
$$17 \cdot 2^{-4} = 43 (\mathrm{mod}\, 61)$$
$$17 \cdot 2^{-5} = 52 (\mathrm{mod}\, 61)$$
$$17 \cdot 2^{-6} = 26 (\mathrm{mod}\, 61)$$
$$17 \cdot 2^{-7} = 13 (\mathrm{mod}\, 61)$$

Finally we find a value which matches, so since

$$17 \cdot 2^{-7} = 2^{8 \cdot 5} (\mathrm{mod}\, 61)$$

then

$$17 = 2^{8 \cdot 5 + 7} = 2^{47} (\mathrm{mod}\, 61).$$

Therefore the solution to our problem is $\log_2 17 = 47$. It should be noted that although for this specific problem we were forced to check each value of *i* in the second series of calculations, it could be the fact that a match might be found earlier. Asymptotic analysis shows that as the size of our field grows, we are better suited to calculate all values and

sort this second list as well. Checking the two lists becomes $O(m)$ but is only done one time and is a matching pass rather than a calculation pass.

The next algorithm to solve the discrete logarithm problem is Pohlig-Hellman algorithm [11],[16]. This algorithm is the result of a significant amount of mathematics, but the result is an algorithm which can provide significantly better performance. If the problem to be solved is $m=\log_g x \bmod p$, first notice

$$p-1=\prod_{i=1}^{k} p_i^{c_i}$$

where p is the prime of the finite field being considered and thus the order, and the k-$p_i$'s are all of the distinct prime factors of p-1. The first main idea, is that if m can be computed mod $p_i^{c_i}$ for every i, then the Chinese Remainder Theorem can be applied to compute m mod (p-1). The way this is done is by letting q be any $p_i$ such that

$$(p-1) = 0 \bmod q^c$$

but

$$(p-1) \neq 0 \bmod q^{c+1}$$

then $\exists$ w such that

$$m \equiv w \bmod q^c$$

and

$$0 \leq w \leq q^c - 1$$

Then

$$m = w + sq^c$$

for some s. And w can be written as

$$w = \sum_{i=0}^{c-1} a_i q^i$$

where the $a_i$, can be determined through a deterministic algorithm. With that, it is clear that the result is a system of modular equations

$$m = w_i \bmod q_i^{ci}$$

which can be solved using the Chinese Remainder Theorem. The only difficulty is the deterministic algorithm noted above for finding the $a_i$ in the summation notation of w above.

Before we explain this algorithm we must prove that the algorithm will always provide us with the correct values. The first step is to show that

$$x^{(p-1)/q} \equiv g^{(p-1)a_o/q} (\bmod p)$$

is always true as it will be the main focus of our algorithm. To begin we re-write the DL problem

$$g^m = x (\bmod p)$$

but from above we see that

$$g^{w+sq^c} \equiv x (\bmod p)$$

therefore

$$g^{(x+sq^c)(p-1)/q} \equiv x^{(p-1)/q} (\bmod p).$$

So our statement above we are attempting to show is true if

$$g^{(x+sq^c)(p-1)/q} \equiv g^{a_0(p-1)/q} (\text{mod } p)$$

which is true if and only if

$$\frac{(p-1)(x+sq^c)}{q} \equiv \frac{(p-1)a_0}{q} (\text{mod } p-1).$$

To show this is always true, look at

$$\frac{(p-1)(x+sq^c)}{q} - \frac{(p-1)a_0}{q} = \frac{(p-1)}{q}(x+sq^c-a_o)$$

$$= \frac{(p-1)}{q}\sum_{i=0}^{c-1} a_i q^i + sq^c - a_0$$

and by changing indices we can remove the $a_0$ term

$$= \frac{(p-1)}{q}\sum_{i=1}^{c-1} a_i q^i + sq^c$$

$$= (p-1)\sum a_i q^{i-1} + sq^{c-1}$$

$$\equiv 0(\text{mod } p-1)$$

therefore we have show original statement always to be true.

We now can describe the algorithm by using that result. First compute,

$$d_0 = x^{(p-1)/q} (\text{mod } p)$$

then we must solve

$$d_o \equiv g^{a_0(p-1)/q} (\text{mod } p)$$

simply by iterating $a_0$=0,1,2,... until the statement is satisfied, which gives us our value of $a_0$. If our c from the summation equation for w is 1, we are finished, if not we can modify our original DL equation by letting

$$x_1 = xg^{-a_0}$$

so then the equation from above for w now becomes

$$w_1 = \log_g x_1 \bmod q^c$$

and by the same argument as in the original DL problem we can write $w_1$ in summation notation with the same values except the lower limit on the summation is increased by one

$$w_1 = \sum_{i=1}^{c-1} a_i q^i \quad .$$

Now, by the key result we proved above

$$x_1^{(p-1)/q^2} \equiv g^{(p-1)a_1/q} (\bmod p)$$

so compute $d_1$ like before

$$d_1 = x^{(p-1)/q^2} (\bmod p)$$

Then again we calculate

$$g^{(p-1)i/q} (\bmod p)$$

for i=0,1,2,... until

$$d_1 = g^{(p-1)i'/q} (\bmod p)$$

which means that $a_1 = i'$ which solves the above equation. We repeat this process until we have solved for all c of the $a_i$ from our original summation representation of w. We have then created the first of our system of equations solving our discrete log problem mod a power of a prime factor of p-1. By repeating this procedure for each of the prime factors of p-1, we have a system of equations we can use via the Chinese Remainder Theorem to solve the DL problem mod p. There is some difficulty in addressing the placement of this algorithm in the hierarchy of efficiency. The fact that the number of operations required to use this technique is based on the size of the factors of the order of the group, i.e. p-1 implies that if the size of the largest factor is large, the algorithm is no longer an improvement over Shank's Algorithm. The solution requires

$$O(\sum c_i(\log(p-1) + \sqrt{p_i}\log p_i)$$

time to solve [11]. As you can see, if each of the factors $p_i$ are small, than this algorithm is in fact a significant improvement. If instead, p-1 has a large factor, call it $p_k$ than the order is approximately $O(\sqrt{p_k}\log p_k)$ which is not better than Shank's algorithm, yet still a great improvement over or original brute force method.

Now that the description of the algorithm is complete, it should help to look at a small example to illustrate the algorithm. The problem we will solve is the same one we solved using Shank's Algorithm

$$\log_2 17 = a.$$

First we must factor p-1

$$p-1=16=2^2\cdot 3\cdot 5$$

thus we need to generate the three equations

$$a\equiv b_1 \bmod 4$$
$$a\equiv b_2 \bmod 3$$
$$a\equiv b_3 \bmod 5$$

at which point we can use the Chinese Remainder Theorem to solve for $a$. First we will find $b_1$ to do so we note that since we will be looking at equations mod 4, and $4=2^2$ that our solution $b_1$ can be written as

$$b_1 = a_0 + 2a_1 .$$

It should be noted that these $a_i$'s are those in the summation equation for w above, and not related to the $a$ as the solution of the entire problem in any other way. We will solve for $a_0$ first. First we will calculate $d_0$ as

$$d_0 = x^{(p-1)/q} = 17^{60/2} = 17^{30} \bmod 61 = 60$$

then we must find the value of $i$ such that $d_0 = g^{(p-1)i/q} \pmod p$

$$g^{(p-1)0/q} \pmod p = 0$$
$$g^{(p-1)1/q} \pmod p = 60$$

Since our relation is satisfied for $i$=1, we know that $a_0=1$. Now we will calculate $a_1$ but to calculate $d_1$ we must first calculate $x_1$

$$x_1 = x\cdot g^{-1} = 17\cdot 2^{-1} = 17\cdot 31 = 39 (\bmod 61)$$

Now we can calculate $d_1$ as

$$d_1 = x_1^{(p-1)/q^2} = 39^{60/4} = 39^{15} = 60 (\mathrm{mod}\, 61)$$

but since we again are trying to satisfy the relation $d_1 = g^{(p-1)i/q} (\mathrm{mod}\, p)$ and we already have calculated that for $i$=1 the result of the left side is 60, again $a_1$=1. We can then construct our value of $b_1$ as

$$b_1 = a_0 + 2a_1 = 1 + 2 \cdot 1 = 3$$

so our first equation for the Chinese Remainder Theorem is

$$a \equiv 3 \bmod 4 .$$

We then proceed to generate the second equation, by calculating $d_1$ for our second prime namely for $q$=3

$$d_1 = x^{(p-1)/q} = 17^{60/3} = 17^{20} = 13$$

Next we again search for the value of $i$ so that the relationship $d_1 = g^{(p-1)i/q} (\mathrm{mod}\, p)$ holds

$$g^{(p-1)0/3} (\mathrm{mod}\, 61) = 1$$
$$g^{(p-1)1/3} (\mathrm{mod}\, 61) = 47$$
$$g^{(p-1)2/3} (\mathrm{mod}\, 61) = 13$$

So for $i$=2 our relationship holds. Since the power of 3 in the factorization of p-1 is only 1, we have constructed our second equation

$$a \equiv 2 \bmod 3 .$$

Finally we proceed to the generation of third and final equation, by calculating $d_l$ for our final prime, namely for $q$=5

$$d_1 = x^{(p-1)/q} = 17^{60/5} = 17^{12} = 20$$

Next we again search for the value of $i$ so that the relationship $d_l = g^{(p-1)i/q} \pmod{p}$ holds

$$g^{(p-1)0/5} \pmod{61} = 1$$
$$g^{(p-1)1/5} \pmod{61} = 9$$
$$g^{(p-1)2/5} \pmod{61} = 20$$

So again for $i$=2 our relationship holds. Since the power of 5 in the factorization of p-1 is only 1, we have constructed our second equation

$$a \equiv 2 \bmod 5.$$

We now have all three equations we had set out to construct

$$a \equiv 3 \bmod 4$$
$$a \equiv 2 \bmod 3$$
$$a \equiv 2 \bmod 5$$

From this point we can use the Chinese Remainder theorem to solve for our original discrete log problem. The example of the algorithm described to utilize the Chinese Remainder Theorem presented in the previous section used these equations with the calculations taken mod 60 as necessary for this problem. The result of that example is 47 leading us to the solution

$$47 = \log_2 17.$$

This is the same solution arrived at by Shank's Algorithm shown previously and can be verified that in fact

$$2^{47} \equiv 17 (\mathrm{mod}\, 61).$$

The final algorithm which attacks this discrete log problem is the index calculus method [11],[16]. This method is similar to many of the powerful composite factoring algorithms which will be discussed next. We first must choose a factor base, which is a set of primes small relative to the size of the field. The first step will be to calculate the logs of *these* primes with the correct base in our field. The second step will be to find the log of any other element to the specified base.

First we must define our collection of primes, let our set of primes $\{p_1, p_2, \ldots, p_n\}$ be the n primes in our factor base. Then to solve for the logs of these elements we will construct a set of congruences. If we notice that

$$g^{x_i} \equiv {p_1}^{a_{1i}} {p_2}^{a_{2i}} \cdots {p_n}^{a_{ni}} (\mathrm{mod}\, p)$$

can be written equivalently as

$$x_i \equiv a_{1i} \log_g p_1 + a_{2i} \log_g p_2 + \cdots + a_{ni} \log p_n (\mathrm{mod}\, p-1).$$

To find the log of each of these primes, we need only construct enough congruences of the above type to ensure a unique solution modulo (p-1). The problem becomes finding powers of our base $g$ which factor into only prime elements in our factor base. This is

where, in practice, the algorithm can be modified to exploit specific attributes of the hardware and software which will be solving the problem. For instance, by having a larger factor base, we are more likely to find powers of our base which can be factored using only elements in the factor base. On the other hand, we must then also construct and solve more congruences, as well as store more values for use later in factoring our arbitrary element. For that reason, this first step is usually carried out as a pre-computation step, that is, before the algorithm is run, since the base of the logarithms will not be secret, an appropriate factor base can be selected and the logs generated. Once this is done, the same values for the factor base and their respective logs can be used for any element to solve the problem for. For a large enough number of problems in the same finite field, and the same logarithm base, this pre-computation step becomes negligible.

The next step, is to factor our arbitrary number, call it $x$ using this factor base and the logarithms of its elements. To do this we simply calculate a $d$ such that

$$d = x \cdot g^w \pmod{p}$$

for some $w$, and such that $d$ can be factored using only powers of elements contained in our selected factor base. Once an appropriate $d$ has been found, we simply note the congruence

$$x \cdot g^w \equiv {p_1}^{b_1} {p_2}^{b_2} \cdots {p_n}^{b_n} \pmod{p},$$

which like above can be rewritten equivalently using the log function as

$$\log_g x + w \equiv b_1 \log_g p_1 + b_2 \log_g p_2 + \cdots + b_n \log_g p_n (\mathrm{mod}\ p-1).$$

Since in our pre-computation step 1, we calculated $\log_g p_i$ for each i, and the $b_i$ were calculated in factoring our product $d$, and $w$ is known, we can quickly solve for $\log_g x$.

To illustrate this algorithm we will solve the same problem as in the previous two methods, $\log_2 17$ in our finite field of 61 elements. We will limit our factor base to $\{2,3,5\}$ since our field is relatively small. First we must perform the pre-calculation step of the algorithm, that is we must find powers of our base, 2, which can be factored using only powers of elements in our factor base. We find that

$$2^8 = 12(\mathrm{mod}\ 61) = 2^2 \cdot 3$$
$$2^{29} = 30(\mathrm{mod}\ 61) = 2 \cdot 3 \cdot 5$$

since our base is in our selected factor base, we now have enough equations to solve for the logarithm of each. By rewriting the above equations we find

$$8 \equiv 2\log_2 2 + \log_2 3(\mathrm{mod}\ 60)$$
$$29 \equiv \log_2 2 + \log_2 3 + \log_2 5(\mathrm{mod}\ 60)$$

and since trivially

$$\log_2 2 = 1$$

we can easily solve for the other two, namely

$$\log_2 3 = 6$$
$$\log_2 5 = 22.$$

Now that we have completed the pre-computation step, we must find a value for $d$ which can be factored with our factor base. After trying several values we find that

$$17 \cdot 2^{21} (\text{mod } 61) \equiv 12 = 2^2 \cdot 3$$

by taking the logarithm we see that

$$\log_2 17 = 2\log_2 2 + \log_2 3 - 21 (\text{mod } 60)$$

and using the logarithms calculated above in the pre-computation step, we see that

$$\log_2 17 = -13 = 47 (\text{mod } 60).$$

By checking the other methods, or simply by calculating the exponentiation of our base 2, we can see that this algorithm yields the correct solution.

It is worth noting that twice, once during the pre-computation, and once to find the $d$ we were forced to calculate a value for an arbitrary element in our field, and check to see if it satisfies some property. While this appears imprecise, with a reasonable size of the factor base, one can generally find appropriate values with only a few unsuccessful calculations. In our example, if we added 7 and 11 and 13 to the factor base, nearly every value can be factored so virtually no unsuccessful calculations are required. Even with this problem of determining the appropriate size of the factor base, and which elements to include, asymptotic running time for this algorithm has been studied. For reasonable assumptions for the factor base and field size, the running time for the pre-

computation phase is $O(e^{(1+o(1))\sqrt{\ln p \ln\ln p}})$ while the time to calculate an arbitrary discrete log is $O(e^{(1/2+o(1))\sqrt{\ln p \ln\ln p}})$ [16].

### 3.1.2 The Discrete Logarithm problem over Elliptic Curves

Our discussion of the discrete logarithm problem, and the algorithms used to solve it, thus far have been focused on the discrete logarithm problem over the field $F_p$. The other group we will discuss the DL over is that of an elliptic curve. The problem over this group is slightly different. We do not attempt do utilize some multiplication over this group, rather we think of the discrete logarithm problem as

$$x = \log_g m$$

which instead of studying $g^x = m$, we instead look at

$$xg = m.$$

While this appears to be a completely different problem, it is, in fact the same, since the operation we defined in the previous section for elliptic curves is addition, we must define the discrete logarithm problem in this manner. If we called that operation multiplication, the DL problem would have the same notation as for $F_p$.

To understand the DL problem over an elliptic curve, we will now show a small example of the problem. First we must select a curve, for example

$$y^2 = x^3 + 7,$$

we must also select a field to work over, for simplicity we will choose one of the form $F_p$, and for our example we will choose a smaller field, $F_{11}$. We next will find all of the points in the group. This step is not necessary, or even advisable, in practice; our purpose for this is to make our discussions easier for the explanation of the discrete logarithm problem. The points listed below in (x,y) pairs are

| | | | |
|---|---|---|---|
| (2,2) | (2,9) | (3,1) | (3,10) |
| (4,4) | (4,7) | (5,0) | (6,6) |
| (6,5) | (7,3) | (7,8) | $\boldsymbol{O}$ |

Where $\boldsymbol{O}$ indicates the point at infinity discussed in the previous section. We next must select our $g$, or our base for the logarithm. Although any one of our points could be chosen, with the exception of the point at infinity, some of the points generate the entire group and make for a more interesting example. For this reason, we choose g=(4,4). Since we explained the addition operation over elliptic curves in the previous section we will only present the results here. The notation $2g$ represents the same as it would in other types of elementary algebra, that is $2g=g+g$,

| | |
|---|---|
| 1g=(4,4) | 7g=(7,8) |
| 2g=(6,6) | 8g=(3,1) |
| 3g=(2,9) | 9g=(2,2) |
| 4g=(3,10) | 10g=(6,5) |
| 5g=(7,3) | 11g=(4,7) |
| 6g=(5,0) | 12g=$\boldsymbol{O}$. |

We must now simply select a DL problem to solve, for example

$$x = \log_g b$$

where g=(4,4) and b=(2,2). A brute force search would produce x=9 in much the same way that we generated all of the multiples of g=(4,4).

We will next illustrate Shank's Algorithm simply to show that the two problems are indeed the same and can be attacked in many of the same ways. First we notice that the order of the group is 12. We must first calculate

$$m = \lceil \sqrt{12} \rceil = 4.$$

We next solve for s=4g=(3,10), then calculate the original table,

$$\begin{aligned} 0s &= \boldsymbol{O} \\ 1s &= (3,10) \\ 2s &= (3,1) \\ 3s &= (4,4). \end{aligned}$$

Finally we start to calculate $b+(-i)g$ for $i$=0,1,2,3 until a match is found in the above table

$$\begin{aligned} b+(-0)g &= (2,2) \\ b+(-1)g &= (3,1) \end{aligned}$$

for which we see that a match is with 2s. So, we put together our information to see that

$$b+(-1)g = 8 \cdot g$$

or

$$b = 9g.$$

By checking the table above we can see that indeed, 9(4,4)=(2,2).

Thus we have illustrated that Shank's algorithm does still solve this problem. We could similarly show this for the Pohlig-Hellman method. When solving the DL problem over an elliptic curve using the Pohlig-Hellman method, we must keep in mind that the first step is factoring the order of the cyclic group. We mentioned in the previous section that a deterministic algorithm created by Schoof calculates the order of the points on an elliptic curve, and noted that in a complexity theoretical sense it is an efficient method. Yet, when solving for the discrete logarithm problem for elliptic curves over large prime fields, we find that this step of calculating order requires significant overhead which must be incurred before the Pohlig-Hellman algorithm can even be used. There is an even more significant problem when we try to use the Index Calculus method. There is no way to extend this method, in general, to any group or even to all elliptic curves in general. In fact, as of yet, there is no technique to solve the DL problem over all groups (or even all elliptic curves) which works in sub-exponential time. There is the significant exception for the supersingular curves as described in the previous section. In [8], Koblitz notes that this collection of curves is susceptible to a specialized attack which is more efficient to those applicable to any cyclic group. Thus, although for this group of curves, it is much simpler to compute the order, and thus choose a curve which is most suited to resist attacks like Pohlig-Hellman, by selecting a curve whose order has a large prime factor, it is not advisable to use one of this type for actual cryptographic applications as it could be attacked by this alternate, specialized method. It is for these two reasons, that using elliptic curves over a finite field, is becoming the most widely

studied version of the problem. In truth, elliptic curves over finite fields have only recently been studied for this purpose. It is unreasonable to believe that no further progress can be made, but as current research and algorithms remain, to use elliptic curves provides a more difficult problem than over a finite field alone.

The methods discussed to attack the Discrete Logarithm problem illustrate several important facts. The first is from a practical standpoint, we have noted that the most efficient algorithms for solving the Discrete Logarithm problem over any cyclic group require the complete factorization of the order of the group to be comprised of only small integers. By choosing an order with at least one large factor, we can ensure that the Pohlig-Hellman algorithm is as inefficient as possible. The second important fact is to realize that for our cryptographic applications we can, indeed ensure that the cyclic groups chosen possess the structure necessary to prevent any known methods from being able to easily attack the problem. This is not to say that some new method will not be developed which could add another requirement to ensure the problem is as difficult as possible to solve. On the contrary, we must keep in mind, that although some of the algorithms which can solve various of our problems, can be quite efficient, they often require a certain structure to be so. By careful construction, we can usually force the algorithms to work with worst case conditions.

### 3.2 The Factoring Problem

The next mathematical problem from which many of these PRBGs draw their strength is the problem of factoring composite numbers. Although the problem of factoring numbers is used in many cryptographic applications, we will not spend as much time on the techniques of solving this problem as we have on the problem of discrete logarithms. To state the problem more specifically, let $n$ be the composite number, the problem is to find $p_i$ and $a_i$ such that

$$n = \prod_{i=1}^{m} p_i^{a_i} .$$

As we stated before, for our purposes, and most of those of cryptographic importance, we are looking to solve

$$n = p \cdot q$$

where $p$ and $q$ are two prime numbers. Much like the problem of discrete logs, there is a simple brute force method one can use to solve the factoring problem. That is, one can simply divide our composite number $n$ by each number less than $n$. There is an obvious improvement to this simple algorithm which greatly improves efficiency, that is to note that only the numbers less than or equal to the square root of $n$ need to be checked. If no numbers less than the square root evenly divide $n$ it must be a prime, because if $n = p \cdot q$ and $p > \sqrt{n}$ then $q < \sqrt{n}$.

There is another marked improvement which can be made, and that is to check only the primes less than $\sqrt{n}$. This seems like it should be more difficult, since we would first have to determine if each number is a prime before checking if it evenly divides our composite number $n$. In fact, this is not difficult at all and the technique has a name, the sieve of Eratosthenes. The idea is to write down every number less than $n$. Start with the first prime 2, and cross off all multiples of two on the list: 4,6,8.... Then look for the smallest number which is larger than 2 which has not been crossed off, in this case 3. We repeat the process with 3, crossing off multiples of 3: 6,9,12,.... We again look for the next smallest prime, which would be 5. This process is continued until you check the number $\lceil \sqrt{n} \rceil$. The numbers remaining are the only primes less than $n$. By simply checking each of these, we can improve the efficiency. Although this is quite a good technique for factoring small numbers, the size of numbers used for cryptographic applications are often larger than 300 decimal digits. Storing a table for each integer less than some $n$ in this context would require a prohibitably large amount of memory, and checking each of the numbers would require far too much time. We must attempt to find other techniques for factoring numbers.

One interesting technique can be best demonstrated by looking at a small example. If the number we needed to factor was 377, we could notice that 377 is the

difference of the two numbers 441 and 64. If we further notice that $441=21^2$ and $64=8^2$ then we can rewrite 377 as

$$377 = 441 - 64 = 21^2 - 8^2 = (21-8)(21+8) = 13 \cdot 29.$$

Thus we have managed to factor our composite number 377 by simply finding two perfect squares whose difference is equal to our composite. This technique can solve some harder problems, but the difficulty in finding two perfect squares whose difference is equal to a specific composite number, is a non-trivial task. For this reason, this technique is rarely used for the factoring of cryptographically useful composite numbers. The next technique is called Pollard's p-1 method [16]. This is the first of the techniques which require the input of another number which serves as a bound much in the same way that our factor base was necessary for the Index Calculus method for solving the discrete logarithm problem. We begin with our number $n$ which we would like to factor and we have the input value for the bound $B$ to work from. We begin by calculating

$$a \equiv 2^{B!} (\mathrm{mod}\, n)$$

we then find the greatest common divisor (gcd), which Euclid's algorithm from the previous section can solve

$$d = \gcd(a-1, n).$$

If our number $d$ is larger than one and not equal to $n$ we have found that $d$ is a factor of $n$.

The proof is rather simple and relies on only three main points, for notation we will assume that $p$ is a factor of our number $n$. First we note that

$$a \equiv 2^{B!} (\text{mod } n) \Rightarrow a \equiv 2^{B!} (\text{mod } p)$$

since p|n. The second important fact is Fermat's Small theorem discussed in the previous section which in our case implies that

$$2^{p-1} \equiv 1 (\text{mod } p).$$

As we noted before, (p-1)|B!, so we can see that

$$a \equiv 1 (\text{mod } p)$$

Finally, the final main mathematical point is that if

$$p|(a-1) \text{ and } p|n$$

then

$$p|\text{gcd}(a-1, n).$$

The key to this technique, is that there exists a factor $p$ of our number $n$ such that ($p$-1) has only small factors (i.e., those less than our bound B). With this in mind, for cryptographic applications which rely on the factoring problem as the basis for its security, the two large primes $p$ and $q$ must be chosen so that one less than each has at least one large factor. By choosing these primes this way, we can render this attack inefficient.

The next three algorithms for factoring numbers have been designed to factor very large numbers. Some examples are the very famous RSA-129 number which was

generated with the advent of the RSA encryption method which was expected to withstand years of factoring attempts. The three advanced algorithms are the quadratic sieve, the elliptic curve and the number field sieve. The running times of each are presented below [16],

| | |
|---|---|
| quadratic sieve | $O(e^{(1+o(1))\sqrt{\ln n \ln\ln n}})$ |
| elliptic curve | $O(e^{(1+o(1))\sqrt{2\ln p \ln\ln p}})$ |
| number field sieve | $O(e^{(1.92+o(1))(\ln n)^{1/3}(\ln\ln n)^{2/3}})$ |

where $p$ is the smallest prime factor of $n$. It should be noted, that if $p$ represents the smallest prime, the elliptic curve method finds a single factor of the composite number. If this algorithm is to be used to completely factor a composite number, then $p$ represents the second largest prime factor. The composite numbers of cryptographic interest are the product of two primes of similar size; and for this type of number, the second largest prime factor is, in fact, the smallest prime factor. The difference between the sieving algorithms and the elliptic curve method, is that the running time for the sieving algorithms depends only on the size of the composite number, whereas the running time for the elliptic curve method depends on the smallest factor of the composite number. Through the expressions for running time above, we might expect that the quadratic sieve and the elliptic curve algorithms would run in the same time for these types of cryptographic numbers, yet the constants represented by *o(1)* have great impact on the actual running time. In practice, for composite numbers of cryptographic interest, the sieving algorithms typically outperform the elliptic curve method. The other interesting

aspect of the elliptic curve method, is that the technique requires searching for an elliptic curve whose order can be completely factored by powers of small primes. It is exactly this type of elliptic curve that we aim to avoid in choosing a curve for which the discrete logarithm problem is difficult. It is in this way that the two mathematically difficult problems are related. More information on the two sieving algorithms, quadratic and number field, can be found in Pomerance's article [12], and more information on the elliptic curve method can be found in Lenstra's paper [9].

### 3.3 Quadratic Residues

The last difficult problem that we will mention is that of quadratic residues. A discussion of the problem is found in the paper by Blum, Blum and Shub [2]. We briefly discussed quadratic residues in the previous section and will present the problem they can present, but we will limit our discussion to the specific situations which arise in our study of PRBGs. We will study quadratic residues in the following situation, let $n$ be the product of two distinct odd primes, then let $Z_n^*$ represent the multiplicative group of integers mod $n$ as described in the previous section. We find that exactly one fourth of the elements in this multiplicative group will be a quadratic residue. The problem is to determine if a given element of the group is in this set of quadratic residues. As always there is a brute force solution, calculate $a^2$ for each element $a$ in our group $Z_n^*$. The solutions for the quadratic residue problem, are split into two types. If the $n$ is a prime number, then there are techniques which can be used to efficiently attack the quadratic

residue problem. If instead, $n$ is a composite number, as will be the case for our study, each of the algorithms require a complete factorization of $n$ before beginning the algorithm. For this reason, we will not dwell on this problem, for although it is an important problem, the factoring aspect forces this problem to be at least as hard as that of factoring, therefore it is sufficiently difficult for our needs.

# 4. Theory of Cryptographically Strong Pseudo-Random Bit Generators

In this section we will begin by describing ordinary pseudo-random bit generators, we will then specify requirements which when satisfied by these ordinary pseudo-random bit generators allow them to be considered cryptographically strong. The types of bit generators we are most interested in are those with an iterative implementation. The basic idea is to start with some input string, and convert it into a pseudo-random output string of greater length.

## 4.1 Definition of a Pseudo-Random Bit Generator

A pseudo-random bit generator $G_k$ is a mapping

$$G_k:\{0,1\}^k \rightarrow \{0,1\}^l$$

where $k$ and $l$ are positive integers, $\{0,1\}^a$ represents any $a$-bit binary string, and

$$P(k) = l$$

where $P(k)$ is a polynomial function [16]. The idea is to express that the generator maps random input strings into pseudo-random output strings which are longer than the input.

The generator $G_k$ is comprised of several mappings which depend upon a set $I_k$ called the instance space. An element in the instance space is called an instance or an "instance of the generator". The first mapping is $s_k$ which maps our initial input string into the instance space

$$s_k:\{0,1\}^k \rightarrow I_k .$$

A simple illustration of the mapping $s_k$ is given in section 5.1. The second mapping is a function $f_k$ which is the iterative portion of our generator, and which maps the instance space to itself

$$f_k: I_k \rightarrow I_k .$$

As an example, in one of the generators discussed in the next section, $I_k$ are the points on an elliptic curve and $f_k$ is a computationally defined permutation of these points. The third function maps the instance space to a pseudo-random output string

$$v_k: I_k \rightarrow \{0,1\}^b$$

where $b$ is the number of bits which can be generated from each instance and added to the pseudo-random output string. For most of the generators we will study in the next section, this value $b$ will be the value 1, that is, for each iteration of the function $f_k$, only one bit will be added to the output string. For the rest of this paper, we will drop the subscript $k$ from our mappings and refer to them as simply s, f, and v.

Using this modified notation we will next describe how the three functions work together to produce our pseudo-random bit string. If we consider a random bit-string input, $r$, we will use our first function $s$ to generate the first instance for iteration

$$x_0 = s(r).$$

We then will use our iterating function to generate the sequence of instances

$$x_i = f(x_{i-1}).$$

Finally we use our bit-string extracting function $v$ on each of the instances to generate our output bit-string

$$v(x_{l-1})v(x_{l-2})\cdots v(x_1)v(x_0).$$

The length of the output string can be any number less than or equal to that defined by our value l=P(k).

## 4.2 Requirements for Cryptographic Strength

We will now describe some of the requirements for one of our PRBGs to be cryptographically strong. Before we begin, we must make note of another function which some theorems will make use that is merely the combination of two mappings

$$b(x) = vf^{-1}(x).$$

The idea, is while $s, f$ ,and $v$ should be easy to calculate (i.e. polynomial time) in order for G to be considered a PRBG, $b$ should not be. We recall that the function $f$ for our generators will be one of our mathematically difficult problems presented in the previous section. Since the function $v$ should be able to be computed quickly, for $b$ to be difficult, $f^{-1}$ must be difficult, i.e. inverting or solving the problem upon which $f$ is based should be a complexity theoretic difficult problem.

### 4.2.1 Statistical Tests

We start by discussing statistical tests. The statistical tests should be run on the output string, as opposed to the instances. One might want to begin by some very elementary statistical tests such as counting the number of 0's and 1's to ensure that the occurrences of each is the same for random inputs. Yet, by passing such a simple test, it would be hard to immediately classify the output as indistinguishable from random input. In fact, no matter how sophisticated the statistical tests become that any generator passes, it is conceivable that the very next test constructed would distinguish between our pseudo-random output and random data. What we must ensure is that every statistical test which runs in polynomial time is unable to distinguish our pseudo-random data from random data. The idea of passing an arbitrary statistical test is found in most papers discussing PRBGs including Blum and Micali [3] and Kaliski [5]. We define an arbitrary statistical test $T$ that runs in polynomial time which outputs a 0 or a 1 with the input of a bit string. Passing all such tests requires that for sufficiently large bit-strings, $T$ will output a 1 with the same probability whether the input string is chosen from a random source, or from the PRBG being tested. More precisely for sufficiently large k

$$|\Pr[T(G(x)) = 1] - \Pr[T(x') = 1]| \leq \frac{1}{P(k)}$$

where P(k) is a polynomial function and G(x) is the output string from our PRBG and $x'$ is a random string of the appropriate length.

### 4.2.2 Next Bit Tests

While the above test is concerned with entire output string compared to random output strings, this test is concerned with the prediction of the next bit in the pseudo-random sequence given the previous bits generated. We must ensure, that the probability of predicting the next bit of the generator by one of the polynomial time statistical tests T from above is not significantly greater than 1/2. More precisely we can state that

$$\Pr[T(G(x)_{[i,i-1]}) = G(x)_i] \leq \frac{1}{2} + \frac{1}{P(k)}$$

where again P(k) is a polynomial function, and $x$ is our random input, [3] and [5].

## 4.3 Sufficient Conditions for Cryptographic Strength

We will state the three conditions which are sufficient for a PRBG to be considered cryptographically strong. The proof of the sufficiency is given in Blum and Micali [3]. The three necessary conditions are accessibility, stability, and unapproximability. The first condition of accessibility requires that given a random input string $r$ the mapping $s(r)$, which is the first step in the generator, selects an element from the instances uniformly in polynomial time. More precisely

$$\left| \Pr[s(r) = x_0 = -\frac{1}{|I_k|} \right| \leq \frac{1}{P(k)|I_k|}$$

where $x_0 \in I_k$ and $|I_k|$ is the number of instances. The second condition, stability, states that the iterated function $f$ is a permutation of the instances in $I_k$, or more precisely

$$|f(x)|=|x|.$$

The addition of the constraint that $f$(x) be computable in polynomial time is often added, although our statement at the outset of this discussion concerning the three mappings of our generator noticed that for G to be run in polynomial time, each of the three mappings must also run in polynomial time. The third sufficient condition unapproximability, states that our mapping $b$ can not be predicted in much the same way the statistical tests were shown effective

$$\Pr[C(x) = b(x)] \leq \frac{1}{2} + \frac{1}{P(k)}$$

where C(x) is any polynomial time mapping from instances into bit strings of the appropriate size.

When all three of these conditions have been met, we have ensured not only a cryptographically strong PRBG, but a Blum-Micali pseudo-random bit generator. Work has been done to consider conditions which are both necessary and sufficient, and these results are discussed in Kaliski [5]. This work allows for the construction of PRBG which are cryptographically strong, yet do not rely on functions $f$ which are uninvertable for every instance. Though this result may prove useful in constructing future cryptographically strong PRBGs, the generators that we will discuss in the next section are all of the Blum-Micali type. Although we discussed the statistical test and the next bit test, our three conditions do not appear to make use of these results. In their paper

[3] Blum and Micali prove that the three conditions stated above, necessarily give rise to generators which produce output which satisfies these two tests. This is an important result, as the sufficient conditions stated above are more easily validated than the statistical tests.

# 5. Construction of Cryptographically Strong PRBGs.

While we set forth a set of conditions which are sufficient for cryptographic strength of PRBGs in the previous section, we would now like to study examples of such generators. We will restrict ourselves to generators which are of the Blum-Micali type satisfying the three conditions for cryptographic strength presented in the previous section.

## 5.1 Blum-Blum-Shub Generator

We will start our discussion of the construction of cryptographically strong PRBGs by describing one of the first generators constructed, one which is still in use today both in practice and as a benchmark by which newly constructed generators are compared. This generator was first described in Blum, Blum and Shub [2], and is thus called the Blum-Blum-Shub (or BBS) generator, and the strength of the algorithm uses aspects of all three computationally difficult problems discussed in the above section: factoring, the discrete logarithm, and quadratic residues. We will start with the description of the various mappings and functions which comprise the generator and then show that the conditions are met classifying the generator as cryptographically strong. All of the calculations for this generator are on $Z_N^*$ the multiplicative group of integers modulo $N$, which is defined in section 2.1, but for this application, $N$ is restricted to be of the form $N = p \cdot q$ with $p$ and $q$ are both primes such that

$$p \equiv q \equiv 3 (\mathrm{mod}\, 4).$$

The instances, $I_k$, for the generator are $x$, such that $x$ is a quadratic residue in $Z_N^*$, where $N$ must be chosen to be sufficiently large. Our mapping from the random input to an initial instance, $s$ will in practice be some complicated construction, but for purposes of illustration, it is sufficient to consider the following construction. Regard the input string as an integer value written in binary notation, square this value and find the residue class in $Z_N^*$. We note that there is a statistically insignificant possibility that the bit string chosen will not produce a value relatively prime to $N$, in which case, a new bit-string should be chosen. The iterative function $f$ is the simple function

$$x_i = f(x_{i-1}) = x_{i-1}^2 (\mathrm{mod}\, N).$$

We then will define our function $v$ as the parity of the bits in our instance $x_i$.

To illustrate this generator we will generate the first several bits of an example, let $p$=11 and $q$=19 therefore N=209. If our input random bit string is 1110111, we convert this into a decimal value, giving us 119. We square this value to create the first of the instances

$$119^2 \equiv 158 (\mathrm{mod}\, 209)$$

thus $x_0$=158. We then iteratively perform the squaring operation on our instances to generate $x_1, x_2, \ldots, x_7 = 93,80,130,180,5,25,207$. The output bits will be constructed using the parity function $b_0 = parity(x_0) = parity(119) = 1$. The rest of the parity bits are $b_1, b_2, \ldots, b_7 = 1,0,0,0,1,1,1$. Thus we have calculated

G(1110111)=11100010.

We could continue calculating additional pseudo-random output bits, but not indefinitely. Using 158 as a seed for the instances, we find that instance generates a cycle of order 12 thus four more bits can be generated yielding an output bit string of 110011100010. It should be noted that although we ordered the bits $b_l b_{l-1} \ldots b_1 b_0$ some other references order the output bits in the opposite order. Often it depends on the application, for if the output will be used bit-wise, it is not likely to be grouped and thus will be in the opposite order as that presented here, while if the output will be used to construct a pseudo-random key for a separate crypto-system, the bits most often would be reversed as shown above.

One feature if this PRBG is the ease with which we can compute an arbitrary bit in the output. To compute the $i^{th}$ bit in the output, we simply compute

$$x_i = x_0^{2^i} \pmod{N}$$

and

$$b_i = parity(x_i).$$

In practice this is useful, for instance if a file containing a sequence of pseudo-random bits generated from the BBS-generator has a corrupted section, only these bits need be re-generated.

We will make two notes before continuing on with the discussion of the strength of this generator. First the example uses numbers significantly smaller than any which would ever be used in cryptographic applications. While in our example the output string was twice as long as the random input string, for other small examples this may not hold true. It will only be for large values of $N$ that the size of the output string will be consistently and significantly larger than that of the input string. The second point is to keep in mind what information is public in this system. The only information that an attacker or cryptanalyst has access to is $N$, the output string and the algorithm. Information which is required to be private to ensure security of the generator are the prime factors $p$ and $q$, as well as the input bits. If an attacker gains access to any of these pieces of information, the system is vulnerable. Keeping $p$ and $q$ private is not a difficult task, as after $N$ is computed, the factors are no longer needed and can be safely discarded. This leaves only the random input bit-string, the key, to which serious attention must be paid.

The three sufficient conditions for cryptographic strength of a PRBG are easily shown to be satisfied for the BBS generator if we show that it is difficult (i.e. can not be done in polynomial time) to calculate $f^{-1}(x)$. In Blum, Blum, and Shub [2], it is shown that factoring our composite number $N$ is sufficient to invert our function $f$. The cryptographic strength of the BBS-generator relies upon the belief that inverting $f$ is in

fact equivalent to factoring $N$ (see Stinson [16]). The proof of cryptographic strength under this hypothesis is also presented in [2].

One question that remains concerns the lengths of the output strings. In order for a generator to satisfy the definition of a pseudo-random bit generator, it must generate output strings which are longer than the random input strings. The authors of [2] assert that this condition can be satisfied for this generator, by choosing a more sophisticated function $s$.

### 5.2 The RSA PRBG

The RSA pseudo-random bit generator [16], is based on the RSA crypto-system named after its creators Rivest, Shamir, and Adleman. The RSA crypto-system is widely considered one of the most secure public key systems today. Its use as a general cryptographic system is limited only by the relatively slow speed with which it generates the ciphertext. The security of the RSA PRBG again rests in the problem of factoring large composite numbers. The algorithm can be perceived as an extension of the BBS generator presented above. Again, we choose two primes, $p$ and $q$ and multiply them together to form half of the public information N. We then must choose an exponent $b$ such that $\gcd(\varphi(N), b) = 1$, where $\varphi(N)$ is the Euler Phi function. For our purposes, since N is the product of two primes, $p$ and $q$, $\varphi(N) = (p-1)(q-1)$. The algorithm proceeds in much the same way as that of the BBS generator above, except that the

value of the input function $s$ need be a quadratic residue. The function $f$ which iterates the instances is replaced by

$$x_i = f(x_{i-1}) = x_{i-1}{}^b (\text{mod } N).$$

We extract the bits from the instances in the same way as the BBS-generator, by using the parity function.

To illustrate this generator we will present a small example, keeping in mind that the magnitude of these numbers is small enough as to afford us no security. We will choose our $p$ and $q$ the same as above in the example for the BBS generator, $p$=11, $q$=19, and multiplying them together yields N=209. We must now find our exponent $b$ which is relatively prime to

$$\varphi(N) = (p-1)(q-1) = 10 \cdot 18 = 180 = 2^2 \cdot 3^2 \cdot 5.$$

In the interest of simplicity we will simply choose 7. We will again use the random input bit string 1110111 which when converted to an integer value is 119. We first calculate

$$x_1 = x_0{}^b (\text{mod } N) = 119^7 (\text{mod } N) = 92.$$

We continue to iteratively calculate $x_i$ and find

$$x_1, x_2, \ldots, x_{11} = 92, 93, 157, 130, 169, 5, 168, 207, 81, 16, 36$$

and we can extract our bits to see

$$b_1, b_2, \ldots, b_{11} = 0, 1, 1, 0, 1, 1, 0, 1, 1, 0, 0$$

so our resulting output-bit string is 00110110110. We can see that this string is different from that generated in our BBS-generator example above, but the system can be shown cryptographically strong using the same arguments used for the BBS-generator. This is possible since both systems rely on the factoring of a composite number into exactly two large primes for its security.

### 5.3 The Discrete Logarithm Generator over a Finite Field

We will now consider a pseudo-random bit generator which uses the difficulty of the discrete logarithm problem over a finite field for its security. The foundation for the cryptographic strength of this algorithm is proven by Blum and Micali in [3], and an explicit generator based upon this work can be found in Stinson [16]. We will again start with a description of the basic algorithm. Since we will be performing our operations over a finite field $F_p$ we must first select a $p$ for the prime order for our computations. We then must select a primitive element $g$, modulo our prime $p$. For this generator, the instances are the elements of $Z_p^*$, the multiplicative group of integers modulo $p$. We construct the seed element for our algorithm in $\mathbf{Z}_p^*$, by mapping the random input bit-string to a residue class $x_0$ prime to $p$. Our iterative function $f$ then generates the instances using the formula

$$x_i = f(x_{i-1}) = g^{x_{i-1}} (\mathrm{mod}\, p).$$

The function $v$ for this generator is no longer the parity function, but rather

$$v(x_i) = \begin{cases} 1 & if \quad x_i \geq (p+1)/2 \\ 0 & if \quad x_i < (p+1)/2 \end{cases}.$$

To illustrate this generator we will again show a specific example using numbers not large enough for cryptographic security. We will choose our prime element to be 223. We find that the element 3 is a generator for the cyclic group of order 222, modulo 223. We will again use the random input bit string 1110111. Our mapping $s$ again will simply convert the random bit stream into an integer, in this case 119; thus $x_0 = 119$. We then calculate the first instance for our generator applying our function $f$

$$x_1 = 3^{119} (\text{mod } 223) = 129.$$

By iterating our function $f$, we can generate the elements

$$x_1, x_2, \ldots, x_{12} = 129, 87, 174, 171, 155, 107, 11, 85, 168, 14, 65, 22.$$

To generate the output string we must apply our function $v$ to extract a bit from each of these instances. We note that (p+1)/2 = 112, therefore our first bit is

$$b_1 = v(129) = 1$$

since $129 \geq 112$. The rest of the bits generated are

$$b_1, b_2, \ldots, b_{12} = 1, 0, 1, 1, 1, 0, 0, 0, 1, 0, 0, 0$$

Thus we have the relation that

G(1110111)=000100011101.

It should be noted that the generator is not limited to 12 bits, rather we simply generated the first 12 bits to serve as a comparison to the other generators.

In their paper [3], Blum and Micali reduce the problem of cryptographic strength for this generator to the statement that the function $f$ can not be inverted with any polynomial time function. Unlike the previous cases relying on factoring, inverting the function $f$ depends upon solving the discrete logarithm problem which we have discussed as a computationally hard problem in section 3.

An improvement was made to this algorithm from an implementation standpoint by Long and Wigderson [10]. They noticed that in the function $v$ in the discrete logarithm generator, only one bit was being extracted per iteration of $f$. This is often the case, as the proofs of cryptographic strength are typically easier when considering only one bit per iteration. Yet for the discrete logarithm problem, they showed that $v$ can be modified to extract $\log(n)$ bits, where the prime $p$ is an $n$-bit prime. They proof used shows that if an algorithm can be constructed to solve the discrete logarithm over the finite field $F_p$ using the knowledge added by extracting $\log(n)$ bits, it can be modified to solve the discrete logarithm problem with only extracting one bit. The technique they use relies upon noticing that the function $v$ splits the field up into two partitions of equal size. If a modification is made to split the field up into $2^k$ partitions, we can extract k

bits. By restricting the size of k to log(n), we are still afforded the complete security provided by the discrete logarithm problem.

### 5.4 The Discrete Logarithm Generator over Elliptic Curves

For our next generator, we change the group over which the computations occur. We have already discussed the discrete logarithm problem as it relates to elliptic curves as defined over finite fields. Kaliski, in [5], constructs a pseudo-random bit generator using the discrete logarithm problem over an elliptic curve. In the section with background mathematics above, we defined an elliptic curve as

$$y^2 = x^3 + ax + b$$

yet for now, we will restrict ourselves to a more simple case, when the coefficient $a$ is set to 0, or more explicitly

$$y^2 = x^3 + b \qquad \text{(simple case)}$$

After we describe the generator, we will briefly mention the modifications required to use arbitrary elliptic curves to construct a generator.

The definition for the functions $f$ and $v$ are slightly more involved than those for the PRBG over a finite field. We begin by choosing a curve for our calculations, ensuring it is of the form as the simple case; we will represent it as E. We must next find a generator for the group, and will represent it as $g$. The instances for the generator are

simply the points $p$ in the group over our elliptic curve, where $p$=(x,y). We now can define our function $f$ as

$$f(p) = h(p)g$$

where

$$h(p) = \begin{cases} y & if \quad p = (x, y) \\ p & if \quad p = O \end{cases}$$

recalling that $O$ represents the point at infinity. Similarly, we must take some care in constructing an appropriate mapping for $v$ as well. In [5], it is shown that $v$ can be defined as

$$v = \begin{cases} 1 & if \quad P = O \\ 1 & if \quad P = (x, y) \text{ and } y \geq (p+1)/2. \\ 0 & otherwise \end{cases}$$

To illustrate this generator we will work through a small example. We will use the same elliptic curve and finite field as we did for the example of the discrete logarithm problem in the earlier section. The curve we selected was

$$y^2 = x^3 + 7,$$

and the finite field we were working over was $F_{11}$, so the elements of the group, and therefore instances of the generator are of the form $p$=($x$,$y$), such that $x,y \in \mathbf{Z}_p$. We found that the element $g$=(4,4) generates the cyclic group of order 12. To begin the algorithm, we must establish the initial instance, we shall assume that the element (7,8)

was chosen at random as the seed so $x_0$=(7,8). We now begin the iterative portion of the algorithm to generate the instances of the generator

$$x_1 = f(x_0) = 8g = (3,1)$$

similarly we can generate the rest of the instances

$$\begin{aligned}
x_2 &= f(x_1) = 1g = (4,4)\\
x_3 &= f(x_2) = 4g = (3,10)\\
x_4 &= f(x_3) = 10g = (6,5)\\
x_5 &= f(x_4) = 5g = (7,3)\\
x_6 &= f(x_5) = 3g = (2,9)\\
x_7 &= f(x_6) = 9g = (2,2)\\
x_8 &= f(x_7) = 2g = (6,6)\\
x_9 &= f(x_8) = 6g = (15,0)\\
x_{10} &= f(x_9) = 0g = O\\
x_{11} &= f(x_{10}) = 11g = (4,7).
\end{aligned}$$

We notice here that since the entire group is of order 12, we can no longer continue to iterate to generate more instances. We now must extract the bits from the instances, we first calculate the value for partitioning the instances

$$\frac{(p+1)}{2} = \frac{(11+1)}{2} = 6$$

thus, for the point at infinity and any points whose $y$ coordinate is greater than or equal to 6, the corresponding bit value will be one. The values for the bits are

$$b_1, b_2, \ldots, b_{12} = 1, 0, 0, 1, 0, 0, 1, 0, 1, 0, 1, 1.$$

Therefore the resulting output bit string is 110101001001.

The main result from [5] is that a generator defined in this way over an elliptic curve, satisfies the conditions sufficient for classification as a Blum-Micali generator. We will now note the changes required for arbitrary elliptic curves to be used as opposed to limiting ourselves to those of the type of the simple case. The problem in using the general case is that it requires two such elliptic curves where the second is referred to as the Tate twist of the first curve [5]. Using general curves as the basis for a pseudo-random number generator avoids a reduction of the discrete logarithm problem for the elliptic curve to that of the finite field, Menezes [11].

When comparing all of the generators discussed, it is important to differentiate between theoretical and practical differences. The differences between the PRBG using discrete logarithms over an elliptic curve and the PRBG using discrete logarithms over a finite field, are largely practical. They are theoretically the same generator. The only theoretical difference, is related to the attacks on the discrete logarithm problem in the two groups. We noted that the Index Calculus method for solving discrete logarithms in a Finite Field has not been modified to solve such problems over an elliptic curve. It is conjectured that, in fact, any technique to solve the discrete logarithm problem over elliptic curves efficiently, (i.e. sub-exponential), will require an entirely new technique, based on further study of the structure of the groups generated by these elliptic curves.

The main result connecting all of these generators, is that the sufficient conditions for a Blum-Micali PRBG are often not difficult to prove, providing a function $f$ can be constructed which is not invertible in polynomial time. This is not to say that this requirement can replace the three sufficient conditions, yet it is one aspect which can ease in constructing a PRBG.

## Conclusion

The announcement pertaining to the cracking of the Netscape security system, brought to public attention the fact that creating and implementing secure encryption systems is more difficult than a simple function call. Many modern encryption techniques require higher mathematics and impressive amounts of computing power. Even the perfect security provided by one-time pad, requires significant planning on the acquiring of the random numbers required to afford the method any security at all. Pseudo-random bit generators can be in integral step in increasing the level of security in both of these and many other applications. What is most important to note concerning the set of generators discussed, is the fact that the security of each is based upon a mathematically difficult problem. Even more noteworthy is the fact that these problems are not esoteric or obscure in any way; they are problems which have been studied for centuries and upon which significant progress has been made. The type of uninvertable function required to base the security of a PRBG on is fairly well defined, and any other functions of this type could be shown to provide equal or better security.

Through the study of complexity theoretic PRBGs we implemented each of the systems discussed in the previous section. We made no attempt to make use of the strengths of the platforms we conducted our test on, and the tests were straightforward implementations of the algorithms presented. We were able to generate the cryptographically strong bits as expected and the only limits of the flexibility of the

generators with respect to the size of the numbers used, were only those presented by the amount of memory and disk space on the machines. In our testing, we discovered an obstacle which could lower the security in certain cases of the generators. Occasionally we would inadvertently select initial conditions which provided short cycles of pseudo-random bits. By short cycles of pseudo-random bits, we mean that the iterating function $f$ in our generator repeats for smaller, often much smaller, values than we would like. One problem is that we are dealing with numbers which are too small for cryptographic purposes and expecting to extrapolate that information onto applications using much larger numbers. Although we did not search for this problem in fields of 500 bits or larger, which would be those used for cryptographic applications, we were limited by memory and disk space.

One of the smaller examples of the problem is presented here concerning the discrete logarithm generator over finite fields. If we choose as our prime p=20011, we can find that the element 12 generates the corresponding cyclic group of order 20010. By selecting random elements and iterating our function $f$ until the original value is calculated as a result, we can explicitly find the lengths of these cycles of the permutation given by $f$. The first column is the smallest element used to generate the corresponding cycle and the second column is the cycle length.

1: 8825
8: 6296
9: 1960
38: 1148
17: 818
14: 526
86: 330
208: 80
404: 26
6571: 1

Since the number of bits required to express the prime for our field, 20011, is 14 bits, as long as each cycle produces more than 14 bits the definition of the pseudo random bit generator is satisfied. As you can see the element 6571 has a cycle 1 and obviously fails this requirement since

$$12^{6571} = 6571$$

using the exponentiation over our finite field. If we selected this element as our seed value, our pseudo-random string would be a string composed entirely of 1s. Other examples were found for which more than one element failed this requirement.

After reproducing this issue for other generators, we looked to current research on PRBGs for an explanation or a possible solution. Work mentioned in [5] and [15] show that the probability of selecting a cycle short enough to render the generator insecure becomes very small as the size of the groups becomes large. It is still possible, in general, to inadvertently select one of these short cycles. Some preliminary work has

been done to modify the BBS-generator so as to prevent elements having short cycles from being selected in any cases. Although this is not a significant concern from a theoretical standpoint, from that of implementation, having the probability close to zero for selecting one of these short cycles is not a strong enough condition to rely on the security of a crypto-system using such a generator. This area of short cycles of bits, is one which is currently being and must continue to be investigated. Addressing this problem could very well be the final step in assuring that the generation of pseudo-random bits of the type discussed in this paper are cryptographically secure enough to satisfy the conditions necessary for use for key generation for public encryption crypto-systems and for the one-time pad.

References

1. Bender, A. and Castagnoli, G., "On the Implementation of Elliptic Curve Cryptosystems." *Adv. in Cryptology - Crypto '89*. Springer-Verlag, 1990, 186-192.

2. Blum, L., Blum, M., and Shub, M., "Comparison of Two Pseudo-Random Number Generators." *Adv. in Cryptology - Crypto '82*. Plenum Press, 1983, 61-78.

3. Blum, M. and Micali, S., "How to Generate Cryptographically Strong Sequences of Pseudo-Random Bits." *SIAM Journal on Computing*. Nov. 1984, 850-864.

4. Goldberg, I., and Wagner, D., "Randomness and the Netscape Browser." *Dr. Dobb's Journal*. Jan. 1996, 66-76.

5. Kaliski, B. S., "A Pseudo-Random Bit Generator Based on Elliptic Logarithms." *Adv. in Cryptology - Crypto '86*. 1987, 84-103. Prelim. version of an MIT PhD thesis.

6. Kaufman, C., Perlman, R., and Speciner, M., *Network Security: Private Communication in a Public World*. Prentice Hall, 1995.

7. Knuth, D., *The Art of Computer Programming: Volume 2, Seminumerical Algorithms* 2nd edition. Addison-Wesley, 1981.

8. Koblitz, N., *A Course in Number Theory and Cryptography*. Springer-Verlag, 1987.

9. Lenstra, H.W. Jr., "Factoring Integers with Elliptic Curves." *Ann. of Math.* **126** (1987), 649-673.

10. Long, D. and Wigderson, A., "The Discrete Logarithm Hides O(log n) Bits." *SIAM Journal on Computing*. Apr. 1988, 363-372.

11. Menezes, A.J., *Elliptic Curve Public Key Cryptosystems*. Kluwer Academic Publishers, 1993.

12. Pomerance, C., "A Tale of Two Sieves." *Notices of the American Mathematical Society*. Dec. 1996. 1473-1485.

13. Sandberg, J., "Major Flaw in Inteernet Security System is Discovered by Two Purdue Students." *The Wall Street Journal*. Feb. 20 1996. B7B.

14. Schneier, B., *Applied Cryptography*. John Wiley & Sons, 1996.

15. Shepp. L. A. and Lloyd, S. P., "Ordered Cycle Lengths in a Random Permutation." *Transactions of the American Math. Soc.* **121** (1966), 340-357.

16. Stinson, D. R., *Cryptography: Theory and Practice*. CRC Press, 1995.

# Vita

Eric Jason Pine was born June 12, 1973 in Alburquerque, New Mexico; son of Vernon W. and Zoe Lynda. Graduated from Thomas Jefferson High School for Science and Technology in June 1991. Received a BS in Mathematics from Lehigh University in January 1995 graduating with highest honors. He is a member of Phi Eta Sigma, Tau Beta Pi, and Phi Beta Kappa.

# END OF TITLE