The Blum-Goldwasser cryptosystem

Mike Hogan*

May 5, 2010

Abstract

In this project we will investigate the Blum-Goldwasser Cryptosystem, a public-key, probabilistic cryptosystem. We will implement the cryptosystem on a computer using the Python programming language. In addition, we will analyze the security and computational complexity of the cryptosystem. Along the way, we will also investigate the Blum-Blum-Shub streamcipher, an essential component of the cryptosystem. In addition, we will explore the number-theoretic background (Chinese remainder theorem, Fermat's little theorem, quadratic residues, and so on) needed for the analysis of these topics. Finally, we will improve upon the estimate for the period of the Blum-Blum-Shub sequence.

Contents

1 Introduction **2**

2 Background **2**
2.1 Complexity 3
2.2 Kerckhoff's Principles of Cryptographic Security 4
2.2.1 The system should be unbreakable in practice 4
2.2.2 Security through obscurity is bad 4
2.2.3 The public key is easily communicable and changeable 4
2.2.4 Cryptogram should be transmissable digitally 4
2.2.5 Encryption apparatus is portable and operable by a single person 4
2.2.6 The system is easy to use 4
2.3 Number theory 4

3 Pseudo-random Binary Sequences **9**
3.1 Generalities 10
3.2 BBS stream cipher 10

*Department of Mathematics, United States Naval Academy, Honors Thesis 2009-2010, Advisor Prof. Joyner, wdj@usna.edu

4 BG cryptosystem **15**
4.1 Encryption 15
4.2 Decryption 16
4.3 Complexity 18
4.4 Security 18

5 Conclusion **19**

6 Appendix: Sage code **19**

1 Introduction

The need for information security has been a paramount concern for military operations dating back to the Roman Empire and the rule of Julius Caesar. According to his biographer, Suetonius, Caesar kept messages confidential by using a simple shift cipher. The Japanese code, known as JN-25 to the Americans, was broken in 1942 to allow advanced knowledge of the impending invasion of Midway, and contributed to the American victory in the same battle. However, since the beginning of the information age, the need for cryptographic security has expanded to include encryption of bank information for everday consumers making online purchases or even everyday communications by all people.

In addition to the need for security in everyday communications, the need for security in military communications has never been higher. Engaged in wars on two fronts with an evolving enemy and having forward deployed assets in locations all over the world, we cannot afford to have communications intercepted and used against us. I hope that my research can increase the knowledge of the basics behind current cryptosystems and lead to new thinking on improving our current methods.

In this paper, I will be taking an in depth look at the Blum-Goldwasser crypstosystem, a probabilistic, public-key cipher with minimal, if any, usage in applications today. I will discuss the math behind the cryptosystem, as well as the complexity, efficiency, and security. Through the Blum-Goldwasser system, I will also explore the Blum-Blum-Shub stream cipher, a key component of the system. Using special Blum primes, I will establish a better estimation for the period of the pseudorandom sequence than that given in [BBS] (see p. 378). Finally, I will implement the Blum-Goldwasser cryptosystem in Sage with the Python coding language.

2 Background

Cryptosystems are used to ensure that your messages are not intercepted and read by an adversary you don't want to gain knowledge of your correspondances. There are two things that distinguish cryptosystems from one another: the complexity, or efficiency, of the system and the security behind the system. In this section, [Men] is used as a general reference.

2.1 Complexity

In order to discuss the complexity of the cryptosystem, we first need to define an algorithm as well as two measures of complexity, O-notation and Ω-notation.

Definition 1 *(Algorithm)* An *algorithm* is a well-defined computational process that receives an input and, after a finite number of steps, yields an output.

An example of a well-known algorithm, and one that will be used for the decryption of the Blum-Goldwasser cryptosystem, is the Extended Euclidean Algorithm.

`Algorithm`:
INPUT:(a, b): integers, $a \geq b$, $b > 0$
OUTPUT: (d, x, y): where $d = gcd(a, b)$ and integers x, y which satisfy $ax + by = d$

1. Initialize $x_1 = 0, x_2 = 1, y_1 = 1, y_2 = 0$
2. While $b > 0$,
   (a) Set $q =$floor$(a/b), r = a - qb, x = x_2 - qx_1, y = y_2 - qy_1$
   (b) Set $a = b, b = r, x_2 = x_1, x_1 = x, y_2 = y_1, y_1 = y$
3. Set $d = a, x = x_2, y = y_2$, and return (d, x, y)

The following notations are two of the methods to measure the complexity of an algorithm. O-notation will give us an upper bound of complexity, while Ω-notation will yield a lower bound.

Definition 2 *(O-notation)* Let f, g be positive functions on the natural numbers. Then we say

$$f(n) = O(g(n))$$

if there exists a constant $c > 0$ and an $n_0 > 0$ such that $0 \leq f(n) \leq c \cdot g(n)$ for all $n \geq n_0$.

An example of O-notation is $n^2 + 1 = O(n^2)$. This is true because $n^2 + 1 \leq 2n^2$, for all $n \geq 1$, after taking $c = 2$.

Definition 3 *(Ω-notation)* Let f, g be positive functions on the natural numbers. Then we say

$$f(n) = \Omega(g(n))$$

if there exists a constant $c > 0$ and an $n_0 > 0$ such that $f(n) \geq c \cdot g(n) \geq 0$ for all $n \geq n_0$.

To further clarify these definitions, let us look at the complexity of the Extended Euclidean Algorithm defined above.

Remark 1 Let $f(a, b)$ be the Extended Euclidean Algorithm. Then $f(a, b) = O(logn)$. Also, since $f(a, b)$ can be found in any number of steps, there is no lower bound to it, thus Ω does not exist for this algorithm.

2.2 Kerckhoff's Principles of Cryptographic Security

Auguste Kerckhoff articulated six rules for cryptographic security in his 1883 article, "La cryptographie militaire," part of *Journal des sciences militaires*.

2.2.1 The system should be unbreakable in practice

While a system that is mathematically unbreakable is ideal, a fundamentally secure cryptosystem needs only to be unbreakable in practice. The Blum-Goldwasser cryptosystem is breakable only through factoring $n = pq$, a problem that is believed practically impossible if n is 1000 bits.

2.2.2 Security through obscurity is bad

An adversary who knows what system is being used should not have any advantage in breaking it. The Blum-Goldwasser system is public-key, therefore it will be widely known by any adversary. In our application, an extra step is being taken to ensure against chosen ciphertext attacks, making it not quite a public-key system. However, adversaries will still have the same trouble in breaking the system even if they know the method of encryption.

2.2.3 The public key is easily communicable and changeable

The public-key is easily changed (as will be seen in the simple Python code below) and can be transferred from party to party without compromising the integrity of the system.

2.2.4 Cryptogram should be transmissable digitally

Since the only transmission is a t-length binary string, it is easily transferrable through digital means, including telegraph, the original apparatus used for Kerckhoff's theories.

2.2.5 Encryption apparatus is portable and operable by a single person

The apparatus is a computer with a Python compiler.

2.2.6 The system is easy to use

There is no need for knowledge of the system in order to use our implementation. The user needs only enter the message he or she wishes to encrypt. The implementation calculates everything for the user.

2.3 Number theory

The basis for the security of the Blum-Goldwasser cryptosystem, as well as the actual encryption and decryption, is in computing quadratic residues. In this section, I will discuss theorems generally and in the more specific case of $n = pq$, p, q distinct odd primes, that will apply to the Blum-Goldwasser cryptosystem.

The number theory behind the Blum-Goldwasser cryptosystem uses the *modulo* operation. The modulo operation yields the remainder after the division of two numbers. This lends to a set of congruencies between numbers.

Example 1 $15 \mod 2 = 1, 17 \mod 2 = 1$, so $15 \mod 2 \equiv 17 \mod 2 \equiv 1$.

Definition 4 *(Quadratic residue)* Let $a > 0$ be an integer. We say a is a *quadratic residue* $\pmod m$ if the congruence $x^2 \equiv a \pmod m$ is solvable. The set of all quadratic residues a, $0 < a < m$, is denoted by Q_m.

The order of Q_m is important for security. The larger the group, the harder it will be to randomly stumble across our quadratic residues.

Lemma 1 Let m be a prime number, $m \neq 2$. Q_m is a subgroup of $(\mathbb{Z}/m\mathbb{Z})^\times$ of index 2. In other words,

$$|Q_m| = \frac{m-1}{2}.$$

proof: Let $G = (\mathbb{Z}/m\mathbb{Z})^\times$. Clearly, $|G| = m - 1$. Let $f : G \to G$ be defined as $f(x) = x^2 \mod m$. Then, $f(ab) = (ab)^2 = a^2b^2 = f(a)f(b)$ since G is abelian. Thus, f is a homomorphism. By definition, $Image(G) = Q_m$. Since $|Ker(f)| = |G|/|Q_m|$ and $|Ker(f)| = |\{1, m-1\}| = 2$, then $|Q_m| = (m-1)/2$.$\square$

The Blum-Goldwasser system utilizes the product of two primes, $n = pq$. We want to look at the order of the group Q_n.

Lemma 2 Let p, q be odd primes with $n = pq$. Then the order of the group of quadratic residues mod n is given by:

$$|Q_n| = |Q_p||Q_q|.$$

Example 2 Here is an example of Lemma 2 entered using `Sage`:

Sage

```
sage: n = 11; QR = list(Set([x^2%n for x in range(1,n) if
gcd(x,n)==1])); QR; len(QR)
[1, 3, 4, 5, 9]
5
sage: n = 13; QR = list(Set([x^2%n for x in range(1,n) if
gcd(x,n)==1])); QR; len(QR)
[1, 3, 4, 9, 10, 12]
6
sage: n = 11*13; QR = list(Set([x^2%n for x in range(1,n) if
gcd(x,n)==1])); len(QR)
30
```

We can prove this formally as well:

proof: Let p, q be distinct odd primes, and let $n = pq$. Suppose $x^2 \equiv a \pmod{p}$ and $x^2 \equiv a \pmod{q}$ for some $x \in \mathbb{Z}$. Then $x^2 \equiv a \pmod{pq}$. Further, since $x^2 \equiv a \pmod{p}$ has either 0 or 2 solutions, and $x^2 \equiv a \pmod{q}$ has either 0 or 2 solutions, we see $x^2 \equiv a \pmod{pq}$ has either 0 or 4 solutions. In particular, if $G = (\mathbb{Z}/n\mathbb{Z})^\times$, then $G \to G^2 = \{x^2 \mid x \in G\}$ is a 4-1 map, so

$$|Q_n| = |G^2| = \frac{1}{4}|G| = \frac{(p-1)(q-1)}{4} = |Q_p||Q_q|.$$

□

Example 3 Here are a couple of examples of the above proof entered into `Sage`. You will see that 4 solutions is only reached when x^2 has 2 solutions each mod p and mod q.

Sage

```
sage: p = 23; q = 47

sage: a = randrange(2,20); a
2
sage: solns1 = [x for x in range(p) if (x2-a)%p == 0];
len(solns1)
2
sage: solns2 = [x for x in range(q) if (x2-a)%q == 0];
len(solns2)
2
sage: solns = [x for x in range(p*q) if (x2-a)%(p*q) == 0];
len(solns)
4

sage: a = randrange(2,20); a
13
sage: solns1 = [x for x in range(p) if (x2-a)%p == 0];
solns1; len(solns1)
[6, 17]
2
sage: solns2 = [x for x in range(q) if (x2-a)%q == 0];
solns2; len(solns2)
[]
0
sage: solns = [x for x in range(p*q) if (x2-a)%(p*q) == 0];
solns; len(solns)
[]
0

sage: a = randrange(2,20); a
5
sage: solns1 = [x for x in range(p) if (x2-a)%p == 0];
solns1; len(solns1)
```

```
[]
0
sage: solns2 = [x for x in range(q) if (x2-a)%q == 0];
solns2; len(solns2)
[]
0
sage: solns = [x for x in range(p*q) if (x2-a)%(p*q) == 0];
solns; len(solns)
[]
0

sage: a = randrange(2,20); a
16
sage: solns1 = [x for x in range(p) if (x2-a)%p == 0];
solns1; len(solns1)
[4, 19]
2
sage: solns2 = [x for x in range(q) if (x2-a)%q == 0];
solns2; len(solns2)
[4, 43]
2
sage: solns = [x for x in range(p*q) if (x2-a)%(p*q) == 0];
solns; len(solns)
[4, 372, 709, 1077]
4

sage: a = randrange(2,20); a
7
sage: solns1 = [x for x in range(p) if (x2-a)%p == 0];
solns1; len(solns1)
[]
0
sage: solns2 = [x for x in range(q) if (x2-a)%q == 0];
solns2; len(solns2)
[17, 30]
2
sage: solns = [x for x in range(p*q) if (x2-a)%(p*q) == 0];
solns; len(solns)
[]
0

sage: a = randrange(2,20); a
2
sage: solns1 = [x for x in range(p) if (x2-a)%p == 0];
solns1; len(solns1)
[5, 18]
2
sage: solns2 = [x for x in range(q) if (x2-a)%q == 0];
solns2; len(solns2)
[7, 40]
```

```
2
sage: solns = [x for x in range(p*q) if (x2-a)%(p*q) == 0];
solns; len(solns)
[87, 524, 557, 994]
4
```

Theorem 1 *(Chinese Remainder Theorem)* Suppose $n_1, n_2, \ldots, n_k$ are positive integers which are pairwise coprime. Then, for any given integers $a_1, a_2, \ldots, a_k$, there exists an integer x solving the system of simultaneous congruences

$$\begin{aligned} x &\equiv a_1 \pmod{n_1} \\ x &\equiv a_2 \pmod{n_2} \\ &\vdots \\ x &\equiv a_k \pmod{n_k} \end{aligned}$$

Furthermore, all solutions x to this system are congruent modulo the product $n = n_1 n_2 \cdot \cdots \cdot n_k$.

The unique solution to the Chinese Remainder Theorem is given by Gauss' Formula.

Theorem 2 *(Gauss' Formula)* In the Chinese Remainder Theorem, the solution x is given by

$$x = a_1 N_1 M_1 + a_2 N_2 M_2 + \cdots + a_k N_k M_k \ ,$$

where

$$\begin{aligned} N_i &= n/n_i \\ M_i &= N_i^{-1} \pmod{n_i} \ . \end{aligned}$$

In our case, if $n = pq$,

$$\begin{aligned} x &\equiv r \pmod{p} \ , \\ x &\equiv s \pmod{q} \ , \end{aligned}$$

then $x \equiv rq(q^{-1} \bmod p) + sp(p^{-1} \bmod q) \pmod{pq}$.

Theorem 3 *(Euler's theorem)* For any modulus n and any integer a coprime to n, we have

$$a^{\varphi(n)} \equiv 1 \pmod{n}$$

where $\varphi(n)$ denotes Euler's totient function counting the integers between 1 and n that are coprime to n.

Theorem 3 is a generalization of Fermat's Little Theorem, because if n is a prime number, then $\varphi(n) = n - 1$.

Definition 5 *(Carmichael's λ-function)* Let $M = 2^e \cdot {P_1}^{e_1} \cdot \cdots \cdot {P_k}^{e_k}$, where $P_1, \ldots, P_k$ are distinct odd primes. Carmichael's λ-function is defined by

$$\lambda(2^e) = \begin{cases} 2^{e-1} & \text{if } e = 1 \textit{ or } 2, \\ 2^{e-2} & \text{if } e > 2, \end{cases}$$

and $\lambda(M) = lcm[\lambda(2^e), (P_1 - 1) \cdot {P_1}^{e_1 - 1}, \ldots, (P_k - 1) \cdot {P_k}^{e_k - 1}]$.

Remark 2 When $M = pq$, with p, q distinct odd primes, then $\lambda(M) = lcm[(p - 1), (q - 1)]$. We will be using this later.

Theorem 4 *(Carmichael's Theorem)* Let a, n be integers such that a is coprime to n. Then, $a^{\lambda(n)} \equiv 1 \pmod{n}$.

Remark 3 Let $n = pq$ be the product of two primes. By Theorems 3 and 4, $a^{\phi(n)} \equiv a^{\lambda(n)} \equiv 1 \pmod{n}$. This shows that $\phi(n)$ is an upper bound for the period of the sequence $\{a^j \in (\mathbb{Z}/n\mathbb{Z})^\times | j = 0, 1, 2, \ldots\}$. It is also possible that the "true period" is a proper factor of $\phi(n)$.

3 Pseudo-random Binary Sequences

Digital communication is centered around the transmission of binary sequences, strings of 1's and 0's, that represent letters, numbers and symbols. A pseudo-random sequence is a sequence of numbers that appears random to a user, but actually repeats after some point. The length of the sequence before it repeats is referred to as the *period* of the sequence. In order to formally define a *pseudo-random binary sequence* , we must first define its *autocorrelation* .

Definition 6 *(Autocorrelation)* Let $a = \{a_n\}_{n=1}^{\infty}$, $a_n \in \{0, 1\}$ be a sequence of numbers considered "random." We say the *autocorrelation* of a is

$$C(k) = C(k, a) = \lim_{N \to \infty} \frac{1}{N} \sum_{n=1}^{N} (-1)^{a_n + a_{n+k}}.$$

If a is periodic with period P, then the autocorrelation is reduced to

$$C(k) = \frac{1}{P} \sum_{n=1}^{P} (-1)^{a_n + a_{n+k}}.$$

We will also be examining the "runs" in a binary sequence. A run is a subsequence of all 0's or 1's. The length and frequency with which each run appears has a role in determining if the sequence is pseudo random or not.

Definition 7 *(Pseudo-random binary sequence)* Let $\{a_n\}_{n=1}^{\infty}$, $a_n \in \{0, 1\}$, be a binary sequence with period P. We say this sequence is *pseudo-random* provided the following conditions hold (*Golomb's Principles*) [Gol]:

1. balance: $|\sum_{n=1}^{P}(-1)^{a_n}| \leq 1$.

   In order to satisfy this condition, a sequence with k 0's will have j 1's, $(k-1) \leq j \leq (k+1)$.

2. low autocorrelation:
$$C(k) = \begin{cases} 1, & k = 0, \\ \epsilon, & k \neq 0, \end{cases}$$
   for some "small" ϵ. For sequences satisfying these first two properties, it can be seen that $\epsilon = -1/P$ must hold.

3. Proportional Runs Property: In each period, half the runs are length 1, one-fourth are length 2, one-eighth are length 3, etc. Moveover, there are as many runs of 1's as there are of 0's.

3.1 Generalities

Now that a pseudo-random binary sequence is defined, it is important to know how this pertains to a cryptosystem. Let $m = m_0 m_1 \ldots m_t$ be a message in binary, and let $c = c_0 c_1 \ldots c_t$ be a pseudo-random binary sequence. If you add m and c component wise with a modulo of 2, you will receive a new message $b = b_0 b_1 \ldots b_t$ that is completely different from the other two. However, if you want to retrieve m again, simply add c and b in the same manner, and the result yielded is m. This is called a *stream cipher*. The Blum-Goldwasser cryptosystem relies heavily on a stream cipher known as the Blum-Blum-Shub stream cipher.

3.2 BBS stream cipher

The Blum-Blum-Shub pseudorandom number generator was created in a 1986 paper by Lenore Blum, Manual Blum and Michael Shub [BBS]. It creates a streamcipher that meets all the requirements of a pseudorandom binary sequence.

Definition 8 *(Blum-Blum-Shub streamcipher)* Let p, q be two distinct prime numbers such that $p \equiv 3 \pmod 4$ and $q \equiv 3 \pmod 4$. Let $n = pq$ and let $0 < r < n$ be a random number. We define x_0, the first number of the Blum-Blum-Shub pseudorandom number generator as
$$x_0 = r^2 \pmod n.$$
Each proceeding seed can be defined as
$$x_{i+1} = {x_i}^2 \pmod n,$$
or, more generally by Theorem 3,
$$x_i = ({x_0}^{2^i \mod (p-1)(q-1)}) \pmod n.$$
The streamcipher, $b = b_1 b_2 \ldots b_t$, is created by setting $b_i = x_i \mod 2$, thus yielding a pseudorandom string of 0's and 1's.

Our implementation uses a pseudo-random number generator built into `Sage` to generate the initial seed x_0. Many of these pseudo-random number generators are linear feedback shift registers, as discussed in [Bro]. All of these pseudo-random number generators are periodic over time. This creates a possible problem with the random seed and the Blum-Blum-Shub generator. Obviously, a short period will lead to replicated numbers, and make a message more easily decipherable. Similarly, a repeated random seed allows for easy decryption by an adversary if they have both the original message and the encrypted version . In their paper, Blum, Blum and Shub prove that in certain conditions the period, $\pi(x_0) = \lambda(\lambda(n))$. Let us define $ord_n x$ to be the order of $x \mod n$.

Theorem 5 *(Period of x_0)* Let n be a number as described above and let x_0 be a quadratic residue of it. Let $\pi(x_0)$ be the period of the BBS sequence, $x_0, x_1, x_2, \ldots$. If $ord_{\lambda(n)/2}(2) = \lambda(\lambda(n))$ and $ord_n(x_0) = \lambda(n)/2$, then $\lambda(\lambda(n)) = \pi(x_0)$.

In order to prove the equality of the statements, it is enough to show that $\pi(x)|\lambda(\lambda(n))$ and $\lambda(\lambda(n))|\pi(x)$.

proof: Recall, since $\pi = \pi(x_0)$ is the period of the sequence $x_0, x_1, \ldots$, it is the least possible integer such that $x_\pi = {x_0}^{2^\pi} \pmod n = x_0$. Also, note that $ord_n x$ is an odd number. Thus, by Theorem 4,

$$2^{\lambda(ord_n x_0)} \equiv 1 \mod (ord_n x_o).$$

By definition of π, $2^\pi \equiv 1 \mod (ord_n x_0)$, since $x_0 = {x_0}^{2^\pi} \pmod n$. Therefore, $\pi|\lambda(\lambda(n))$.

Conversely, by assumption, $\lambda(n)/2$ is the least positive integer such that ${x_0}^{\lambda(n)/2} \pmod n = 1$. So, $2^\pi \mod (\lambda(n)/2) = 1$. But, since ${x_0}^{2^\pi} \pmod n = x_0$, it is also true that ${x_0}^{2^\pi - 1} \pmod n = 1$. Therefore, $\lambda(n)/2|2^\pi - 1$. So, by assumption, $2^{\lambda(\lambda(n))} \mod (\lambda(n)/2) = 1$, and, since $2^\pi \mod (\lambda(n)/2) = 1$, it can be seen that $\lambda(\lambda(n))|\pi$. Therefore, $\pi(x_0) = \lambda(\lambda(n))$, when all the assumptions hold. $\square$

To examine this further, we need to define a type of prime that will have uses in our examples.

Definition 9 *(Special Blum Prime)* Let p be a prime number. We say p is a *special Blum prime* if and only if $p \equiv 3 \pmod 4$, $p = 2p_1 + 1$ and $p_1 = 2p_2 + 1$ with p_1, p_2 prime numbers. A number $n = pq$ is *special* if and only if p, q are distinct special Blum primes.

According to Theorem 5, we need an x_0, our initial quadratic residue, with order of $\lambda(n)/2$. Here are some Sage examples that demonstrate how often a quadratic residue has the required period.

Example 4 Here is an example, using Sage, plotting the multiplicative order of each quadratic residue of $n = 23 \cdot 47$. In this specific example, we use two special Blum primes. Note that $\lambda(23 \cdot 47)/2 = 253$.

Sage
```
sage: p = 2*11+1; q = 2*23+1; n = p*q
sage: QR = quadratic_residues(n)
```

```
sage: QR.remove(0)
sage: QR = [x for x in QR if gcd(x,n)==1]
sage: R = IntegerModRing(n)
sage: L = [R(x).multiplicative_order() for x in QR]
sage: list_plot(L)
```

The `list_plot` command is seen in Figure 1 below.

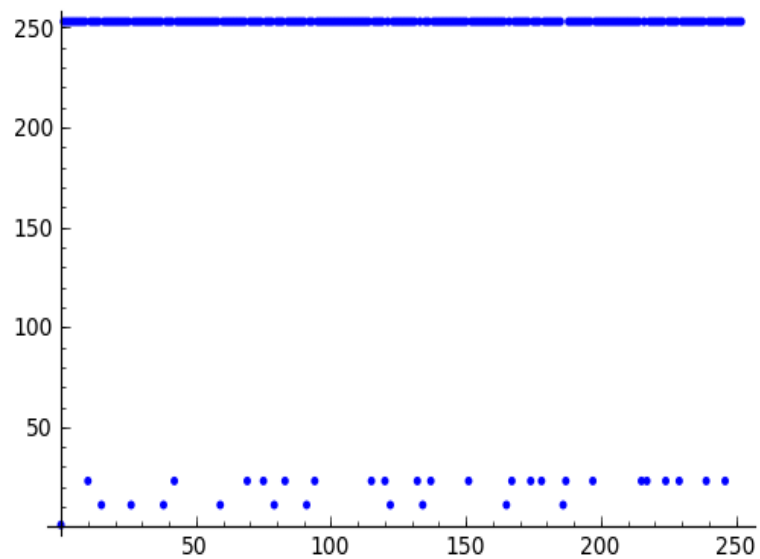

Figure 1: Quadratic residue orders $\pmod{1081}$. The dark line shows that a majority of residues have period $\lambda(x_0)/2$

Below, we have a table of a sampling of quadratic residues with their multiplicative orders as presented in the graph above.

Sage

```
sage: [(QR[i],L[i]) for i in range(10)]

[(1, 1),
 (2, 253),
 (3, 253),
 (4, 253),
 (6, 253),
 (8, 253),
 (9, 253),
 (12, 253),
 (16, 253),
 (18, 253)]
sage: [(QR[i],L[i]) for i in range(100,111)]
```

```
[(404, 253),
 (418, 253),
 (426, 253),
 (427, 253),
 (430, 253),
 (432, 253),
 (439, 253),
 (440, 253),
 (441, 253),
 (450, 253),
 (455, 253)]
sage: [(QR[i],L[i]) for i in range(115,125)]

[(484, 23),
 (486, 253),
 (487, 253),
 (491, 253),
 (495, 253),
 (507, 23),
 (512, 253),
 (518, 11),
 (519, 253),
 (524, 253)]
```

Example 5 Here is larger example using $n = 91261 = 262 \cdot 347$. As you can see, almost all multiplicative orders are 22663. In fact, only 303 of the 22663 quadratic residues $\pmod{91261}$ are not of the desired order.

Sage

```
sage: p = 263; p1 = 131
sage: q = 347; q1 = 173
sage: n = p*q; n
91261
sage: QR = quadratic_residues(n)
sage: QR.remove(0)
sage: QR = [x for x in QR if gcd(x,n)==1]
sage: R = IntegerModRing(n)
sage: L = [R(x).multiplicative_order() for x in QR]
sage: a = L[1]; a
22663
sage: odds = [i for i in range(len(L)) if L[i]!=a]
sage: len(odds); len(L)
303
22663
sage: list_plot(L)
```

The `list_plot` command is seen in Figure 2 below.

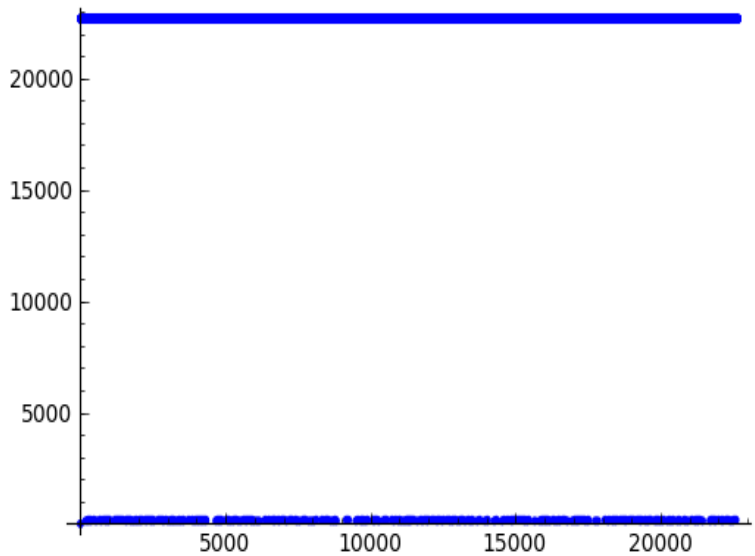

Figure 2: Quadratic residue orders $\pmod{91261}$

The above examples demonstrate that almost all quadratic residues meet the needs of Theorem 5. Looking further, at special numbers specifically, we can estimate $\pi(x_0)$.

Theorem 6 *(Period of a special number)* Let $n = pq$ be a special number and x_0 be a quadratic residue satisfying the conditions of Theorem 5. Then, $\pi(x_0)$ can be estimated asymptotically from below by $n/8$, as $n \to \infty$.

Remark 4 This improves on the estimate $\Omega(\frac{N}{(lnlnN)^2})$ given in [BBS].

proof: Let the above assumptions hold. Then,

$$\begin{aligned}\lambda(n) &= \lambda(pq)\\ &= lcm(p-1, q-1)\\ &= lcm(2p_1, 2q_1)\\ &= 2p_1q_1.\end{aligned}$$

Thus, continuing with Carmichael's function,

$$\begin{aligned}\lambda(\lambda(n)) &= \lambda(2p_1q_1)\\ &= lcm(p_1-1, q_1-1)\\ &= lcm(2p_2, 2q_2)\\ &= 2p_2q_2\\ &\approx n/8.\end{aligned}$$

□

Special Blum primes only occur as $1/(n^3 \ln^3 2)$ of all primes, but a special number has the maximum $\pi(x_0)$ approximation. In his paper, Brock discusses linear feedback shift registers and their periods. While BBS itself is not a LFSR, it can be examined in a similar manner.

Theorem 7 *(Blum-Blum-Shub as a LFSR)* Let $a_0, a_1, a_2, \ldots, a_{p-1}$ be a given BBS sequence with period p. Define $a_n = a_{n-p}$. Then, the BBS sequence can be considered a LFSR with a key length of p, approximated by $\lambda(\lambda(n)) = n/8$.

Sketch of proof: The shortest LFSR of a sequence can be defined by the connection polynomial as described in [Bro]. With a small key, the Berlekamp-Massey method is efficient to define this connection polynomial and break the sequence. However, with a long key, specifically of length $n/8$, this is just as inefficient as trying to calculate the quadratic residues.

The period length of this sequence is the one of the inherent security issues present in the Blum-Blum-Shub sequence. A large period makes it hard for an adversary to determine the seed x_0 by listening to the sequence and determining where it repeats, while a small period makes the sequence more susceptible to attacks. Another possible attack, however, comes from an adversary finding the initial seed computationally. The fact that x_0 is a quadratic residue ensures security here. If we examine p prime for quadratic residues, we will discover there are $(p-1)/2$ quadratic residues. Observe that, for $n = pq$, $[(p-1)(q-1)]/4$ numbers in $(\mathbb{Z}/m\mathbb{Z})^\times$ are quadratic residues. If someone could determine the quadratic residuosity of a number $\mod n$, then the security of the system may be compromised. However, there is an assumption about the difficulty of finding quadratic residues [BBS, GM].

Definition 10 *(Quadratic residuosity assumption)* The *quadratic residuosity assumption* states that it is "hard on average" to compute the quadratic residue of a number in $(\mathbb{Z}/m\mathbb{Z})^\times$. More formally, for each polynomial $f(n)$, the complexity is $\Omega(f(n))$, so there is no polynomial time algorithm to determine the quadratic residue of a number.

This means that for large enough n, it is virtually impossible, by sheer time constraints, to compute the initial seed x_0, making this number generator safe from this mathematical attack.

4 BG cryptosystem

The Blum-Goldwasser cryptosystem [BG] makes use of the BBS streamcipher to encrypt a message. In addition, the decryption algorithm is used to determine the initial seed of BBS, x_0, without compromising the integrity of the system.

4.1 Encryption

Definition 11 *(XOR operation)* Let $a = a_1 a_2 \ldots a_t$ and $b = b_1 b_2 \ldots b_t$ be binary sequences. We define $c = a \oplus b$ as

$$c_k = \begin{cases} 1, & a_k \neq b_k, \\ 0, & a_k = b_k \end{cases}$$

for all $1 \leq k \leq t$.

Consider a message $m = m_1 m_2 \ldots m_t$ a binary string of length t. Let x_0 be a random seed $x_0 \in Q_n$. Let $b = b_1 b_2 \ldots b_t$ be the Blum-Blum-Shub streamcipher of length t associated to x_0, n. Compute $c = b \oplus m$, where $\oplus$ indicates the XOR operation. This defines the ciphertext output $c = c_1 c_2 \ldots c_t$ of the Blum-Goldwasser encryption algorithm. Alice sends the ciphertext c along with a number $y = x_0^{2^{t+1}} \pmod{n}$.

Remark 5 Note: If we were to take $c \oplus b$ again, it will yield the original message m.

4.2 Decryption

Bob receives a ciphertext c and a number $y \in Q_n$. Let

$$d_1 = (\frac{p+1}{4})^{t+1} \mod (p-1)$$

$$d_2 = (\frac{q+1}{4})^{t+1} \mod (q-1)$$

$$u = y^{d_1} \mod p$$

$$v = y^{d_2} \mod q$$

Now, use the Extended Euclidean Algorithm (sec. 2.1) to compute a, b such that $ap + bq = 1$. Set $\overline{x_0} = ubq + vap \pmod{n}$

We now give the proof that the decryption is correct.

Theorem 8 (Decryption algorithm works)

proof: Let (c, y) be the transmitted cipher, where $y = x_0^{2^{t+1}} \pmod{n}$. Since $n = pq$, by the Chinese Remainder Theorem, see theorem 1, for each r, s,

$$x \equiv r \pmod{p}$$

$$x \equiv s \pmod{q}$$

has a unique solution $x \pmod{n}$, given by

$$x = rqb + spa \pmod{n}.$$

We know that $\overline{x_0} = ubq + vap \pmod{n}$ satisfies

$$\overline{x_0} \equiv u \pmod{p}$$

$$\overline{x_0} \equiv v \pmod{q}$$

So,

$$\begin{aligned} \overline{x_0} &\equiv y^{d_1} \mod p \\ \overline{x_0} &\equiv y^{d_2} \mod q \end{aligned} \tag{1}$$

Recall that x_0 is a quadratic residue, and is therefore equal to w^2 for some $w \in \mathbb{Z}/n\mathbb{Z}$. Therefore, when $t = 1$,

$$\begin{aligned} {x_0}^{(\frac{p+1}{2})^t} &\equiv {x_0}^{\frac{p+1}{2}} \mod p \\ &\equiv {x_0}^{\frac{p-1}{2}+1} \mod p \\ &\equiv x_0 \cdot {x_0}^{\frac{p-1}{2}} \mod p \\ &\equiv x_0 \cdot w^{p-1} \mod p \\ &\equiv x_0 \mod p, \end{aligned}$$

because of Fermat's Last Theorem. It follows, for any $t > 1$,

$$\begin{aligned} y^{d_1} &= \left({x_0}^{2^{t+1}}\right)^{(\frac{p+1}{4})^{t+1}} \mod p \\ &\equiv {x_0}^{(\frac{p+1}{2})^{t+1}} \mod p \\ &\equiv {x_0}^{(\frac{p+1}{2})^t(\frac{p+1}{2})} \mod p \\ &\equiv \left({x_0}^{(\frac{p+1}{2})^t}\right)^{\frac{p-1}{2}+1} \mod p \\ &\equiv {x_0}^{(\frac{p+1}{2})^t} {x_0}^{(\frac{p+1}{2})^t(\frac{p-1}{2})} \mod p \\ &\equiv {x_0}^{(\frac{p+1}{2})^t} z^{(p-1)} \mod p \\ &\equiv {x_0}^{(\frac{p+1}{2})^t} \mod p, \end{aligned}$$

where $z = {x_0}^{(\frac{p+1}{2})^t/2}$. Repeat this method for $t+1$ replaced by t:

$$\begin{aligned} {x_0}^{(\frac{p+1}{2})^t} &\equiv {x_0}^{(\frac{p+1}{2})^{t-1}(\frac{p+1}{2})} \mod p \\ &\equiv \left({x_0}^{(\frac{p+1}{2})^{t-1}}\right)^{\frac{p-1}{2}+1} \mod p \\ &\equiv {x_0}^{(\frac{p+1}{2})^{t-1}} {x_0}^{(\frac{p+1}{2})^{t-1}(\frac{p-1}{2})} \mod p \\ &\equiv {x_0}^{(\frac{p+1}{2})^{t-1}} z^{(p-1)} \mod p \\ &\equiv {x_0}^{(\frac{p+1}{2})^{t-1}} \mod p \end{aligned}$$

We can inductively repeat this until the process yields

$$y^{d_1} \equiv x_0 \mod p\,.$$

Similarly,

$$y^{d_2} \equiv x_0 \mod q\,.$$

Therefore, by the uniqueness of the Chinese Remainder Theorem solution and (1), we have

$$\overline{x_0} = x_0\,.$$

□

Since Bob has now calculated the original x_0, he can generate the Blum-Blum-Shub streamcipher by calculating $x_1, x_2, \ldots, x_t$, and recreating $b = b_1 b_2 \ldots b_t$, where $b_i = x_i \mod 2$. When b is calculated, $m = c \oplus b$ is the decrypted message.

Example 6 Here is an example of the encryption and decryption of a string of 1's and 0's implemented in Sage version 4.3.4. We will be using a private key $p = 499$ and $q = 547$ to keep it simple.

Sage

```
sage: from sage.crypto.public_key.blum_goldwasser
sage: import BlumGoldwasser
sage: bg = BlumGoldwasser(); bg
The Blum-Goldwasser public-key encryption scheme.
sage: p = 499; q = 547
sage: pubkey = bg.public_key(p, q); pubkey
272953
sage: prikey = bg.private_key(p, q); prikey
(499, 547, -57, 52)
sage: P = "10011100000100001100"
sage: C = bg.encrypt(P, pubkey, seed=159201); C
([[0, 0, 1, 0], [0, 0, 0, 0], [1, 1, 0, 0], [1, 1, 1, 0],
[0, 1, 0, 0]], 139680)
sage: M = bg.decrypt(C, prikey); M
[[1, 0, 0, 1], [1, 1, 0, 0], [0, 0, 0, 1], [0, 0, 0, 0],
[1, 1, 0, 0]]
sage: flatten(M)
[1, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1, 0, 0]
sage: M = "".join(map(lambda x: str(x), flatten(M))); M
'10011100000100001100'
sage: M == P
True
```

4.3 Complexity

The Blum-Goldwasser cryptosystem is a very efficient system both for encryption and decryption, comparable to the current standard, RSA encryption. During encryption, the Blum-Goldwasser cryptosystem actually performs faster than the standard in all but a few special cases. It is in linear time and based on generating the BBS stream cipher. In decryption, the initial calculations have a fixed number of steps, with additional steps based on message size. This makes it less efficient than RSA for short messages, but a quicker decryption for long messages. Decryption is also accomplished in linear time [Men].

4.4 Security

It has been discussed in previous papers on the subject that the Blum-Goldwasser cryptosystem is susceptible to a chosen-ciphertext attack. This attack is based on an attacker finding a ciphertext and its decryption without knowing the key. With this knowledge, the attacker may be able to determine the initial seed x_0, thus destroying the security of the system. Our model, however, takes a few steps to correct this. Firstly, we

have a random initial seed for each message we send. This means that with a chosen-ciphertext attack an adversary may gain knowledge of one initial seed, but the x_0 for a second message is one of $\frac{(p-1)(q-1)}{4}$ quadratic residues. Thus, the probability of having the same seed is extremely low. Another possible way to block this attack is with an authentication challenge. This would require challenging the attacker for the private key, similar to a password to gain network access. This should also work to prevent this attack.

5 Conclusion

The Blum-Goldwasser cryptosystem is an efficient system for both encryption and decryption with slight security deficiencies to a chosen ciphertext attack. Through my research I have tried to find a possible means to cover these deficiencies. We have theorized that changing the initial seed, x_0, in a pseudorandom method, the system may then be immune to these attacks. Based on the fact that there will be $\frac{(p-1)(q-1)}{4}$ different initial seeds, for a large n, the same message could be sent using each quadratic residue as the initial seed and each create a different ciphertext, c. Is this enough to classify this system secure? If so, is it financially feasible in comparison to RSA? This system has no public uses currently, but based on its efficiency and security, the Blum-Goldwasser cryptosystem could replace RSA in some daily uses.

6 Appendix: Sage code

This is the initial code on Python implemented by Prof Joyner and me. The code has since been cleaned and added to Sage 4.3.4 by Minh Van Nguyen. A special thanks to Minh for his hard work in the implementation.

Sage

```
"""
Functions written to implement the Blum-Goldwasser
cryptosystem, written as part of an honors math
project at the USNA in 2010.

As part of official goverment work, this is in the
public domain.

REFERENCES:
    Blum-Goldwasser cryptosystem
    http://en.wikipedia.org/wiki/Blum-Goldwasser_cryptosystem

    Menezes, Alfred; van Oorschot, Paul C.; and Vanstone, Scott
    A.  Handbook of Applied Cryptography. CRC Press, October
    1996. http://www.cacr.math.uwaterloo.ca/hac/
    (see chapter 8)

    M. Blum, S. Goldwasser, "An Efficient Probabilistic Public
```

```
    Key Encryption Scheme which Hides All Partial Information",
    Proceedings of Advances in Cryptology - CRYPTO '84, pp.
    289-299, Springer Verlag, 1985.

The implementation below is a blend of these.

AUTHORS:
    M. Hogan and D. Joyner (wdjoyner@gmail.com)

Last modified: 2010-02-06
"""

def num2bin(x):
    """
    Converts integer in range (1,255) to binary.

    EXAMPLES:
        sage: num2bin(100)
    """
    return [floor(x/2**(7-i))%2 for i in range(8)]

def string2ascii(m):
    """
    Converts a string of characters to a sequence of
    0's and 1's using the Python ord command.

    EXAMPLES:
        sage: string2ascii("usna2010")

    """
    L = []
    for a in m:
        L.append(ord(a))
    M = [num2bin(x) for x in L]
    return flatten(M)

def ascii2string(M):
    """
    M is a ciphertext message of 0's and 1's of length 8k.
    This returns a string of characters representing that
    list in ascii.

    EXAMPLES:
        sage: M = [0,1,0,1,0,1,0,0,0,1,0,0,0,0,1,0]
        sage: ascii2string(M)
        'BT'

    """
    m = len(M)
    k = int(m/8)
```

```
    S = []
    for i in range(k):
        s = sum([2**(7-j)*M[8*i+j] for j in range(8)])
        S.append(chr(s))
    sumS = ""
    for s in S:
        sumS = s + sumS
    return sumS


def carmichael(n):
    """
    The Carmichael function of a positive integer n,
    denoted \lambda(n) in the literature, is defined as
    the smallest positive integer m such that

    \[
    a^m \equiv a \pmod n,
    \]
    for every integer a that is both coprime to and smaller than n.
    In other words, in more algebraic terms, it defines the
    exponent of the multiplicative group of residues modulo n.

    EXAMPLES:
        sage: carmichael(4)
        2
        sage: euler_phi(4)
        2
        sage: carmichael(8)
        2
        sage: euler_phi(8)
        4
        sage: carmichael(10)
        4
        sage: euler_phi(10)
        4
        sage: carmichael(36)
        6
        sage: euler_phi(36)
        12
        sage: n = 100; a = 11; (power_mod(a,carmichael(n), n) - 1)%n
        0
        sage: n = 111; a = 11; (power_mod(a,carmichael(n), n) - 1)%n
        0

    REFERENCES:
        http://en.wikipedia.org/wiki/Carmichael_function

    """
    L = factor(n)
```

```
    if n == 2:
        return 1
    if n == 4:
        return 2
    t = len(L)      # the no. of dist. prime factors of n
    if t == 1:      #  n is a prime power
        p = L[0][0]
        if p == 2:  # so n is a power of 2
            return ZZ(n/4)
        if p>2:
            return ZZ((p-1)*n/p)
    powers = [L[i][0]**L[i][1] for i in range(t)]
    return lcm([carmichael(m) for m in powers])


def bbs(N, x0, L):
    """
    Implements the Blum-Blum-Shub pseudo-random number generator.

    INPUT:
      N - product of two primes, each congruent to 3 (mod 4)
      x0 - a seed, 1<x0<N   (possibly a square (mod N)?)
      L - the length of the sequence

    OUTPUT:
      a  pseudo-random sequence of 0's and 1's

    NOTE:
        Under some reasonable hypotheses, Blum-Blum-Shub [1]
        sketch a proof that the period of the BBS stream cipher
        is equal to carmichael(carmichael(N)). This is verified
        below in a few examples by using the Sage function
        lfsr_connection_polynomial (written by Tim Brock)
        which computes the connection polynomial of a linear
        feedback shift register sequence. The degree of that
        polynomial is the period.

    EXAMPLES:
        sage: p = next_prime(1015); q = next_prime(1100)
        sage: p%4 == 3; q%4 == 3
        True
        True
        sage:  bbs(p*q, 999, 10)
        [1, 1, 1, 1, 1, 1, 0, 0, 0, 1]
        sage: carmichael(carmichael(7*11))
        4
        sage: bbs(7*11, 13, 16)
        [1, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1, 0]
        sage: s = [GF(2)(x) for x in bbs(7*11, 13, 60)]
        sage: lfsr_connection_polynomial(s)
```

```
        x^3 + x^2 + x + 1
        sage: carmichael(carmichael(11*23))
        20
        sage: s = [GF(2)(x) for x in bbs(11*23, 13, 60)]
        sage: lfsr_connection_polynomial(s)
        x^19 + x^18 + x^17 + x^16 + x^15 + x^14 + x^13 + x^12\
        + x^11 + x^10 + x^9 + x^8 + x^7 + x^6 + x^5 + x^4 + x^3 + x^2 + x + 1

    REFERENCES:
        [1] Lenore Blum, Manuel Blum, and Michael Shub. "Comparison
        of two pseudo-random number generators", Advances in
        Cryptology: Proceedings of Crypto '82.
        http://dsns.csie.nctu.edu.tw/research/crypto/HTML/PDF/C82/61.PDF
    """
    if L<0: return []
    b = [x0%2] # = [x0**2%2]
    R = IntegerModRing(N)
    x0 = R(x0)
    for i in range(L-1):
        x1 = x0**2
        b.append(ZZ(x1)%2)
        x0 = x1
    return b

class BGCryptosystem():
    """
    Class implementing Blum-Goldwasser cryptosystems.

    EXAMPLES:

    """
    def __init__(self, public_key):
        self._public_key = public_key

    def public_key(self):
        """
        Returns the public key of the cipher.

        EXAMPLES:
            sage: p = next_prime(1015); q = next_prime(1100)
            sage: BG = BGCryptosystem(p*q)
            sage: BG.public_key()
            1123957

        """
        return self._public_key

    def __str__(self):
        """
        Print method.
```

```
        EXAMPLES:
            sage: p = next_prime(1015); q = next_prime(1100)
            sage: BG = BGCryptosystem(p*q)
            sage: print BG
            BGCryptosystem(1123957)

        """
        return "BGCryptosystem(%s)"%self.public_key()

    def __repr__(self):
        """
        Display method.

        EXAMPLES:
            sage: p = next_prime(1015); q = next_prime(1100)
            sage: BG = BGCryptosystem(p*q)
            sage: BG
            Blum-Goldwasser cryptosystem with public key
           1123957.

        """
        return "Blum-Goldwasser cryptosystem with public key
        %s."%self.public_key()

    def random_key(self):
        """
        Returns a random key to be used by the cipher.

        EXAMPLES:

        """
        N = self.public_key()
        return floor(rand()*N)

    def encrypt(self, M, r):
        """
        Implements the Blum-Goldwasser public key encryption
        algorithm.

        INPUT:
          M - a cleartext string (the message)
          r - a random number 1 < r < N

        EXAMPLES:
            sage: p = next_prime(1015); q = next_prime(1100)
            sage: p%4 == 3; q%4 == 3
            True
            True
            sage: BG = BGCryptosystem(p*q)
```

```
        sage: c, y = BG.encrypt("Hello World", 999)
        sage: len(c)
        88
        sage: y
        760299

    TODO: Check if r can be defined inside the method,
          and removed as an input parameter.

    """
    N = self.public_key()
    m = string2ascii(M)
    L = len(m)
    R = IntegerModRing(N)
    x0 = ZZ(R(r**2))
    b = bbs(N, x0, L)
    return [(b[i]+m[i])%2 for i in range(L)],
    power_mod(x0, 2**L, N)

def decrypt(self,c,y,p,q):
    """
    Implements the Blum-Goldwasser decryption algorithm.

    INPUT:
      b - a list of 0's and 1's of length L (the ciphertext)
      y - a number 1 < y < N = p*q
      p, q - primes, each congruent to 3 (mod 4)

    OUTPUT:
      a string (the cleartext message)

    EXAMPLES:
        sage: p = next_prime(1015); q = next_prime(1100)
        sage: p%4 == 3; q%4 == 3
        True
        True
        sage: BG = BGCryptosystem(p*q)
        sage: c, y = BG.encrypt("Hello World", 999)
        sage: BG.decrypt(c,y,p,q)
        'Hello World'

    """
    N = p*q
    if not(N == self.public_key()):
        raise ValueError("Your private key (%s,%s)
        are incorrect"%(p,q))
    L = len(c)
    pqgcd, a, b = xgcd(p,q)
    #rp = power_mod(y, int((p+1)/4)**L, p)
    #rq = power_mod(y, int((q+1)/4)**L, q)
```

```
        #pi = power_mod(p, -1, q)
        #qi = power_mod(q, -1, p)
        #x0 = (q*qi*rp + p*pi*rq)%N
        d1 = power_mod(int((p+1)/4), L, p-1)
        d2 = power_mod(int((q+1)/4), L, q-1)
        u = power_mod(y, d1, p)
        v = power_mod(y, d2, q)
        R = IntegerModRing(N)
        x0 = ZZ(R((v*a*p+u*b*q)))
        b = bbs(N, x0, L)
        d = [(x[0]+x[1])%2 for x in zip(b,c)]
        s = ascii2string(d)
        reverse_s = s[::-1]
        return reverse_s
```

References

[BBS] L. Blum, M. Blum, and M. Shub. *A simple unpredictable pseudo random number generator*. SIAM Journal on Computing, Volume 15, Issue 2, pp. 364-383. Society for Industrial and Applied Mathematics, Philadelphia, 1986.

[BG] M. Blum and S. Goldwasser. *An Efficient Probabilistic Public-Key Encryption Scheme Which Hides All Partial Information*. In Proceedings of CRYPTO 84 on Advances in Cryptology, pp. 289-299, Springer, 1985. the decry

[Bro] T. Brock. *Linear Feedback Shift Registers and Cyclic Codes in SAGE*. Rose-Hulman Undergraduate Mathematics Journal, Vol. 7, Issue 2, 2006.

[GM] S. Goldwasser and S. Micali. *Probabilistic Encryption and How to Play Mental Poker Keeping Secret All Partial Information*. In Proceedings of the Fourteenth Annual ACM Symposium on theory of Computing, pp. 365-377, ACM, New York, 1982.

[Gol] S. Golomb, *Shift register sequences*. Aegean Park Press, Laguna Hills, Ca, 1967

[Men] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996.

[S] W. Stein, *Sage: Open-source Mathematical Software*, version 4.3.4, 2010.