# Applying cryptographically secure pseudorandom numbers to one-time pad cryptography

Juan Velasco
North Dakota State University, Computer Science Department

## Abstract

In recent years there has been an increase in the usage and manufacturing of low-cost ARM GNU/Linux computing devices such as the Raspberry Pi W Zero. These low-cost computing devices have consistently been integrated into the Internet of Things (IoT) network since they come with equipped with networking capabilities. As such, there exists a need to develop lightweight cryptosystems for these resource-constrained devices. And indeed, quite a bit of work has already been done in this area of research, resulting in several lightweight symmetric encryption algorithms. There are two main types of encryption techniques used today, namely, symmetric-key algorithms and asymmetric-key algorithms. However, there also exits a third type of encryption technique called a "one-time pad" (OTP) that is habitually ignored despite the fact it provides perfect secrecy [5], mainly because it is often impractical to implement. In this paper we propose practical implementation of a one-time pad using the Blum-Blum Shub (B.B.S) pseudorandom number generator, which is proven to be cryptographically secure [3] [6] [15]

## Introduction

Encryption algorithms play a crucial role in today's tech-savvy society. They are usually used "in the background" when performing a variety of online activities such as electronic payments, text messaging, online banking and so on. In modern cryptography, there are two main types of encryption algorithms: symmetric-key algorithms and asymmetric-key algorithms. The most commonly used symmetric-key encryption algorithms are AES (Rijndael), DES, 3DES, RC2, Blowfish, and RC6. All these algorithms were designed more than ten years ago in an era where the Internet of Things (IoT) did not exist. As such, these encryptions algorithms do not consider the performance needs of super-inexpensive, super-small computers such as the Raspberry PI Zero W. In the realm of asymmetric-key algorithms, the most commonly used algorithms are: RSA and ElGamal. It is well known that asymmetric-key algorithms are almost 1000 time slower than symmetric-key algorithms, because they require more computational processing power [1] [2]. These performance issues make asymmetric-key algorithms also not very suitable for IoT devices. There has also been a lot of work done in researching lightweight encryption algorithms specifically designed for computing devices with constrained resources. Some lightweight encryption algorithms are Tiny Encryption Algorithm, ChaCha20, Speck and Rabbit. In fact, ChaCha20 has already been selected as a replacement for RC4 in TLS [4]. Nonetheless, in this paper we will explore a third type of encryption technique known as a one-time pad, which when implemented properly, is indecipherable from a purely mathematical perspective.

Despite the fact OTP offers perfect secrecy [5], it is often disregarded under the basis that it is impractical to implement. Per definition, OTP requires: 1) The length of the key to be of equal or greater length than the message to be encrypted. 2) The key (also known as pad value) to be truly random. 3) The key must be generated and exchanged in a secure way. 4) The key can only be used once, hence the name "one-time" pad. The fact OTP requires generating a lot of true random data and the inability of IoT devices to generate such data without special-purpose hardware random number generators (TRNG) does pose a serious problem for all computers without a TRNG. The way this has been historically solved is by using software-generated pseudorandom numbers (PRNG) [9] [16]. For cryptographic purposes, a PNRG must be *cryptographically secure* [9]. For our proposed implementation, we opted to use the Blum-Blum Shub (B.B.S) pseudorandom number generator (PRNG), which is proven to be cryptographically secure.

Using B.B.S PRNG addresses most of the problems argued against the usage of OTP. B.B.S PRNG, being cryptographically secure, can generate a lot of pseudorandom numbers which are indistinguishable by polynomial time statistical tests from numbers generated by a TRNG [3]. Before using OTP, both users need to copy of the key. The biggest concern when using OTP is how to securely exchange this key. If Alice sends is to send 1 kilobyte worth of encrypted data, she must also somehow transmit a 1 kilobyte key. B.B.S PRNG has the practical property of being a trapdoor generator [7]. This means there is no need to transmit the full key, Alice can just send the inputs (secret seeds) used to generate the key, and Bob can generate the key himself and decrypt the message. From now on we shall refer to this key as pad value and we will refer to the secret seeds used to generate the pad value as the key. Key exchange is discussed on section 4.3, but it is sufficient to say that when it comes to OTP security, an "offline" key exchange is still the most secure and practical approach. As discussed, using B.B.S PRNG makes it practical to implement OTP since it allows us to generate cryptographically secure pseudorandom data and users only need to exchange the much smaller secret seeds used to generate the pad value.

Despite OTP encryption not being very popular for commercial systems, OTP have been used for very critical communications throughout history, particularly in SIGSALY [11] which protected the highest level of voice communications between the Allies in WWII, and the Washington-Moscow hotline [12] which connected Russia leaders and the US. In fact, OTP encryption is being used today for keeping the most sensitive US government communications secret. The US government employs heavily armored trucks and armed guards to transport random numbers (the key) between the Pentagon and remote locations [13]. But individuals who do not possess the resources governments do should still be able to secure their communications the way the military and government do.

While it is true that OTP makes little sense for mass market encryption protocols like TLS, for personal uses, a pure software-based OTP solution can still work for small groups of people wishing to have long term security for the confidentiality of their messages. Individuals do not have the resources needed to employ OTP the way governments have been doing it for decades. And solutions to bring OTP to normal everyday people have been very scarce. As of the writing of this paper, *Jericho Comms* [14] is the only pure software-based OTP solution we have found that aims to bring OTP encryption to the public.

*Jericho Comms* generates random data from pictures taken in RAW mode and it even allows users that do not have cameras with that feature to generate random data using a simple webcam. Nonetheless, that solution still leaves people in the most resource-constrained situations helpless as there are requirements that may not be possible to meet every time (such as the usage of cameras to generate random data). The proposed implementation can bring OTP encryption to that market as it does not require any specialized hardware. It relies on a mathematical formula to generate cryptographically secure pseudorandom data. A computational device smaller than the size of a credit card (like the Raspberry pi W Zero) can generate cryptographically secure pseudorandom bits to encrypt or decrypt any messages on the fly. The usage of digital storage devices to exchange and/or store the key is also not an essential requirement for the proposed implementation to work. A 64-bit number can be written on a small piece of paper and that number could generate pad values that are a few terabytes long (See section 6.2.5).

## 2. Background

This section provides background information related to prior work in the fields of IoT devices and one-time pad cryptography. First, we discuss the theory behind a one-time pad. This is followed by an overview of the cryptographic security behind B.B.S PRNG.

### 2.1 Theory Behind a One-time Pad

OTP encryption is a very old and quite simple technique to understand. The sender and receiver both agree on a random key or "pad", which *has* to be as long or longer than the message to be sent. The sender encrypts the message (usually by XORing the message with the pad value) and sends it. Only the receiver can decrypt the message since he is the only one who possess the correct key. After decrypting the message, the key gets destroyed and should never be used again.

Example:

- Alice and Bob agree on random key $K$ which is as long as message $M$
- Alice creates ciphertext $C = M \oplus K$ sends it to Bob and destroys $K$
- Bob decrypts the message $M = C \oplus K$ and destroys $K$

OTP encryption may seem simple and straightforward, but the following rules are mandatory to ensure the security of the message. Not following these rules will always result in weakened security.

1. The OTP key should consist of truly random data
2. The OTP key should be as long, or longer than the message to be sent
3. Only two copies of the OTP key should exist
4. The OTP key should be used only once
5. Both copies of the OTP key are destroyed immediately after use.

One-time pad encryption is an equation with two unknowns, which is mathematically unsolvable. It is therefore impossible for any eavesdropper to decrypt or break the message without the proper pad value, regardless of any existing or future cryptanalytic technology, infinite computational power or infinite time [9]. When the pad value is truly random, and each part of the pad value is independent of every other part, OTP yields perfect secrecy according to [5]. From [5]:

**"**An encryption scheme $(Gen, Enc, Dec)$ over a message space $M$ is **perfectly secret** if for every probability distribution over $M$, every message $m \in M$, and every ciphertext $c \in C$ for which $Pr[C = c] > 0$:

$$\Pr[M = m \mid C = c] = \Pr[M = m]\text{"}$$

In other words, an attacker with a ciphertext at hand that is perfectly secret will not learn anything new about the message that was not already known. It is important not to fall in the mindset of believing that OTP will hide all information about the underlying message. Suppose an attacker intercepted a ciphertext of length 22. Through a brute force attack the attacker will end up generating every possible pad value of length 22. For example, the attacker may find a pad value that generates the message *"he called her adorable"* but he may also find another pad value that generates the message "*she said you are cruel".* These results may confuse people as it is commonly believed OTP is uncrackable in the sense that it hides all information about the underlying message, which is not the case as observed. However, the main point here is that after using a brute force attack on this hypothetical ciphertext of length 22, the attacker is no closer to knowing *which* of these solutions is the right one—this is what is meant by OTP yielding perfect secrecy.

Manual OTP ciphers (the usage of OTPs without using computers) are still being used today for sending secret messages via number stations or one-way voice links that may be heard on short-wave radio bands. For a more detailed look of the one-time pad cipher and its history with several examples, please refer to the paper *Secure Communications with the One Time Pad Cipher,* by Dirk Rijmenants [9].

### 2.2 Overview of the Blum-Blum-Shub pseudorandom number generator

B.B.S. PRNG was proposed in 1986 by Lenore Blum, Manuel Blum and Michael Shub [3]. It is based on a simple mathematical formula: $x_{i+i} = x_i^2\ (mod\ n)$. It outputs a sequence by repeatedly reducing squares modulo the product of two *Blum primes*.

Definition 1: a *Blum Prime* number is a prime congruent to 3 modulo 4.

B.B.S PRNG algorithm is as follows:

- Generate $p$ and $q$, such that $p, q$ are equal length ($|p| = |q|$) distinct primes congruent to 3 modulo 4. $p \equiv q \equiv 3\ (mod\ 4)$ such that $gcd(p, q) = 1$
- Compute $n = p * q$
- Choose a random seed $x_0 \in \{2, \dots n-1\}$ such that $gcd(s, n) = 1$
- The sequence is defined as $x_{i+1} = x_i^2\ (mod\ n)$ and $b_i = parity(x_i)$
- The output is $b_0, b_1, b_2, \dots$

Thus, with inputs $(N, x_0)$ B.B.S PRNG outputs the pseudorandom sequence of bits $b_0, b_1, \dots$ obtained by setting $x_{i+1} = x_i^2\ (mod\ n)$ and extracting the bit $b_i = parity(x_i)$.

The security of B.B.S PRNG depends on assumption of the intractability of the quadratic residuosity problem [3]. Vazirani and Vazirani also proved it to be cryptographically secured under the assumption that integer factorization is intractable [8]. Generating one pseudo-random bit $\{b_i\}$ requires $n^2$steps (one modular multiplication). This makes this generator slow compared to normal non-cryptographically secure PRNGs, however, performance can be increased by using it on hardware devices which already have circuitry for performing modular multiplication. There are also efficient techniques for implementing modular multiplication in software. Furthermore, research by Vazirani and Vazirani shows that up to log (log($n$)) bits can be securely extracted per iteration, where $n$ is the size of the modulus. This can increase the performance significantly.

**Example 1**

- Let $p = 7,\ q = 19$
- Then $n = p \cdot q = 7 \cdot 19 = 133$
- Let $x_0 = 100$
- The sequence is:
  - $x_1 = 100^2\ (mod\ 133) = 25$ $\qquad b_1 = parity(x_1) = 1$
  - $x_2 = 25^2\ (mod\ 133) = 93$ $\qquad b_2 = parity(x_2) = 1$
  - $x_3 = 93^2\ (mod\ 133) = 4$ $\qquad b_3 = parity(x_3) = 0$
  - $x_4 = 4^2\ (mod\ 133) = 16$ $\qquad b_4 = parity(x_4) = 0$

  - ○ $x_5 = 16^2\ (mod\ 133) = 123 \qquad b_5 = parity(x_5) = 1$
- Output: $b_0, b_1 \dots b_5 = 011001 \dots$

Notice that calculating $x_6 = 123^2\ (mod\ 133) = 100$ gives back the seed $x_0$. This is because the sequences generated are periodic, with a period "usually" equal to $\lambda(\lambda(n))$ [3]. In the example above, the sequence generated with input $(n, x_0) = (133, 100)$ = 100, 25, 93, 4, 16, 123… has a period of 6. Here, $\lambda(\lambda(133)) = 6$. Being able to determine the period of a sequence is important when encrypting a message since the generated sequence *must* be of equal or greater length than the message to be encrypted (at least in the case where only one bit per iteration is extracted).

Lenore Blum, Manuel Blum and Michael Shub proposed using B.B.S PRNG for *public-key cryptography* by having Alice construct and publish a number $N_A$, her *public key*, with the prescribed properties: $N_A = P_A * Q_A$, where $P_A$ and $Q_A$ are distinct equal length randomly chosen primes both congruent to 3 mod 4. Alice would then keep the primes $P_A$ and $Q_A$ as her *private key*.

Supposed Bob wants to send a $k$-bit message $\vec{m} = (m_1, .. m_k)$, using Alice's public key he would construct a sufficiently large pad value by selecting a random seed $x_0 \in \{2, \dots N_A - 1\}$ such that $gcd(s, N_A) = 1$. He would then use B.B.S PRNG with input $(N_A, x_0)$ to generate the pad value $\vec{b}= (b_1, \dots b_k)$. Bob then encrypts the message, $\vec{m} \oplus \vec{b} = (m_1 \oplus b_1, \dots m_k \oplus b_k)$ and sends BOTH, the encrypted message and $x_{k+1}$, the next number in the pseudo random sequence over public channels.

Alice with her private key $P_A$ and $Q_A$ would then be able to efficiently compute the pad value backwards $b_k, b_{k-1}, \dots\ b_1$. From that, by XORing $\vec{b}$ with the encrypted message she decrypts the message $\vec{m}$.

Our implementation proposal differs from the above in that we do not advocate for Bob to send the encrypted message AND $x_{k+1}$ to Alice. There are two main reasons for this. First, computing the pad value backwards from $x_{k+1}$ with factors $P$ and $Q$ of $N$ is around three times slower than simply computing the pad value from $x_0$. Second, sending $x_{k+1}$over public channels potentially compromises the security of the OTP. $x_{k+1}$ would be sent as plaintext which means an attacker could possess a value piece data that can give insightful information about the pseudo random bit sequence used to encrypt the message. An attacker can simply take Alice's public key $N_A$ and Bob's $x_{k+1}$ (both which were sent over public channels) and keep on computing the sequence. Since sequences produced by B.B.S PRNG have a fix length and are periodic, an attacker can compute the whole sequence with input $(N_A, x_{k+1}\ )$ and use that to efficiently decrypt the message.

## 3. Algorithm description

This section provides information about the proposed algorithm implementation based on the prior work of [3]. First, a general overview is presented. Then, specific elements of the approach are discussed.

### 3.1 Overview

The encryption of a message $m$ using a one-time pad value $b$ is defined as:

$$Enc(m, b) = m \oplus b$$

Encryption takes place at the bit level. Suppose Bob wants to send a $k$-bit message $\vec{m} = (m_1, .. m_k)$, using B.B.S PRNG, a pseudorandom bit sequence $\vec{b} = (b_1, \ldots b_k)$ is generated as described in example 1. The ciphertext is $C = Enc(\vec{m}, \vec{b}) = \vec{m} \oplus \vec{b} = (m_1 \oplus b_1, \ldots m_k \oplus b_k)$.

The key is defined as the set of integers $(N, x_0)$, where $N$ is the product of two distinct large prime numbers $p, q$ such that $(|p| = |q|), p \equiv q \equiv 3 \ (mod\ 4)$ and $gcd(p, q) = 1$. $x_0$ is a seed in the range $\{2, \ldots n - 1\}$ such that $gcd(S, N) = 1$.

Pad values are generated using B.B.S PRNG with the key as input. A key can produce very small amounts of pseudorandom data (i.e. 6 bits) or very large amounts (i.e. 4,000,000,000+ bits). The number of bits generated depends the length of the sequence produced by the key $(N, x_0)$, therefore, $N$ should be chosen so that the number of bits generated is more than the bits in the message. See section 3.2.

The proposed implementation adheres to the original rules of OTP in that only the encrypted message should be sent over public channels and nothing else. However, as mention in the introduction of this paper, using B.B.S PRNG means there is no need to exchange the full pad value. Alice and Bob only need to agree on the key $(N, x_0)$ used to generate the pad value. Suppose Alice used $N_A = P_A * Q_A = 133$ and $x_{0_A} = 100$. The key would be the secret set of integers $(N_A, x_{0_A})$ or $(100, 133)$. With this key, Alice and Bob can both generate the same pad value and decrypt/encrypt any message.

### 3.2 Generating pseudorandom bits for a one-time pad

The most important aspect when generating pseudorandom bits using B.B.S PRNG is the usage of a sufficiently large modulo number $N$ since this will dictate the maximum number of bits that can be generated with $N$. If the number of bits generated with $N$ is less than the length of the message, then decrypting the message becomes trivial. Fortunately, it is possible to determine the length of any given pseudorandom sequence generated by B.B.S PRNG without computing the whole sequence if and only if $N$ is chosen carefully. Suppose $N$ is of the form prescribed in section 3.1, [3] proved that if $N$ is chosen so that $ord_{\frac{\lambda(N)}{2}}(2) = \lambda(\lambda(N))$ then the sequence generated by $N$ is guaranteed to be of length $\lambda(\lambda(N))$.

**Example 2:**

- Let $p = 7,\ q = 19$
- Then $n = p \cdot q = 7 \cdot 19 = 133$
- Since $ord_{\frac{\lambda(133)}{2}}(2) = 6 = \lambda(\lambda(133))$ the maximum number of bits generated by B.B.S PRNG with input key $(133, 100)$ is 6
- Let $x_0 = 100$
- Calculating the sequence, we get:
  - $x_1 = 100^2\ (mod\ 133) = 25$ $z_1 = parity(x_0) = 1$
  - $x_2 = 25^2\ (mod\ 133) = 93$ $z_2 = parity(x_1) = 1$
  - $x_3 = 93^2\ (mod\ 133) = 4$ $z_3 = parity(x_2) = 0$
  - $x_4 = 4^2\ (mod\ 133) = 16$ $z_4 = parity(x_3) = 0$
  - $x_5 = 16^2\ (mod\ 133) = 123$ $z_5 = parity(x_4) = 1$
  - $x_5 = 123^2\ (mod\ 133) = 100 = x_0$ thus, the sequence repeats
- Bits: 011001

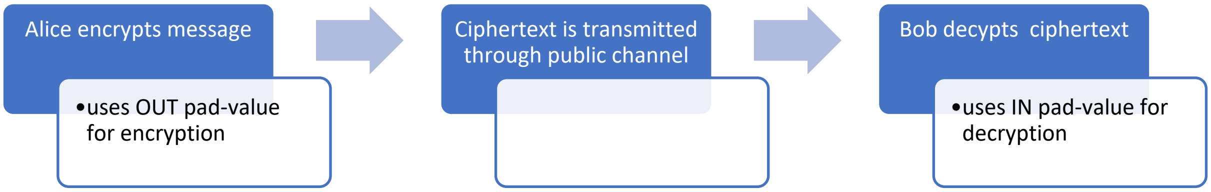
## 4. System Description

This section provides information about how the proposed OTP implementation can be implemented into a practical cryptosystem system. First, we discuss a one-way communication system, then a two-way communication system. The system uses a set of two keys, each key will produce two identical pad-values, one pad-value is called the IN pad-value, and the other (identical) pad-value is called the OUT pad-value.

### 4.1 One-way Communication System

To establish a one-way communication channel, the sender & receiver only need one key, which will produce one *OUT* pad-value for the sender and an identical *IN* pad-value for the receiver.

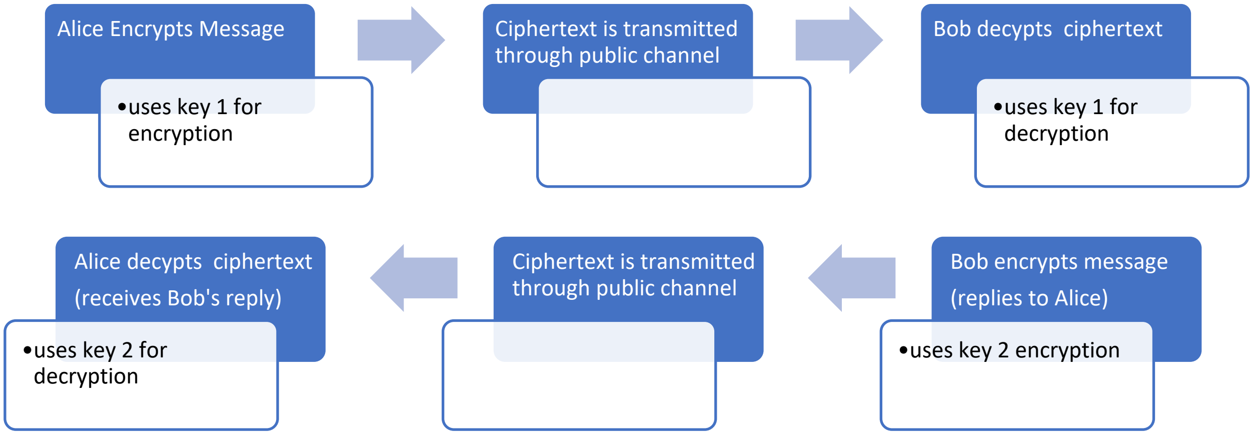
Figure 1:


Alice encrypts message
•uses OUT pad-value for encryption
Ciphertext is transmitted through public channel
Bob decypts ciphertext
•uses IN pad-value for decryption

The usage of multiple IN pad-value copies to allow more than one person to receive the message is possible but not advisable. Multiple copies of the OUT pad-value should *never* be use as that will lead to simultaneous use of the same pad-value, which opens the door for known plaintext attacks. While systems like AES and RSA are believed to be secured against known plaintext attacks, OTP is completely insecure against them, which is why a key should be used once and never again.

### 4.2 Two-way communication system

To establish a two-way communication channel, the system needs two keys, which will produce 2 different pad-values. Key one generates an OUT pad-value for Alice and an IN pad-value for Bob. Key two generates an OUT pad-value for Bob and an IN pad-value for Alice. A single key should never be used for a two-way communication channel.


Alice Encrypts Message
•uses key 1 for encryption
Ciphertext is transmitted through public channel
Bob decypts ciphertext
•uses key 1 for decryption
Alice decypts ciphertext (receives Bob's reply)
•uses key 2 for decryption
Ciphertext is transmitted through public channel
Bob encrypts message (replies to Alice)
•uses key 2 encryption

### 4.3 Key exchange

In order to retain the perfect secrecy provided by OTP, an "offline" exchange of the key is required. This offline key exchange can be, for example, two parties meeting physically and exchanging USB drives containing the keys. It is not advisable to exchange keys online, *even if the key is encrypted using another cipher such as RSA or AES*. This is because exchanging the keys online reduces the security of OTP to the security provided by this cipher. Quantum key distribution (QKD) is the only provable way to exchange keys securely without having to physically meet. However, with QDK the problem moves from a key distribution problem to an engineering problem. The QKD system has to be completely reliable, not have any side channel attacks, and be unsusceptible to induced errors. For now, it is still recommended to exchange keys offline.

### 4.4 Key Generation

The key is defined as the set of integers $(N, x_0)$ where $N$ is the product of two distinct prime numbers $p, q$ such that $(|p| = |q|), p \equiv q \equiv 3\ (mod\ 4)$ and $gcd(p, q) = 1$. $x_0$ is a seed in the range $\{2, \dots n - 1\}$ such that $gcd(N, x_0) = 1$. One important question is: *how big should $P$ & $Q$ be?* Keys should be chosen so that $P$ & $Q$ are at least 32 bits long each. Sometimes the product $N$ will not be necessarily 64 bits long, however, this $N$ should still produce several hundred megabytes of pseudorandom bits. See section 6 for more concrete examples & analysis on the selection of $P$ & $Q$.

### 4.5 Key Usage

There are several ways in which the system can utilize keys to encrypt communications. The most straight-forward method is to use a new key for each encrypted message. This, however, is not optimal. Given the severe limitations in key exchange (i.e. having meet physically), it is important utilize keys in the most efficient way possible. Suppose a key can generate up to 1,000,000 pseudorandom bits, if the message to be encrypted is only 20,000 bits, 98% of the bits produced by this key are wasted if the key is disregarded after use.

A better way to use keys is to utilize as much bits produced by a key as possible, since those bits are cryptographically secure, using them for consecutive messages will not affect the security of OTP.

**Example 3:**

- Suppose key 1 generates bits: 11011000110100111111000100011101001010011101101
- Message #1 bits are: 1010110100001001
- OTP bits left after encryption: 1111000100011101001010011101101
- Message #2 bits are: 10110101000110001101001000
- OTP bits left after encryption: 01101
- Disregard key 1, use next key for next encrypted message

Depending on the length of the messages to be encrypted, a single key could allow the encryption of thousands of messages. The system only needs to know the maximum number of bits that can be generated by a given key and keep track of the last pseudorandom number generated in order to continue generating bits for the next message. Supposed a key can generate 1000 GB of pseudorandom data. That is enough to encrypt one 1kB message per second for over 2500 years—and the key can be written in a tiny piece of paper.

Although the approach proposed above increases the efficiency of the system dramatically, there is another "optimization" that could be made. Recall that B.B.S PRNG produces a periodic sequence given a key $(N, x_0)$. Example 3 disregards a key completely after most or all of its bits are used. This is approach wastes the potential of $N$ since instead of completely disregarding the key, the system could simply choose another seed $x_0 \in \{2, N-1\}$ *such that* $\gcd(S, N) = 1$. Choosing another seed allows the system to generate completely new pseudorandom bits making the use of a single $N$ as efficient as possible.

**Example 4**

- Suppose key $K_1 = (N_1, S_1) = (133, 100)$ is used to generate sequence $X_1$
- Let $X_1 = \{25, 93, 4, 16, 123\} = B_1 = parity(X_1) = \{1,1,0,0,1\}$
- Instead of disregarding $K_1$ completely, only disregard $S_1$ and keep $N_1$
- Let $S_2 = 6$ be the new seed
- Let $K_2 = (N_1, S_2) = (133, 6)$ and use that to generate sequence $X_2$
- Calculating the sequence $X_2$, we get:
  - $x_1 = 6^2\ (mod\ 133) = 36$ $z_0 = parity(x_0) = 0$
  - $x_2 = 36^2\ (mod\ 133) = 99$ $z_1 = parity(x_1) = 1$
  - $x_3 = 99^2\ (mod\ 133) = 92$ $z_2 = parity(x_2) = 0$
  - $x_4 = 92^2\ (mod\ 133) = 85$ $z_3 = parity(x_3) = 1$
  - $x_5 = 85^2\ (mod\ 133) = 43$ $z_4 = parity(x_4) = 1$
  - $x_6 = 43^2\ (mod\ 133) = 120$ $z_5 = parity(x_4) = 0$
  - $x_7 = x_1$
- Bits: 010110

As stated in section 3.2, the length of a sequence generated with $N = 133$ is $\lambda(\lambda(N)) = 6$. It can be said that $\lambda(\lambda(N))$ is the maximum number of bits that can be generated by $N$ with a single seed $x_0$. Using $N$ with multiple seeds allows for the exponential increase in the maximum number of bits that be generated with $N$. In the case where $N = 133$, the maximum number of bits that can be possibly generated is 234 (instead of just 6 when using a single seed). This is because $N = 133$ can produce 39 unique sequences of length 6. Nonetheless, it remains unclear whether using a single $N$ with multiple seeds is as secure as using a single seed with a single $N$. For now, it is recommended to only use a single seed with a single $N$.

## 5. Experimental Methods

This section describes the experimental methods used in this research, including the code used to generate pseudorandom bits, the sample gathering, and the tests for randomness.

### 5.1 Code and Testing

In order to generate the desired bit sequence, B.B.S PRNG was implemented in C++. Testing was conducted in a normal desktop computer running a Linux distribution (Intel Core i7 4770S, 16 GB Ram and a 256GB SSD).

### 5.2 Generating the one-time pad values

The testing program takes two distinct Blum prime numbers $p$ and $q$. The modulo $n$ is calculated to be $n = p \cdot q$. Although up to $log(long(n))$ bits per iteration can be extracted securely, for this testing we only extracted a single bit as the original paper on B.B.S PRNG first proposed. Next, the selected test is run on the output of B.B.S PRNG. All tests are run with seed $x_0 = 4$.

### 5.3 File Encryption: Performance Test

This test calculates how long it takes to encrypt a file with $p$ and $q$ of 16, 20, 32 and 50 bits. The purpose of this tests is to determine if the size of $p$ and $q$ affects encryption speed. Five test files are used. The sizes of the files are: 10kB (1000 bytes), 100kB, 1MB, 10MB and 100MB. Each file is encrypted 100 times with $p$ and $q$ of each specified size. The average time for all 100 encryption rounds is then taken and compared for each file and each size of $p$ and $q$. For this test, the algorithm was implemented using primitive data types as the system allowed for the use of `unsigned __int128` since the testing computer has an integer mode wide enough to hold 128 bits.

### 5.4 Blum Primes Length Test

This test aims to see how big are the sequences generated by $p$ and $q$ of 8, 10, 16, 20 and 32 bits. Ten different Blum primes of each size are selected, and their sequence length is compared. The sequence length is computed by calculating each number in the sequence until the sequence repeats. The purpose of this test is to see if Blum primes of the same bit size produce sequences of the same length and to visualize the difference in sequences generated by Blum primes of different sizes. This test will also illustrate how big sequences generated by B.B.S PRNG can be.

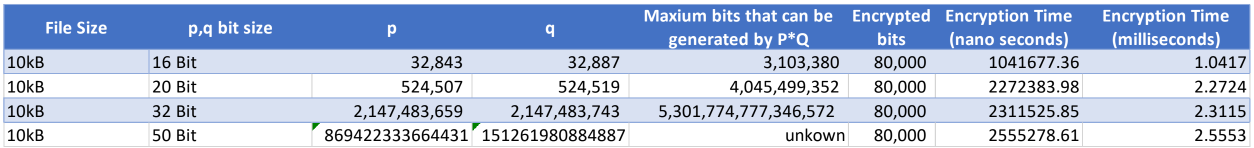
## 6. Results & Analysis

In this paper, five text files were encrypted. Only encryption was tested because the decryption process is exactly the same as the decryption process. Unlike other cryptographic algorithms in which the decryption process is the reverse of the encryption process, with OTP a single algorithm does both. These five files are of size: 10kB, 100kB, 1MB, 10MB, and 100MB. The Blum primes $p$ and $q$ used are listed in each test.

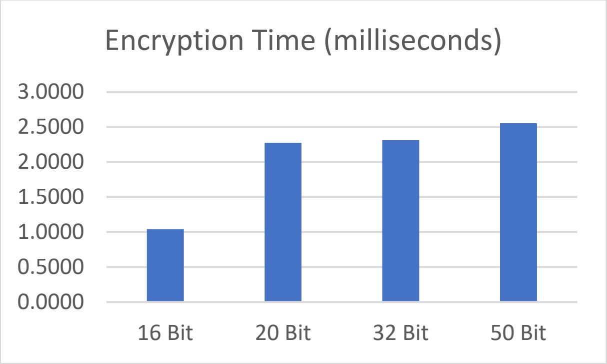
### 6.1 File Encryption: Performance Test

This section analyzes the encryption speed of a text file with Blum primes $p$ and $q$ of 16, 20, 32 and 50 bits. The seed is number 4.

### 6.1.1 Encrypting a 10kB text file

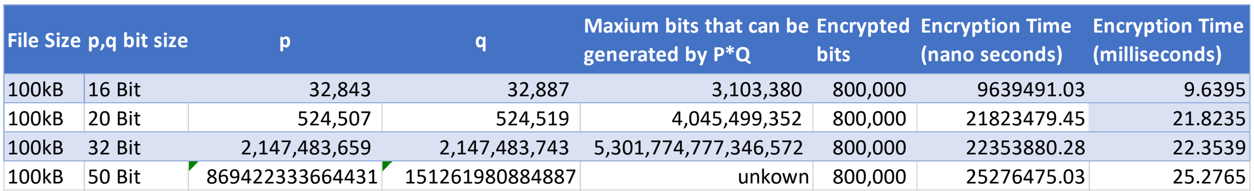


| File Size | p,q bit size | p | q | Maxium bits that can be generated by P*Q | Encrypted bits | Encryption Time (nano seconds) | Encryption Time (milliseconds) |
|---|---|---|---|---|---|---|---|
| 10kB | 16 Bit | 32,843 | 32,887 | 3,103,380 | 80,000 | 1041677.36 | 1.0417 |
| 10kB | 20 Bit | 524,507 | 524,519 | 4,045,499,352 | 80,000 | 2272383.98 | 2.2724 |
| 10kB | 32 Bit | 2,147,483,659 | 2,147,483,743 | 5,301,774,777,346,572 | 80,000 | 2311525.85 | 2.3115 |
| 10kB | 50 Bit | 869422333664431 | 151261980884887 | unkown | 80,000 | 2555278.61 | 2.5553 |

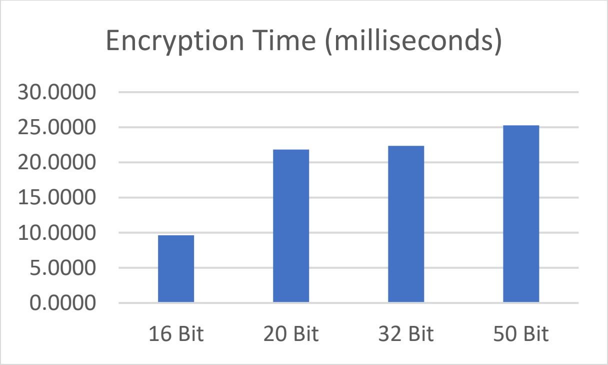


From this first test, it seems to be clear that the length of the key does affect encryption speed significantly.

### 6.1.2 Encrypting a 100kB text file

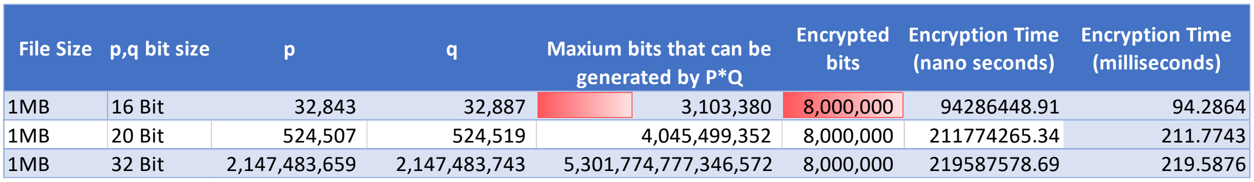


| File Size | p,q bit size | p | q | Maxium bits that can be generated by P*Q | Encrypted bits | Encryption Time (nano seconds) | Encryption Time (milliseconds) |
|---|---|---|---|---|---|---|---|
| 100kB | 16 Bit | 32,843 | 32,887 | 3,103,380 | 800,000 | 9639491.03 | 9.6395 |
| 100kB | 20 Bit | 524,507 | 524,519 | 4,045,499,352 | 800,000 | 21823479.45 | 21.8235 |
| 100kB | 32 Bit | 2,147,483,659 | 2,147,483,743 | 5,301,774,777,346,572 | 800,000 | 22353880.28 | 22.3539 |
| 100kB | 50 Bit | 869422333664431 | 151261980884887 | unkown | 800,000 | 25276475.03 | 25.2765 |

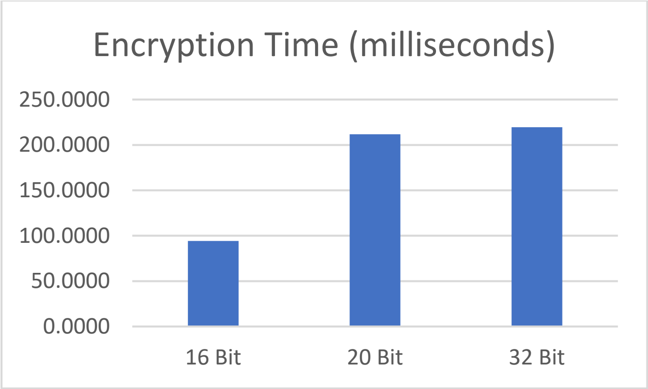


In this test a similar pattern can be observed. The size of the key definitely affects performance. Encrypting with a 32-bit Blum primes takes slightly longer than with a 20-bit Blum prime.

### 6.1.3 Encrypting a 1MB text file



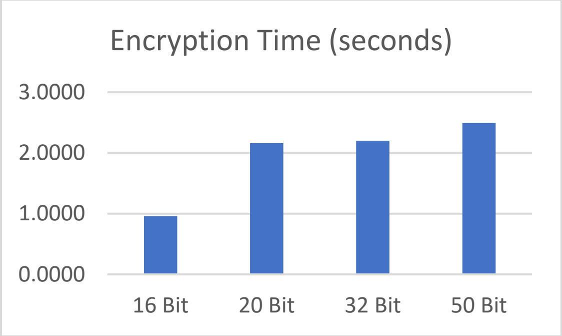
| File Size | p,q bit size | p | q | Maxium bits that can be generated by P*Q | Encrypted bits | Encryption Time (nano seconds) | Encryption Time (milliseconds) |
|---|---|---|---|---|---|---|---|
| 1MB | 16 Bit | 32,843 | 32,887 | 3,103,380 | 8,000,000 | 94286448.91 | 94.2864 |
| 1MB | 20 Bit | 524,507 | 524,519 | 4,045,499,352 | 8,000,000 | 211774265.34 | 211.7743 |
| 1MB | 32 Bit | 2,147,483,659 | 2,147,483,743 | 5,301,774,777,346,572 | 8,000,000 | 219587578.69 | 219.5876 |



It is clear at this point that there is a consistent pattern and a clear relationship between the size of the Blum primes used for encryption and the time it takes to encrypt a file. Although the difference in encryption time between a 20-bit Blum prime and a 32-bit Blum prime is not big, a small increase in encryption time with a 32-bit Blum prime has been observed across all the tests so far. It should be noted that set of 16-bit Blum primes used in this test *is not secure* because only up to 3 million bits can be generated with that set, but to encrypt a 1MB file 8 million bits are required. The numbers are included for consistency's sake.

### 6.1.4 Encrypting a 10MB text file

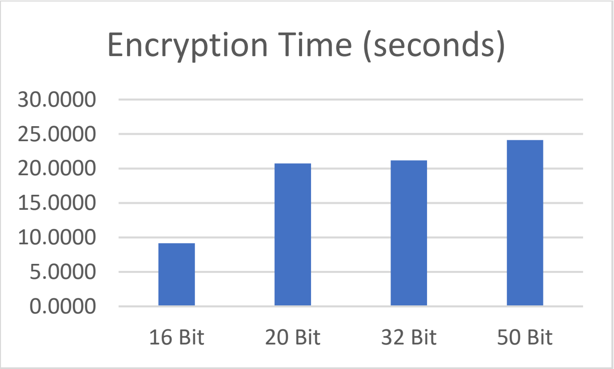
| File Size | p,q bit size | p | q | Maxium bits that can be generated by P*Q | Encrypted bits | Encryption Time (nano seconds) | Encryption Time (seconds) |
|---|---|---|---|---|---|---|---|
| 10MB | 16 Bit | 32,843 | 32,887 | 3,103,380 | 80,000,000 | 960323189.68 | 0.9603 |
| 10MB | 20 Bit | 524,507 | 524,519 | 4,045,499,352 | 80,000,000 | 2161058118.32 | 2.1611 |
| 10MB | 32 Bit | 2,147,483,659 | 2,147,483,743 | 5,301,774,777,346,572 | 80,000,000 | 2199444819.47 | 2.1994 |
| 10MB | 50 Bit | 869422333664431 | 151261980884887 | unkown | 80,000,000 | 2492809509.53 | 2.4928 |



At this point it is worth nothing that perhaps using smaller Blum primes expecting performance to increase may not necessarily be a good approach when picking Blum primes for encryption. Although we can see a clear increase in the time it takes to encrypt a file between 16-bit Blum primes and 20-bit Blum primes, choosing a 32-bit Blum prime over a 20-bit Blum would yield much better security for only a trivial decrease in performance.

### 6.1.5 Encrypting a 100MB text file

| File Size | p,q bit size | p | q | Maxium bits that can be generated by P*Q | Encrypted bits | Encryption Time (nano seconds) | Encryption Time (seconds) |
|---|---|---|---|---|---|---|---|
| 100MB | 16 Bit | 32,843 | 32,887 | 3,103,380 | 800,000,000 | 9164102779.10 | 9.1641 |
| 100MB | 20 Bit | 524,507 | 524,519 | 4,045,499,352 | 800,000,000 | 20724701590.29 | 20.7247 |
| 100MB | 32 Bit | 2,147,483,659 | 2,147,483,743 | 5,301,774,777,346,572 | 800,000,000 | 21172093099.58 | 21.1721 |
| 100MB | 50 Bit | 869422333664431 | 151261980884887 | unkown | 800,000,000 | 24132346206.86 | 24.1323 |



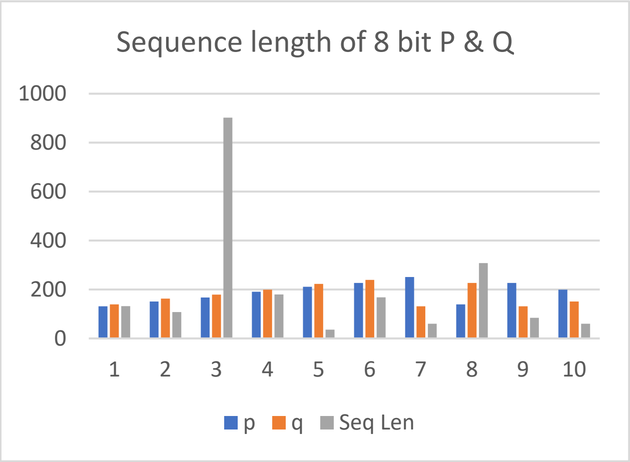
The pattern observed so far has been consistent. Although a bigger Blum prime does increase the encryption time, it does not do so in a linear fashion. Choosing big Blum primes provides better security at a very small cost in decreased performance. This performance hit is trivial when considering the use case of encrypting small amounts of data.

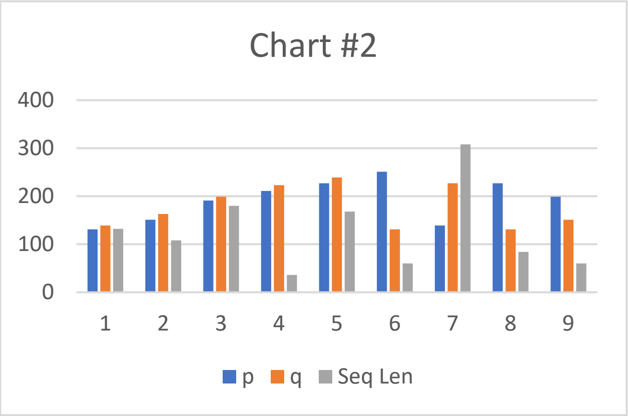
## 6.2 Blum Prime Length Test

This section analyzes the bit sequence length of Blum primes *p* and *q* of 8, 10, 16, 20 and 32 bits with seed number 4.

### 6.2.1 8-bit Blum primes sequence length

| size (bits) | p | q | n = p*q | n size(bits) | Seq Len |
|---|---|---|---|---|---|
| 8 | 131 | 139 | 18209 | 15 | 132 |
| 8 | 151 | 163 | 24613 | 15 | 108 |
| 8 | 167 | 179 | 29893 | 15 | 902 |
| 8 | 191 | 199 | 38009 | 16 | 180 |
| 8 | 211 | 223 | 47053 | 16 | 36 |
| 8 | 227 | 239 | 54253 | 16 | 168 |
| 8 | 251 | 131 | 32881 | 16 | 60 |
| 8 | 139 | 227 | 31553 | 15 | 308 |
| 8 | 227 | 131 | 29737 | 15 | 84 |
| 8 | 199 | 151 | 30049 | 15 | 60 |

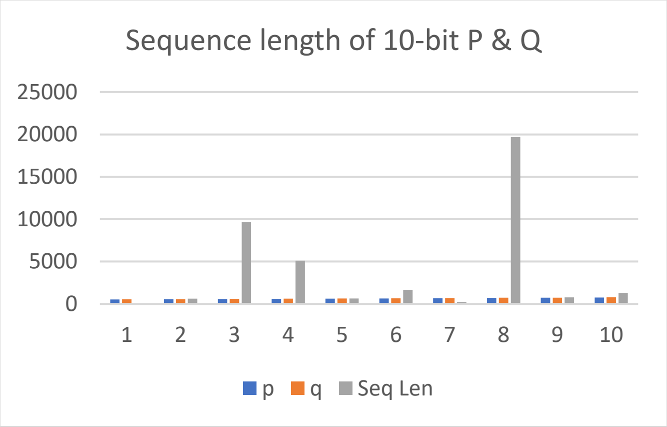


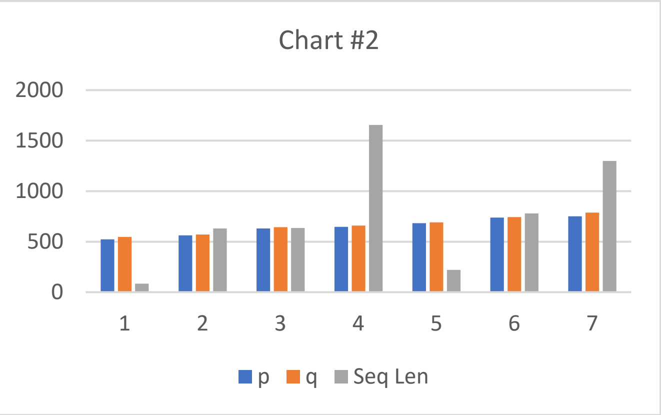


There appears to be no clear relationship between the sequence length and the bit size of 8-bit Blum primes. Chart #1 case (1) shows that the size of $p, q$ and the sequence is similar, but there are clear variations such as in case (2) (6) and (8). There are also outliers (3) and (5). Chart #2 has outlier (3) removed in order to get a closer look at the relationship between the bit size of $p, q$ and the sequence length.

### 6.2.2 10-bit Blum primes sequence length

| size (bits) | p | q | n = p*q | n size(bits) | Seq Len |
|---|---|---|---|---|---|
| 10 | 523 | 547 | 286081 | 19 | 84 |
| 10 | 563 | 571 | 321473 | 19 | 630 |
| 10 | 587 | 599 | 351613 | 19 | 9636 |
| 10 | 607 | 619 | 375733 | 19 | 5100 |
| 10 | 631 | 643 | 405733 | 19 | 636 |
| 10 | 647 | 659 | 426373 | 19 | 1656 |
| 10 | 683 | 691 | 471953 | 19 | 220 |
| 10 | 719 | 727 | 522713 | 19 | 19690 |
| 10 | 739 | 743 | 549077 | 20 | 780 |
| 10 | 751 | 787 | 591037 | 20 | 1300 |



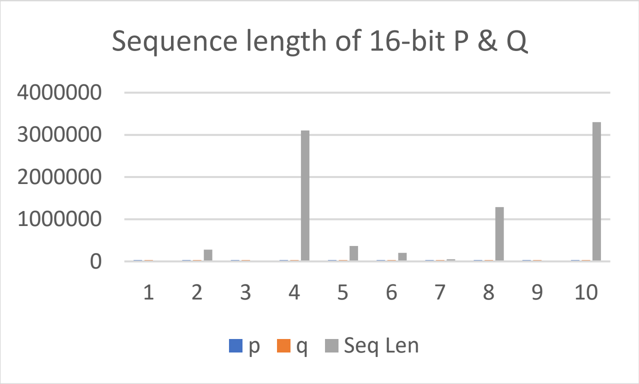


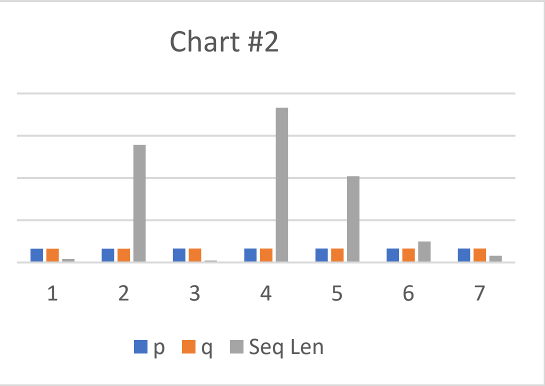
There, again, appears to be no clear relationship between the sequence length and the bit size of Blum primes. Chart #1 shows there are clear outliers. Chart #2 shows the relationship with outliers (3) (4) and (8) from chart #1 removed. At this point there appears to be a pattern emerging: while there is no clear relationship between the sequence length and the size of $p$ and $q$, there appears to be a set of numbers $p, q$ that can produce a distinctly large sequence.

### 6.2.3 16-bit Blum primes sequence length

| size (bits) | p | q | n = p*q | n size(bits) | Seq Len |
|---|---|---|---|---|---|
| 16 | 32771 | 32779 | 1074200609 | 31 | 8484 |
| 16 | 32783 | 32803 | 1075380749 | 31 | 278460 |
| 16 | 32831 | 32839 | 1078137209 | 31 | 4620 |
| 16 | 32843 | 32887 | 1080107741 | 31 | 3103380 |
| 16 | 32911 | 32939 | 1084055429 | 31 | 366338 |

| | | | | | |
|---|---|---|---|---|---|
| 16 | 32971 | 32983 | 1087482493 | 31 | 204204 |
| 16 | 32987 | 32999 | 1088538013 | 31 | 49476 |
| 16 | 33023 | 33071 | 1092103633 | 31 | 1289340 |
| 16 | 33083 | 33091 | 1094749553 | 31 | 16008 |
| 16 | 33107 | 33119 | 1096470733 | 31 | 3302124 |

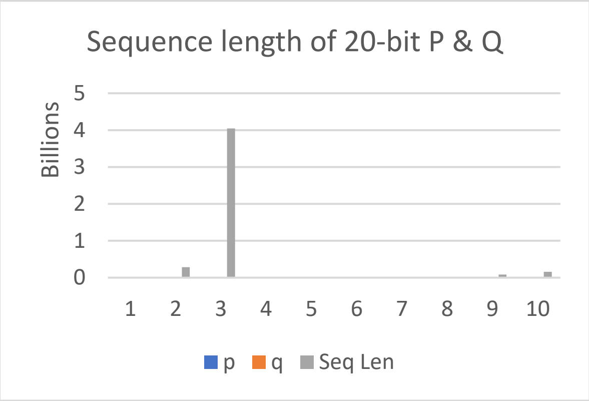


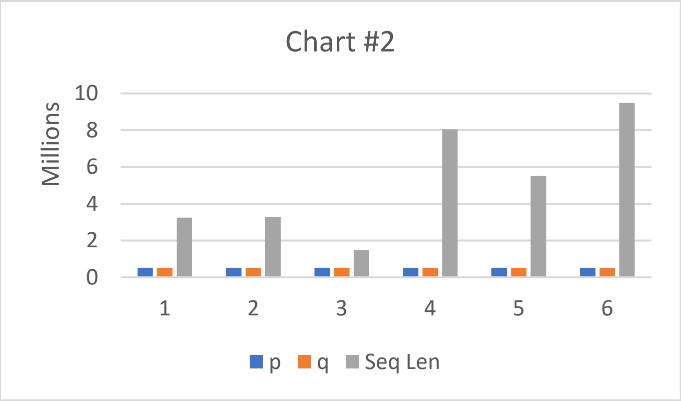


A pattern seems to be distinguishable at this point: there are certain Blum primes $p, q$ that produce a significantly larger sequence than other Blum primes. Chart #1 illustrates the massive difference between the size of $p, q$ and the produced sequence. In chart #2, the outliers (4) (8) (10) from chart #1 have been removed. It can be seen again that the size of $p, q$ is not a predictor of the size of the sequence.

### 6.2.4 20-bit Blum primes sequence length

| size (bits) | p | q | n = p*q | n size(bits) | Seq Len |
|---|---|---|---|---|---|
| 20 | 524347 | 524351 | 274941873797 | 39 | 3250660 |
| 20 | 524387 | 524411 | 274994311057 | 39 | 285199348 |
| 20 | 524507 | 524519 | 275113887133 | 39 | 4045499352 |
| 20 | 524591 | 524599 | 275199914009 | 39 | 3278700 |
| 20 | 524683 | 524707 | 275304842881 | 39 | 1486140 |
| 20 | 524731 | 524743 | 275348919133 | 39 | 8048898 |
| 20 | 524803 | 524827 | 275430784081 | 39 | 5518620 |
| 20 | 524831 | 524863 | 275464373153 | 39 | 9483660 |
| 20 | 524899 | 524939 | 275539956161 | 39 | 86614440 |
| 20 | 524947 | 524959 | 275575652173 | 39 | 158006940 |

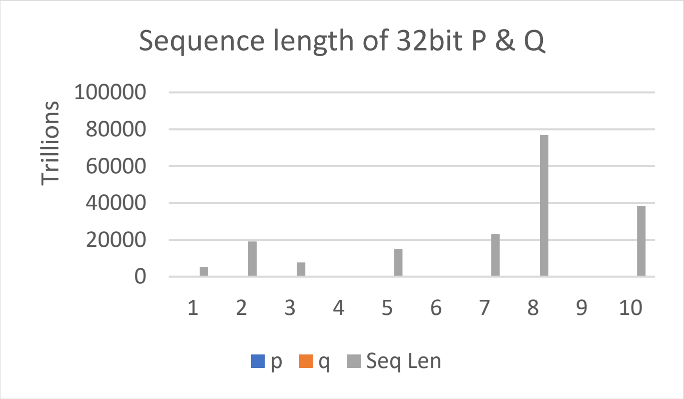


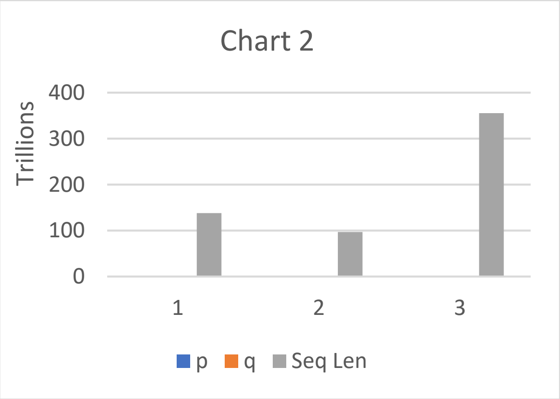


In this case, the pattern identified before continues; however, there is a particular outlier: the set of Blum primes, namely $p = 524507$ and $q = 524519$, that produced a 4 billion sequence. This is remarkable because that means there are Blum primes that can produce hundreds of megabytes worth of potential pad-values. For reference, 4 billion bits is equal to 500MB. Chart #2 has (2) (3) (9) and (10) from chart #1 removed to get a closer look at the smaller sequences generated.

### 6.2.5 32-bit Blum primes sequence length

| size (bits) | p | q | n = p*q | n size(bits) | Sequence Length |
|---|---|---|---|---|---|
| 32 | 2147483659 | 2147483743 | 4611686246060655637 | 63 | 5301774777346572 |
| 32 | 2147483659 | 2147483951 | 4611686692737256709 | 63 | 19086390656045580 |
| 32 | 2147483887 | 2147483951 | 4611687182363597537 | 63 | 7686145096349220 |
| 32 | 2147483999 | 2147484007 | 4611687543140903993 | 63 | 137836785684780 |
| 32 | 2147484083 | 2147483659 | 4611686976205099697 | 63 | 15068204159911308 |
| 32 | 2147484223 | 2147483951 | 4611687903918205073 | 63 | 96903493000380 |
| 32 | 2147484491 | 2147483951 | 4611688479443903941 | 63 | 23051525544851020 |
| 32 | 2147484259 | 2147483951 | 4611687981227627309 | 63 | 76861462559905260 |
| 32 | 2147484499 | 2147484331 | 4611689312667885169 | 63 | 355448053975020 |
| 32 | 2147484491 | 2147484499 | 4611689656265405009 | 63 | 38419219832497476 |





From looking at 32-bit Blum prime numbers, it can be concluded that there is no clear relationship between the size in bits of a Blum prime and the sequence produced by B.B.S PRNG. Furthermore, the smallest sequence in this test is 96 trillion 903 billion 493 million 380 numbers long. Even if only one bit is extracted per iteration, that means this set of Blum primes can generate a pad value of 12.11 TB (terabytes) of cryptographically secure pseudo random data with a single seed.

## 7. Conclusions and future work

The work performed has shown that using B.B.S PRNG to implement a practical communication channel secured by the perfect secrecy provided by OTP is feasible. Even exchanging a single key that is at least 64 bits long can provide plenty of pseudorandom data to secure communications for a long time, especially if the messages exchanged are short. Of course, some concerns remain, such as message authentication. Encryption speed of large files using the proposed OTP implementation could be increase by computing pad-values in parallel, but for this, knowledge of the sequence length produced by the input given to B.B.S PRNG is required. As pointed out in section 3.2, this knowledge can be obtained without computing the whole sequence by carefully choosing $N$, however, more research is needed to investigate if there exists an algorithm that can provide the sequence of the length for any combination of Blum primes and $p, q$ *without* having to compute the whole sequence. In section 4.5 the notion of using multiple seeds with a single $N$ was introduced as a way to increase the numbers of bits that could be generated with a single key. But as with any new work, there are several steps that need to be undertaken before it can be concluded that generating bits through that method is cryptographically secure. There are also numerous engineering and implementation design choices such as robustness to timing attacks, trivial brute force attacks, attacks on guessable and or weak "random" numbers, and so on that need to consider when implementing any cryptographic solution. In the future we hope to work on assessing some of these concerns as they apply to the proposed OTP implementation.

References

[1] J. Edney and W. Arbaugh, *Real 802.11 security*. Boston, MA: Addison-Wesley, 2004.

[2] T. Hardjono and L. Dondeti, *Security in wireless LANs and MANs*. Boston: Artech House, 2005.

[3] L. Blum, M. Blum and M. Shub, "A Simple Unpredictable Pseudo-Random Number Generator", *SIAM Journal on Computing*, vol. 15, no. 2, pp. 364-383, 1986.

[4] A. Langley, W. Chang, N. Mavrogiannopoulos, J. Strombergson and S. Josefsson, "ChaCha20-Poly1305 Cipher Suites for Transport Layer Security (TLS)", *Tools.ietf.org*, 2018.

[5] J. Katz and Y. Lindell, *Introduction to Modern Cryptography, Second Edition*. Hoboken: CRC Press, 2015, pp. 35-37.

[6] P. Junod, *Cryptographic Secure Pseudo-Random Bits Generation: The Blum-Blum-Shub Generator*. 1999.

[7] U. Vazirani and V. Vazirani, "Trapdoor pseudo-random number generators, with applications to protocol design", 24th Annual Symposium on Foundations of Computer Science (sfcs 1983), 1983.

[8] Vazirani U.V., Vazirani V.V. (1985) “Efficient and Secure Pseudo-Random Number Generation (Extended Abstract)”. In: Blakley G.R., Chaum D. (eds) Advances in Cryptology. CRYPTO 1984. Lecture Notes in Computer Science, vol 196. Springer, Berlin, Heidelberg

[9] D. Rijmenants, The complete guide to secure communications with the one time pad cipher, 7th ed. Cipher Machines & Cryptology, 2018.

[10] A. Rukhin, J. Soto, J. Nechvatal, M. Smid, and E. Barker, “A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications,” National Institute of Standards and Technology, Gaithersburg, MD, Special Pub.800-22Rev.1a,2010.

[11] P. D. Weadon, “Sigsaly Story,” NSA.gov, 03-May-2016. [Online]. Available: https://www.nsa.gov/about/cryptologic-heritage/historical-figures-publications/publications/wwii/sigsaly-story.shtml. [Accessed: 19-Jul-2018].

[12] “Hot Line Agreement,” U.S. Department of State. [Online]. Available: https://www.state.gov/t/isn/4785.htm. [Accessed: 20-Jul-2018].

[13] K. Becker, “ECE's Quantum Code Master,” Boston University College of Engineering, 02-Nov-2015. [Online]. Available: https://www.bu.edu/eng/2015/11/02/quantum-code-master/. [Accessed: 02-Aug-2018].

[14] J. M. David, “Technical design,” Jericho Comms™ - Technical design, 23-Apr-2017. [Online]. Available: https://joshua-m-david.github.io/jerichoencryption/information.html. [Accessed: 27-Aug-2018].

[15] Sidorenko A., Schoenmakers B. (2005) Concrete Security of the Blum-Blum-Shub Pseudorandom Generator. In: Smart N.P. (eds) Cryptography and Coding. Cryptography and Coding 2005. Lecture Notes in Computer Science, vol 3796. Springer, Berlin, Heidelberg

[16] Geisler, Martin; Krøigård, Mikkel; Danielsen, Andreas (December 2004). "About Random Bits".