Analysis of a Quantum Attack on the Blum-Micali Pseudorandom Number Generator

Tingfei (Carlos) Feng
Rose-Hulman Institute of Technology

June 2022

1 Introduction

In 2012, Guedes, Assis, and Lula [5] proposed a quantum attack on a pseudorandom number generator named the Blum-Micali Pseudorandom number generator. They claimed that the quantum attack can outperform classical attacks super-polynomially. However, this paper shows that the quantum attack cannot get the correct seed and provides another corrected algorithm that is in exponential time but still faster than the classical attack. Since the original classical attacks are in exponential time, the Blum-Micali pseudorandom number generator would be still quantum resistant.

In the context of growth in the information technology field, randomness becomes critical in applications like secure communication, simulation processes, and even recreational applications like video games. For example, a cryptographic nonce is a random unique value generated in protocols to show the messages' uniqueness, which can protect the message from replay attacks [3]. For video games, the wide use of procedural content generation takes advantage of pseudorandom number generators to create automatic generation of game content [9].

For digital computers, because of their deterministic nature, a random number generator that depicts realistic randomness is not feasible without additional hardware support. Therefore, pseudorandom number generators are designed to take an initial input, as seed, to produce a string of bits in which zero and one's appearances are equally possible.

One requirement for pseudorandom number generators is that they should be cryptographically secure. In other words, attackers cannot retrieve the initial seeds from generated random bit strings easily. This paper focuses mainly on the cryptographically secure pseudorandom number generator, the Blum-Micali pseudorandom number generator, which is based on the discrete logarithm problem [2].

The discrete logarithm problem is to find k as following:

$$\text{Given } a, b, \text{ and } n,\ \log_b a \mod n = k, \text{ find } k \text{ such that } b^k \mod n = a.$$

The problem is cryptographically hard based on the assumption that a digital computer cannot solve the k efficiently under carefully chosen a, b, and n. There is no known classical algorithm that can solve the discrete logarithm problem in polynomial time.

In 1994, Peter Shor discovered a quantum algorithm that can solve the discrete logarithm problem in polynomial time [8]. In quantum physics, different states can be expressed simultaneously in one superposition state. Therefore, a quantum computer can sometimes take advantage of the idea of examining multiple or even all states at the same. Since quantum computers can solve the discrete logarithm in polynomial time, people may wonder if quantum computers can break the Blum-Micali pseudorandom number generator that sets its foundation on the discrete logarithm.

Since the goal is to search for seeds from a large search space, the quantum search algorithm, or Grover's algorithm [4], provides the basic search scheme for finding the seed. Similar to Shor's algorithm, Grover's algorithm takes advantage of superposition states together with the ability to examine all the possibilities in the search space. Assuming that the seed is randomly chosen, the classical search algorithm performs in $\mathcal{O}(P)$ time. However, Grover's algorithm can search a random search space in $\mathcal{O}(\sqrt{P})$ time, which is polynomially faster than classical search.

Combining the ideas of two algorithms, we can adopt a general scheme from Grover's algorithm, which can speed the search for NP-complete problems [7]. The algorithm only should be polynomially faster than a classical attack which simply tries all possible seeds, because Grover's algorithm runs in $\mathcal{O}(\sqrt{P})$ compared to classical search's $\mathcal{O}(P)$. Furthermore, we conclude that the Blum-Micali Generator is quantum resistant to the quantum attacks described.

2 Blum-Micali Pseudorandom Number Generator

For a prime P, the integers from 1 to $P-1$ form a group under multiplication $\mod P$ denoted as Z_P^*. The Blum-Micali pseudorandom number generator [2] has two components:

1. Generator g for Z_P^* which recursively permutes the input under Z_P^*
2. Predicate $B_{P,g}(x)$ which is equal to 0 if x is the principal square root of $x^2 \mod P$, in other words, if x is less than $\frac{P-1}{2}$.

The number generator can be expressed by a large prime P, a generator $g \in \mathrm{Z}_P^*$, and a initial $x_0 \in \mathrm{Z}_P^*$ serving as the seed:

$$x_i = g^{x_{i-1}} \mod P$$
$$b_i = B(x_i).$$

Blum and Micali showed that since $x_i \in \mathrm{Z}_P^*$, instead of checking if x_i is the principal square root of $x^2 \mod P$, the predicate can check if $x_i \leq \frac{P-1}{2}$ [2]. Thus, the generator that produces a random bit string of length n can be described by the following pseudocode [2]:

Algorithm 1 Blum-Micali Pseudorandom Number Generator

Input: g, $seed$, n, P
Output: b

```
x_i ← seed
b = [ ]
for iteration = 1, 2, ..., n do
    if x_i > (P−1)/2 then
        Append 1 to b
    else
        Append 0 to b
    end if
    x_i = g^{x_i} mod P
end for
```

The list b is a random bit string, where the i^{th} random bit is the result of the predicate in the i^{th} iteration. Blum and Micali proved that the bits are equally likely to be zeros and ones [2], and the generator passes Yao's statistical test for a secure pseudorandom number generator [10]. This level of security needs a careful choice of the generator. The generator g should have a long enough non-repeating period of $g^x \mod P$ in order to be able to produce long random bit strings. We can use random strings that have length of $\mathcal{O}(\log P)$ to find their seeds, which is shown in 8.1.

In Figure 1, we have the generator $g = 5$ and the prime $P = 23$. If the initial seed is 3, we can iterate through as $5^3 \mod 23 = 10$, $5^{10} \mod 23 = 9$. Since 3, 10, and 9 are all smaller than 12, the result bit string starts with 000.

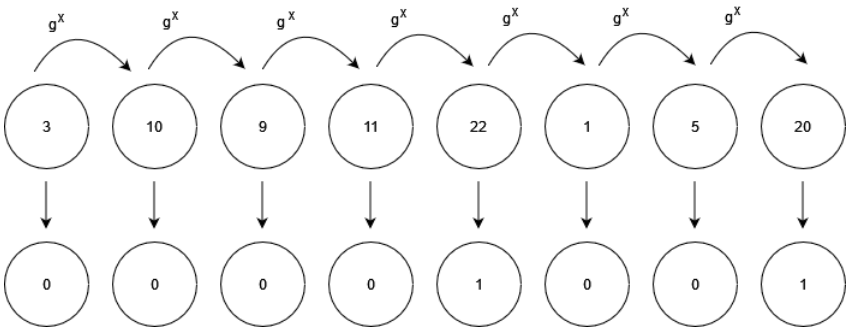

Figure 1: Example of the generator with $P = 23,\ g = 5$

3 Classical Attack

3.1 Algorithm of the Classical Attack

In this section, we propose a simple classical attack to the Blum-Micali pseudorandom number generator. The algorithm tries every possible seed from 1 to $P - 1$. Bits in the random bit string b indicate the if the corresponding x_i is a principal square root, or, in other words, less than $\frac{P-1}{2}$. Thus, we can have an inverse predicate B':

$$B'(x_i, b_i) = \begin{cases} 1 & \text{if } b_i = 0,\ x_i \leq \frac{P-1}{2} \\ 0 & \text{if } b_i = 0,\ x_i > \frac{P-1}{2} \\ 0 & \text{if } b_i = 1,\ x_i \leq \frac{P-1}{2} \\ 1 & \text{if } b_i = 1,\ x_i > \frac{P-1}{2} \end{cases}$$

1 stands for x_i produces the corresponding b_i; 0 stands for x_i fails to produce the corresponding b_i, and, therefore, the initial guess x is false. For each trial, we test if x_i satisfies the random bit string b by iterating x under $x = g^x \mod P$ and testing it with the inverse predicate. The complete classical break of Blum-Micali Pseudorandom number generator is described by Algorithm 2.

3.2 Complexity Analysis of the Classical Attack

Given that the prime is large enough, we can express it in binary form in n bits, where $n = \lceil \log P \rceil + 1$. For exponentiation of $g^x \mod P$, we can implement that by squaring operations and get $\mathcal{O}(\log^3 P)$ for each exponentiation calculations inside the innermost loop line 14. For other calculations inside the innermost loop, comparing x_i would take at most $\mathcal{O}(n)$ by examining all the bits [5]. Substituting P for 2^n, we get $\mathcal{O}(\log P)$. For b_i verification, since b_i is a one-bit value, it takes $\mathcal{O}(1)$. Thus, in total, in the most inner loop, the time complexity is $\mathcal{O}(\log^3 P)$.

Since we loop the verification process by a number of times equal to the length of the random bit string, determining how many bits in the random bit strings we need to break Blum-Micali pseudorandom generator becomes critical. As the predicate B separates the values of x_i into two sets with equal size, the probability of the inverse predicate B' to eliminate a possible seed is 0.5. The number of all possible seeds from 2 to $P - 1$ is $P - 2$, which is $\mathcal{O}(P)$. Thus, we need a random bit string b of size $\mathcal{O}(\log P)$ to eliminate all possible seeds [2]. Thus, to see if the guessed seed is correct, we need to check at worst $\mathcal{O}(\log P)$ times, which gives us a time complexity of $\mathcal{O}(\log^4 P)$. In Appendix A we provide some examples of how many bits we need to break the Blum-Micali Pseudorandom Number Generator.

Since we may guess $\mathcal{O}(P)$ seeds, the total time complexity will be $\mathcal{O}(P \log^4 P)$. After converting P into n bits, this gives us a time complexity of $\mathcal{O}(n^4 2^n)$, which lies in the exponential complexity class. Therefore, if we choose a large P with $n = 64$ or 128, it is impractical to break the Blum-Micali Pseudorandom number generator. For space complexity, since we only store variables like x, x_i, and i bounded by P and j bounded by $\mathcal{O}(\log P)$, the overall space complexity in bits is $\mathcal{O}(n)$. However, in Guedes, Assis, and Lula's paper

Algorithm 2 Classical Attack on the Blum-Micali Pseudorandom Number Generator

Input: g, P, b
Output: x

for $i = 1, 2, \ldots, P-1$ **do**
  $x \leftarrow i$
  $x_i \leftarrow x$
  for $j = 1, 2, \ldots, \text{length}(b)$ **do**
    if $b[j] = 0$ **then**
      if $x_i > \frac{P-1}{2}$ **then**
        goto loop
      end if
    else
      if $x_i \leq \frac{P-1}{2}$ **then**
        goto loop
      end if
    end if
    $x_i = g^{x_i} \mod P$
  end for
  return x
  loop:
end for

[5], they claimed that their classical attack would work in $\mathcal{O}(2^P \log P)$. In terms of number of bits n, the complexity is $\mathcal{O}(2^{2^n} n)$. Their algorithm needs to solve the discrete logarithm problems $\mathcal{O}(\log P)$ times, since they iterate all the possible guesses at the same time and do not save the original guesses before iterating them. Besides that, they claimed that attacking the discrete logarithm problem was $\mathcal{O}(2^P)$. One of the most efficient algorithm for the discrete logarithm problem on classical computers is function field sieve method, which works in $\mathcal{O}(\exp((\sqrt[3]{\frac{32}{9}} + o(1))(\log P)^{\frac{1}{3}}(\log \log P)^{\frac{2}{3}}))$ [1].

4 Quantum Attack

Based on the assumption that we cannot predict whether the new $x = g^x \mod P$ is a principal suare root, a quantum computer, similar to a classical one, needs to examine all the possibilities of the unknown seed. The advantage lies in quantum superposition of states, which enables the algorithm to check all the possibilities at the same time. Guedes, Assis, and Lula's quantum algorithm has three main parts [5]:

1. State Identification
2. State Amplitude Amplification
3. State Recovery

Part 1 will identify the correct seed by a similar procedure as how the generator produces the random bit string. Part 2 can be viewed as a generalized version of Grover's algorithm to find the seed, which aims to amplify the amplitude of the correct seed's state. Part 3 will then use Shor's algorithm to reverse the identification process of Part 1 to find the original seed. We can predict the next bit of the random bit string without Part 3 if we do not want the original seed.

4.1 State Identification

Suppose we have the prime P, the generator g, and enough bits of the random bit string b in length j of $\mathcal{O}(\log P)$. Starting with two registers, we have

- Result register with $n = \lceil \log P \rceil$ qubits

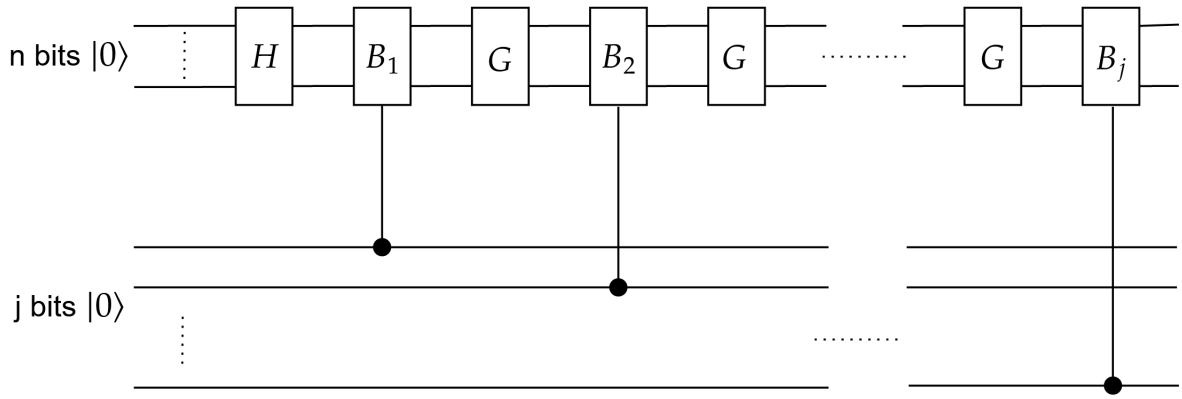

Figure 2: Quantum Circuit for State Identification Part

- Guess register with j qubits

We initialize each qubit of two quantum registers to be $|0\rangle$. Then the initial state of two registers combined is

$$|\psi_{initial}\rangle = |0\rangle^{\otimes n} \otimes |0\rangle^{\otimes j} \tag{1}$$

The quantum circuit for the State Identification Part is shown in Figure 2. We first apply a Hadamard Gate to create a uniformly distributed superposition of all our computational basis, which are $|0\rangle, |1\rangle, |2\rangle ... |2^n - 1\rangle$. The result register is:

$$|\phi\rangle = \frac{1}{\sqrt{2^n}} \sum_{k=0}^{2^n-1} |k\rangle \tag{2}$$

Since no operation is made on the guess register, the overall state of the two registers is

$$|\psi_1\rangle = \frac{1}{\sqrt{2^n}} \sum_{k=0}^{2^n-1} |k\rangle |0\rangle^{\otimes j} \tag{3}$$

We define gate G and gate B_i as following:

$$B_i |x\rangle |y\rangle_i = \begin{cases} |x\rangle \overline{|y\rangle_i} & \text{if } b_i = 0,\ x_i \leq \frac{P-1}{2} \\ |x\rangle |y\rangle_i & \text{if } b_i = 0,\ x_i > \frac{P-1}{2} \\ |x\rangle |y\rangle_i & \text{if } b_i = 1,\ x_i \leq \frac{P-1}{2} \\ |x\rangle \overline{|y\rangle_i} & \text{if } b_i = 1,\ x_i > \frac{P-1}{2} \end{cases}$$

$$G |x\rangle = |g^x \mod P\rangle$$

The B_i gates are gates that examine whether $|x\rangle$ can lead to the correct bit string and change i^{th} bit of guess register $|y\rangle$ to 1 if satisfied. G gates' purpose is to iterate j times to provide the corresponding x values for B_i gates to examine. Since we have j bits of random bit string b and assume that we need j bits to locate the correct seed, the numbers for G gates and B_i gates are both j.

After j iterations of G and B_i gates, since there is no amplitude change during identification process, the overall state is:

$$|\psi\rangle = \frac{1}{\sqrt{2^n}} \sum_{k=0}^{2^n-1} |k\rangle |y_k\rangle, \tag{4}$$

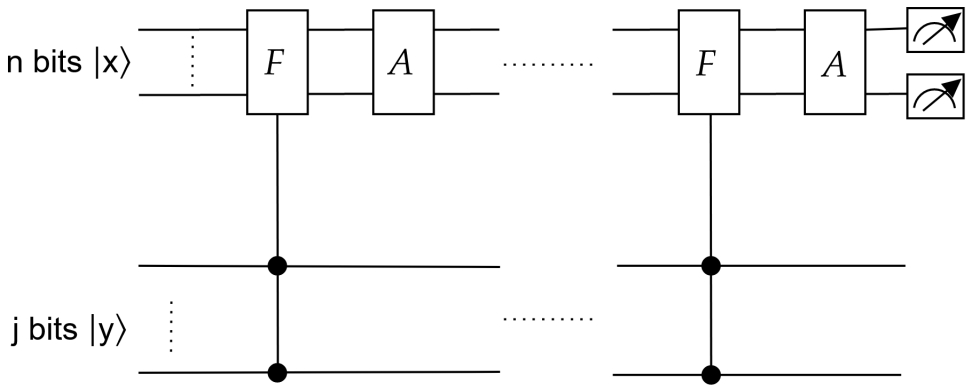

Figure 3: Quantum Circuit for State Amplitude Amplification Part

where $|y_k\rangle$ is the guess register value.

If we isolate the correct seed $|x_j\rangle$ out of other states, we will have:

$$|\psi\rangle = \frac{1}{\sqrt{2^n}} \sum_{k=0,\neq x_j}^{2^n-1} |k\rangle |y_k\rangle + \frac{1}{\sqrt{2^n}} |x_j\rangle \left|y_{x_j}\right\rangle, \tag{5}$$

where $|y_k\rangle$ are the guess register values that are not all ones, and $\left|y_{x_j}\right\rangle$ is the guess register value that is all ones.

4.2 State Amplitude Amplification

From Equation (5), we can measure the result register and guess register. The probability of getting the correct state is equal to the probability of the other wrong answers, which is $|\frac{1}{\sqrt{2^n}}|^2 = \frac{1}{2^n}$. When the big prime P is large, or, in other words, n is large enough, it is impractical for us to measure the correct state. The next step is to solve this problem by amplifying the correct state $|x_j\rangle$ to have an amplitude close to 1.

Shown in Figure 3, we have the amplification quantum circuit with its input as Equation (5). F gates and A gates are described as:

$$F |x\rangle |y\rangle = \begin{cases} |x\rangle |y\rangle & \text{if } |y\rangle \neq \left|y_{x_j}\right\rangle \\ -|x\rangle |y\rangle & \text{if } |y\rangle = \left|y_{x_j}\right\rangle \end{cases}$$

$$A = 2 |\phi\rangle \langle\phi| - \mathbb{I}$$

$$A |x\rangle |y\rangle = (2 |\phi\rangle \langle\phi| - \mathbb{I}^n) |x\rangle \otimes \mathbb{I}^j |y\rangle$$

We notice that for any $|x\rangle$ in Z_P^*, $A |x\rangle |y\rangle$ will be:

$$\begin{aligned} A |x\rangle |y\rangle &= (2 |\phi\rangle \frac{1}{\sqrt{2^n}} - |x\rangle) \otimes |y\rangle \\ &= \frac{2}{\sqrt{2^n}} |\phi\rangle |y\rangle - |x\rangle |y\rangle \end{aligned}$$

This indicates that after applying the A gate, an arbitrary state $|x\rangle$ tensored with its own guess register value $|y\rangle$ will become $|\phi\rangle$ tensored with $|y\rangle$, where $|\phi\rangle$ is the initial state. In other words, the result is the superposition of the state tensored with its guess register value and all other states tensored with the arbitrary state $|x\rangle$'s guess register value $|y\rangle$. These tensor products are not part of the starting state and final state. Therefore, the original starting state and final state in Guedes, Assis, and Lula's paper do not fully represent the intermediate and final states. We define five linearly independent terms to explain the process:

$$\left|\psi_{x_j\perp}\right\rangle = \sum_{k=0,\neq x_j}^{2^n-1} \frac{1}{\sqrt{2^n-1}} |k\rangle |y_k\rangle \tag{6}$$

$$\left|\psi_{x_j}\right\rangle = |x_j\rangle \left|y_{x_j}\right\rangle \tag{7}$$

$$\left|\psi'_{x_j\perp}\right\rangle = \sum_{k=0,\neq x_j}^{2^n-1} \frac{1}{\sqrt{2^n-1}} |k\rangle \left|y_{x_j}\right\rangle \tag{8}$$

$$\left|\psi'_{x_j}\right\rangle = \sum_{k=0,\neq x_j}^{2^n-1} \frac{1}{\sqrt{2^n-1}} |x_j\rangle |y_k\rangle \tag{9}$$

$$|\sigma\rangle = \frac{1}{\sqrt{2^n-1}\sqrt{2^n-2}} \sum_{k=0,\neq x_j}^{2^n-1} \sum_{l=0,\neq x_j,k}^{2^n-1} |l\rangle |y_k\rangle \tag{10}$$

$\left|\psi_{x_j\perp}\right\rangle$ is the superpostion state of wrong guesses tensored with their own result register values. $\left|\psi_{x_j}\right\rangle$ is the state of the correct guess $|x_j\rangle$ tensored with $\left|y_{x_j}\right\rangle$. $\left|\psi'_{x_j\perp}\right\rangle$ is the superposition state of wrong guesses tensored with correct result register values $\left|y_{x_j}\right\rangle$. $\left|\psi'_{x_j}\right\rangle$ is the superposition state of the correct guess $|x_j\rangle$ tensored with all wrong guess register values. $|\sigma\rangle$ is the superposition state of wrong guesses tensored with all other wrong guesses' result register values.

We define a 5×5 transformation matrix Q that is the combination of F and A gates on $|x\rangle$ and $|y\rangle$.

$$\begin{aligned} Q &= A \cdot F \\ &= (2 |\phi\rangle \langle\phi| \otimes \mathbb{I}^j - \mathbb{I}^n \otimes \mathbb{I}^j) \cdot F \end{aligned}$$

$$Q \begin{pmatrix} \left|\psi_{x_j\perp}\right\rangle \\ \left|\psi_{x_j}\right\rangle \\ \left|\psi'_{x_j\perp}\right\rangle \\ \left|\psi'_{x_j}\right\rangle \\ |\sigma\rangle \end{pmatrix} = \begin{pmatrix} \frac{2}{2^n}-1 & 0 & 0 & \frac{2}{2^n} & \frac{2\sqrt{2^n-2}}{2^n} \\ 0 & \frac{2}{2^n}-1 & \frac{2\sqrt{2^n-1}}{2^n} & 0 & 0 \\ 0 & \frac{2\sqrt{2^n-1}}{2^n} & \frac{2(2^n-2)}{2^n}-1 & 0 & 0 \\ \frac{2}{2^n} & 0 & 0 & \frac{2}{2^n}-1 & \frac{2\sqrt{2^n-2}}{2^n} \\ \frac{2\sqrt{2^n-2}}{2^n} & 0 & 0 & \frac{2\sqrt{2^n-2}}{2^n} & \frac{2(2^n-2)}{2^n}-1 \end{pmatrix} \begin{pmatrix} \left|\psi_{x_j\perp}\right\rangle \\ \left|\psi_{x_j}\right\rangle \\ \left|\psi'_{x_j\perp}\right\rangle \\ \left|\psi'_{x_j}\right\rangle \\ |\sigma\rangle \end{pmatrix} \tag{11}$$

We can use the matrix Q^k and a vector v which represents the initial state $|\psi\rangle$ to get the amplitude after k iterations, where the sum of each element in v represents the state. Furthermore, the transformation matrix Q^2 is an identity matrix.

$$v = \begin{pmatrix} \sqrt{1-\frac{1}{2^n}} \left|\psi_{x_j\perp}\right\rangle \\ \frac{1}{\sqrt{2^n}} \left|\psi_{x_j}\right\rangle \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

$$Q^k \cdot v = \begin{cases} \begin{pmatrix} \sqrt{1-\frac{1}{2^n}}\left|\psi_{x_j\perp}\right\rangle \\ \frac{1}{\sqrt{2^n}}\left|\psi_{x_j}\right\rangle \\ 0 \\ 0 \\ 0 \end{pmatrix} & \text{if } k \text{ is even} \\ \frac{1}{2^n\sqrt{2^n}} \begin{pmatrix} (2-2^n)\sqrt{2^n-1}\left|\psi_{x_j\perp}\right\rangle \\ 2^n-2\left|\psi_{x_j}\right\rangle \\ -2\sqrt{2^n-1}\left|\psi'_{x_j\perp}\right\rangle \\ 2\sqrt{2^n-1}\left|\psi'_{x_j}\right\rangle \\ 2\sqrt{2^n-1}\sqrt{2^n-2}\left|\sigma\right\rangle \end{pmatrix} & \text{if } k \text{ is odd} \end{cases} \tag{12}$$

The probability of measuring the correct seed x_j is:

$$\begin{aligned} P &= \begin{cases} (\frac{1}{\sqrt{2^n}})^2 & \text{if } k \text{ is odd} \\ (\frac{2^n-2}{2^n\sqrt{2^n}})^2 + (\frac{2\sqrt{2^n-1}}{2^n\sqrt{2^n}})^2 & \text{if } k \text{ is even} \end{cases} \\ &= \begin{cases} \frac{1}{2^n} & \text{if } k \text{ is odd} \\ \frac{1}{2^n} & \text{if } k \text{ is even} \end{cases} \end{aligned}$$

We can see that the algorithm is not amplifying the amplitude for the state $|x_j\rangle$. We can confirm the result by applying gates A and F directly on the initial state. The results of first and second iteration confirm what we found by the transformation matrix Q^k.

$$AF\left|\psi\right\rangle = \frac{2-2^n}{2^n\sqrt{2^n}}\sqrt{2^n-1}\left|\psi_{x_j\perp}\right\rangle + \frac{2^n-2}{2^n\sqrt{2^n}}\left|\psi_{x_j}\right\rangle + \frac{-2\sqrt{2^n-1}}{2^n\sqrt{2^n}}\left|\psi'_{x_j\perp}\right\rangle + \frac{2\sqrt{2^n-1}}{2^n\sqrt{2^n}}\left|\psi'_{x_j}\right\rangle + \frac{2\sqrt{2^n-1}\sqrt{2^n-2}}{2^n\sqrt{2^n}}\left|\sigma\right\rangle$$

$$(AF)^2\left|\psi\right\rangle = \frac{1}{\sqrt{2^n}}\sum_{k=0}^{2^n-1}\left|k\right\rangle\left|y_k\right\rangle = \left|\psi\right\rangle$$

Therefore, the algorithm will not amplify the amplitude with an initial state that has different amplitudes.

4.3 State Recovery

If we correctly amplify the correct seed's state, we can measure the register to get the seed after j applications of $g^x \mod P$. We can now use this seed to predict future random bit strings. However, if we want to get the original seed, we need to apply j times Shor's period finding algorithm [**?**] to solve the discrete logarithm problem. In this process, we follow the steps given by Jozsa [6].

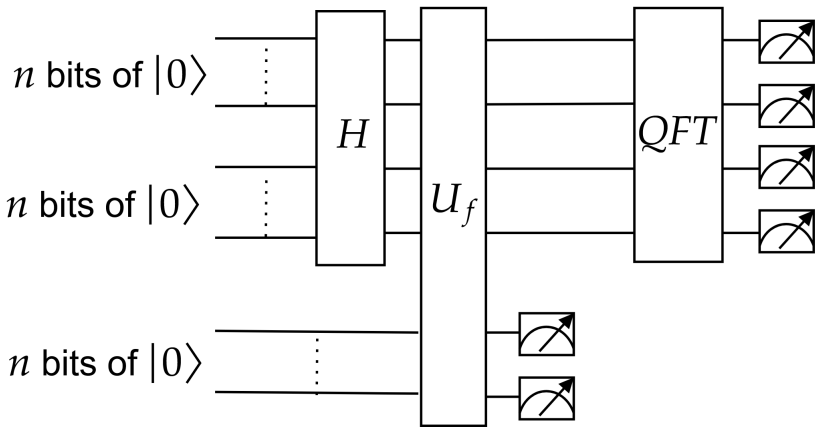

Figure 4: Quantum Circuit for Shor's algorithm

In Figure 4, we show the quantum circuit for Shor's algorithm. The three quantum registers are each initialized to n bits of $|0\rangle$. The initial state is:

$$|\theta_0\rangle = |0\rangle^{\otimes n} |0\rangle^{\otimes n} |0\rangle^{\otimes n} \tag{13}$$

We apply a $2n$-bit Hadamard gate to the first two registers:

$$\begin{aligned} |\theta_1\rangle &= H^{\otimes 2n} \otimes I^{\otimes n} |\theta_0\rangle \\ &= \frac{1}{\sqrt{2^n}} \frac{1}{\sqrt{2^n}} \sum_{a=0}^{2^n-1} |a\rangle \sum_{b=0}^{2^n-1} |b\rangle |0\rangle^{\otimes n} \\ &= \frac{1}{2^n} \sum_{a=0}^{2^n-1} \sum_{b=0}^{2^n-1} |a\rangle |b\rangle |0\rangle^{\otimes n}, \end{aligned} \tag{14}$$

where $|a\rangle$, $|b\rangle$, $|c\rangle$ are the three registers' values.

Because the operation $g^x \mod P$ is in the range of 0 to $P-1$, we should consider the only numbers in Z_P^*. Therefore, the state $|\theta_1\rangle$ should be:

$$|\theta_1\rangle = \frac{1}{P-1} \sum_{a=0}^{P-2} \sum_{b=0}^{P-2} |a\rangle |b\rangle |0\rangle^{\otimes n}. \tag{15}$$

We define a function f given a generator g, a big prime P, and a number x_j in Z_P^*, where $x_j = g^{x_{j-1}} \mod P$:

$$\begin{aligned} f(a,b) &= g^a x_j^{-b} \mod P \\ &= g^a g^{x_{j-1} \cdot -b} \mod P \\ &= g^{a - x_{j-1} \cdot b} \mod P \end{aligned} \tag{16}$$

For two pairs of (a,b), we have $f(a_1,b_1) = f(a_2,b_2)$ if and only if $(a_2,b_2) = (a_1,b_1) + \lambda \cdot (x_{j-1}, 1)$ for some λ in Z_P^*. We can substitute (a_2,b_2) in f:

$$\begin{aligned} f(a_2,b_2) &= g^{a_1 + \lambda \cdot x_{j-1}} \cdot x_j^{-b_1 - \lambda} \mod P \\ &= g^{a_1} \cdot g^{\lambda \cdot x_{j-1}} \cdot x_j^{-b_1} \cdot x_j^{-\lambda} \mod P \\ &= g^{a_1} \cdot x_j^{\lambda} \cdot x_j^{-b_1} \cdot x_j^{-\lambda} \mod P \\ &= g^{a_1} x_j^{-b_1} \mod P \\ &= f(a_1,b_1). \end{aligned}$$

We have the gate U_f implementing the function f, which takes the state $|a\rangle |b\rangle |0\rangle^{\otimes n}$. The gate will compute $f(a,b)$ and do an exclusive or of the result to the last n bits. After applying U_f to the state, we get:

$$|\theta_2\rangle = \frac{1}{P-1} \sum_{a=0}^{P-2} \sum_{b=0}^{P-2} |a\rangle |b\rangle |f(a,b)\rangle .$$

In Figure 4, we measure the third register as $|f(a_o, b_0)\rangle$, which leads to collapse of the superposition. The new state in the first two registers is:

$$|\theta_3\rangle = \frac{1}{\sqrt{P-1}} \sum_{k=0}^{P-2} |a_0 + k \cdot x_{j-1}\rangle |b_0 + k\rangle . \tag{17}$$

Since a_0 and b_0 are chosen randomly, when we measure the quantum registers, we cannot calculate x_{j-1} from $a_0 + k \cdot x_{j-1}$ and $b_0 + k$ composed by random values a_0, b_0, and k. To eliminate the dependency from a_0 and b_0, we first define a function below:

$$\chi_{l_1,l_2}(a,b) = \exp(2\pi i \frac{al_1 + bl_2}{P-1})$$

The gate QFT, which stands for Quantum Fourier Transform, will yield an equally weighted superposition of l_1 and l_2 where $\chi_{l_1,l_2}(x_{j-1}, 1) = 1$. In other words, $x_{j-1} \cdot l_1 + l_2 = 0 \mod P-1$, so $l_2 = -x_{j-1} \cdot l_1$ and $l_1 = 0, 1, ..., P-2$. The state will be:

$$|\theta_4\rangle = \frac{1}{\sqrt{P-1}} \sum_{k=0}^{P-2} \exp(2\pi i \frac{a_0 l_1 - b_0 x_{j-1} l_1}{P-1}) |l_1\rangle |-x_{j-1} l_1\rangle . \tag{18}$$

We then measure the two registers to get $(l_1, -x_{j-1}l_1 \mod P-1)$. From the state, we can see that l_1 is uniformly distributed from 0 to $P-1$. If l_1 is coprime to $P-1$, we can find the multiplicative inverse l_1^{-1} modulo $P-1$ and multiply $-x_{j-1}l_1 \mod P-1$ by $-l_1^{-1}$ to get x_{j-1}. If l_1 is not coprime to $P-1$, we need to repeat the state recovery process with the same inputs. The probability of a random number l_1 being coprime to $P-1$ is $\mathcal{O}(\frac{1}{\log \log P})$ [6], so the average-case number of repetitions for Shor's algorithm is $\mathcal{O}(\log \log P)$. Since we need to have $\mathcal{O}(\log P)$ random bits to break the generator, the overall time complexity for this step is $\mathcal{O}(n^3 \log P \log \log P)$, since the U_f gate is using exponentiation which has complexity of $\mathcal{O}(n^3)$, and the QFT gate has $\mathcal{O}(n^2)$ gates from Section 5.1 in the book *Quantum Computation and Quantum Information* [7]. In terms of the big prime P's number of bits n, it is $\mathcal{O}(n^4 \log n)$.

5 Corrected Quantum Attack

5.1 Algorithm

From Section 6.4 in the book *Quantum Computation and Quantum Information* [7], we get the idea of using Grover's Search Algorithm for NP-complete problems and an oracle implementation to correct the original algorithm. The oracle is a quantum circuit that can tell whether an input is a solution to the problem in polynomial time. To attack the Blum-Micali Pseudorandom number generator, we can easily verify whether a given guess is a solution in polynomial time, which we showed in 3.2. Therefore, the problem is in the NP complexity class. We can adopt the general scheme for solving NP-complete problems to this problem.

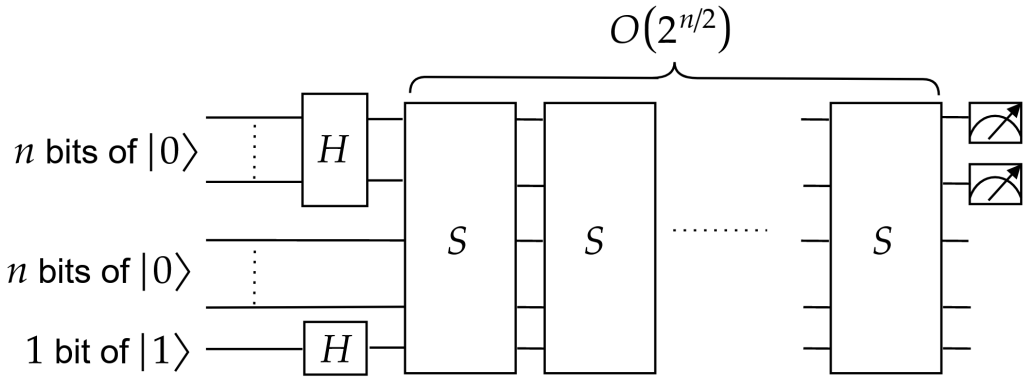

Figure 5: Circuit for Grover's Search Algorithm

From Figure 5, we need n bits of guess register and $n+1$ bits of working register W for the intermediate calculations [7]. We first apply the Hadamard gate to the n-bit guess register, which is initialized to $|0\rangle$. The state will be

$$|\omega\rangle = \frac{1}{\sqrt{2^n}} \sum_{k=0}^{2^n-1} |k\rangle \,. \tag{19}$$

We will apply a Hadamard gate to the last bit of the working bits, which is initialized to $|1\rangle$.

$$\begin{aligned} |W_{n+1}\rangle &= H\,|1\rangle \\ &= \frac{|0\rangle - |1\rangle}{\sqrt{2}} \end{aligned} \tag{20}$$

We need $\mathcal{O}(\sqrt{2^n})$ Grover iterators S, which are circuits based on [7] that amplify the amplitude of the correct seed to 1 so that we can measure the correct solution at the end with a high probability. To be more precise, we have an upper bound on the number of Grover iterators:

$$\text{Number of iterators} = \left\lceil \frac{\pi}{4}\sqrt{2^n} \right\rceil .$$

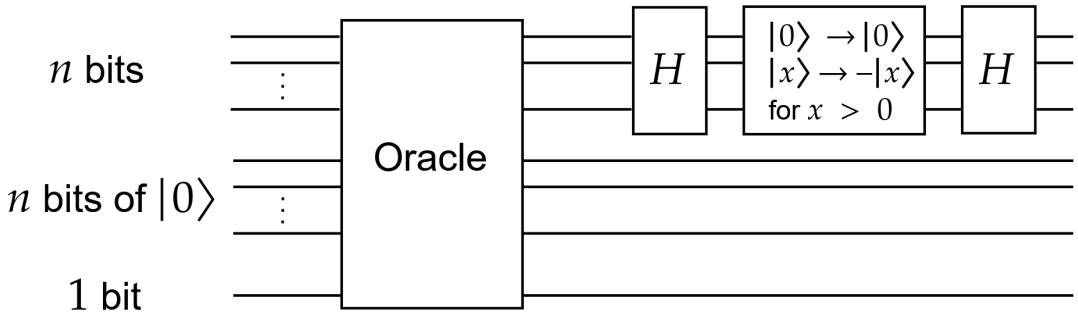

Figure 6: Circuit for Grover Iterator

Figure 6 shows the quantum circuit for one Grover iterator S. The oracle, which is a quantum circuit that can tell whether the input is a correct guess, O, is:

$$O\,|x\rangle\,|W\rangle = (-1)^{f(x)}\,|x\rangle\,|W\rangle\,,$$

where $f(x)$ is:

$$f(x) = \begin{cases} 1 & x \text{ is the seed} \\ 0 & x \text{ is not the seed.} \end{cases}$$

A general state $\sum_k \alpha_k |k\rangle$ after the oracle O becomes:

$$\sum_{k \neq seed} \alpha_k |k\rangle - \sum_{k=seed} \alpha_k |k\rangle ,$$

where there could be multiple solutions, but here we assume that if we have a random bit string with length of $\mathcal{O}(n)$ as in the classical attack, there is only one solution. The result will be:

$$|\omega_1\rangle = \sum_{k \neq x_j} \alpha_k |k\rangle - \alpha_{x_j} |x_j\rangle .$$

Then we apply the next three gates which have a combined effect of $(2 |\phi\rangle \langle\phi| - \mathbb{I})$. We define $\cos\left(\frac{\tau}{2}\right) = \sqrt{\frac{2^n-1}{2^n}}$. As $\alpha_k = \frac{1}{\sqrt{2^n-1}}$, $\alpha_{x_j} = -1$, we can express $|\omega_1\rangle$ as $\cos\left(\frac{\tau}{2}\right) \sum_{k \neq x_j} \alpha_k |k\rangle + \sin\left(\frac{\tau}{2}\right) \alpha_{x_j} |x_j\rangle$. The result of these gates on $|\omega_1\rangle$ is:

$$S |\omega_1\rangle |W\rangle = \cos\left(\frac{3\tau}{2}\right) \sum_{k \neq x_j} \alpha_k |k\rangle - \sin\left(\frac{3\tau}{2}\right) \alpha_{x_j} |x_j\rangle ,$$

where we can see the effects of amplifying the amplitude of the correct seed $|x_j\rangle$ from section 6.1.3 in the book *Quantum Computation and Quantum Information* [7].

The remaining problem is how to construct the oracle O. The circuit in the State Identification stage can check if the input is a correct guess in polynomial time. We can modify it to be the oracle O.

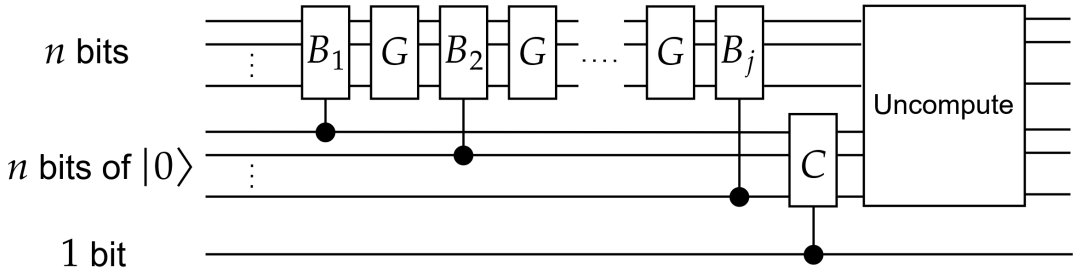

Figure 7: Circuit for Oracle O

From Figure 7, the B_i gates and G gates are the same as the gates described in the State Identification stage. After the circuit of the State Identification stage, the state will be:

$$|\gamma\rangle = \sum_{k=0, \neq x_j}^{2^n-1} \alpha_k |k\rangle |y_k\rangle \otimes \left(\frac{|0\rangle - |1\rangle}{\sqrt{2}}\right) + \alpha_{x_j} |x_j\rangle |11...11\rangle \otimes \left(\frac{|0\rangle - |1\rangle}{\sqrt{2}}\right)$$

The C gate negates the last bit based on the first n bits of the working register W as the following:

$$C(|y_x\rangle \otimes \left(\frac{|0\rangle - |1\rangle}{\sqrt{2}}\right)) = \begin{cases} |y_x\rangle \otimes \left(\frac{|0\rangle - |1\rangle}{\sqrt{2}}\right) & \text{if } |y_x\rangle \neq \left|y_{x_j}\right\rangle \\ |y_x\rangle \otimes \left(\frac{-|0\rangle + |1\rangle}{\sqrt{2}}\right) & \text{if } |y_x\rangle = \left|y_{x_j}\right\rangle . \end{cases}$$

Because of phase kickback, the equation above could also be:

$$C(|y_x\rangle \otimes (\frac{|0\rangle - |1\rangle}{\sqrt{2}})) = \begin{cases} |y_x\rangle \otimes (\frac{|0\rangle - |1\rangle}{\sqrt{2}}) & \text{if } |y_x\rangle \neq |y_{x_j}\rangle \\ -|y_x\rangle \otimes (\frac{|0\rangle - |1\rangle}{\sqrt{2}}) & \text{if } |y_x\rangle = |y_{x_j}\rangle. \end{cases}$$

The state after the C gate will be:

$$|\gamma\rangle = \sum_{k=0,\neq x_j}^{2^n-1} \alpha_k |k\rangle |y_k\rangle \otimes (\frac{|0\rangle - |1\rangle}{\sqrt{2}}) - \alpha_{x_j} |x_j\rangle |y_{x_j}\rangle \otimes (\frac{|0\rangle - |1\rangle}{\sqrt{2}})$$

Then we need to uncompute the state identification part. The oracle O that we constructed successfully flips the sign for the correct seed x_j.

5.2 Complexity Analysis

For oracle O, each G gate is an exponentiation which is $\mathcal{O}(n^3)$, if we calculate exponentiation by squaring. The B_i gate is the comparison between the value and half of the prime P. We can use subtract them in 2's complement and check the sign of the result to see if the value satisfies the condition. This gate will take $\mathcal{O}(n)$ gates. The C gate is a controlled-U operation which takes $\mathcal{O}(n)$ from section 4.3 in the book *Quantum Computation and Quantum Information* [7]. Since the oracle needs $\mathcal{O}(n)$ exponentiations, in total, its runtime complexity is $\mathcal{O}(n^4)$. The uncompute part takes the same time complexity as the state identification part, which is $\mathcal{O}(n^4)$. The overall time complexity of the oracle O is $\mathcal{O}(n^4)$.

In the Grover's iterator S, the Hadamard gate H and the gate which negates $|x\rangle$ where $x > 0$, take $\mathcal{O}(n)$ gates like the controlled-U gate mentioned above. The overall time complexity is $\mathcal{O}(n^4)$.

Therefore, the overall time complexity for the corrected algorithm is:

$$\mathcal{O}(n) + \mathcal{O}(n^4) \cdot \mathcal{O}(2^{n/2}) = \mathcal{O}(n^4 2^{n/2})$$

6 Conclusion

We proved that the algorithm in Guedes, Assis, and Lula's paper cannot work, because of the effects of the additional working bits. We then proposed a corrected version from ideas of solving NP-complete problems by Grover's search algorithm. The corrected quantum attack has a time complexity of $\mathcal{O}(n^4 2^{n/2})$, while that of the classical attack is $\mathcal{O}(n^4 2^n)$. The quantum attack is much better than the classical attack, with a square root time reduction. However, since the time complexity is still in the class of exponential time, we cannot break the Blum-Micali Pseudorandom Number Generator in a practical time when we choose a sufficiently large n like 64. The Blum-Micali Pseudorandom Number Generator is still quantum resistant.

In the future, we may try to simulate the algorithm on Python's quantum computing library Qiskit to verify the algorithm together with its time complexity analysis. The fact that Blum-Micali pseudorandom number generator is still quantum resistant can give us inspiration that discrete logarithms and prime factorization could still be used in cryptographical applications even if there is polynomial algorithm which can solve these problems.

7 Acknowledgement

I would like to extend my deepest gratitude to my adviser, Dr. Joshua Holden, whose guidance, support, and expertise were pivotal throughout the entirety of this research journey.

References

[1] Leonard M. Adleman and Ming-Deh A. Huang. Function field sieve method for discrete logarithms over finite fields. *Information and Computation*, 151(1):5–16, 1999.

[2] Manuel Blum and Silvio Micali. How to generate cryptographically strong sequences of pseudorandom bits. *SIAM Journal on Computing*, 13(4):850–864, 1984.

[3] Internet Engineering Task Force. rfc4949. `https://datatracker.ietf.org/doc/html/rfc4949`, May 2022.

[4] Lov K. Grover. A fast quantum mechanical algorithm for database search. In *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing*, STOC '96, page 212–219, New York, NY, USA, 1996. Association for Computing Machinery.

[5] Elloá B. Guedes, F. M. De Assis, and Bernardo Lula. Quantum attacks on pseudorandom generators. *Mathematical Structures in Computer Science*, 23(3):608–634, 2012.

[6] R. Jozsa. Quantum factoring, discrete logarithms, and the hidden subgroup problem. *Computing in Science Engineering*, 3(2):34–43, 2001.

[7] Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information*. Cambridge University Press, 2021.

[8] Peter W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Journal on Computing*, 26(5):1484–1509, 1997.

[9] Gillian Smith. An analog history of procedural content generation. In *Proceedings of the 10th International Conference on the Foundations of Digital Games (FDG 2015)*. Pacific Grove, CA, USA, June 22-25, 2015.

[10] Andrew C. Yao. Theory and application of trapdoor functions. *23rd Annual Symposium on Foundations of Computer Science (SFCS 1982)*, page 80–91, 1982.

8 Appendix

8.1 Appendix A

We did the test by generating a long enough random bit string with a random seed from Python's standard Mersenne Twister Pseudorandom number generator. For Figure 8, we ran the tests for all the primes less than 5000. For Figure 9, we added random samples between 5000 and 40000. In the two figures, $y = \log_2(x)$. From the two figures, we can see that breaking the Blum-Micali random number generator approximately needs $\mathcal{O}(\log p)$ bits of the output.

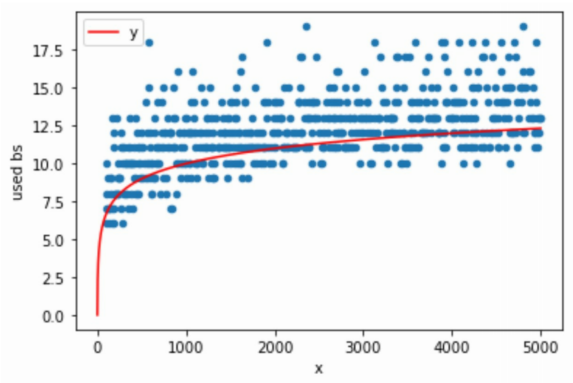

Figure 8: Number of bits for primes less than 5000

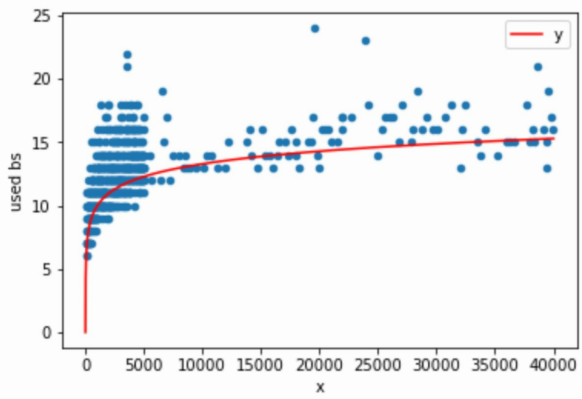

Figure 9: Number of bits for all primes under 5000 and random primes from 5000 to 40000