Abstract
Public-key cryptosystems base their security on well-known number-theoretic problems, such as factorisation of a given number $n$. Hence, prime number generation is an absolute requirement. Many prime number generation techniques have been proposed up-to-date, which differ mainly in terms of complexity, certainty and speed. Pocklington’s theorem, if implemented, can guarantee the generation of a true prime. The proposed implementation exhibits low complexity at the expense of long execution time.

Keywords: Primality, Prime Number Generation, Computational Number Theory.

1 BACKGROUND

Testing the primality of a number is not the same as finding its prime factors. However, for applications such as public-key encryption, knowledge of primality is sufficient. Many primality tests have been devised until now, some better than others. They can be classified into two main categories: deterministic and probabilistic.

- Deterministic
  They provide the definite answer to the question whether a number $n$ is prime or not. The older ones required, at some point, a factorisation of a number related to $n$. This factorisation can prove to be as hard as factoring $n$ itself.
  Such paradigms are the elliptic curve test\(^1\), the APR test and the Lucas-Lehmer test for Mersenne primes.

- Probabilistic
  Generally speaking, they are much easier to implement and their execution time is significantly less than deterministic ones. Once a number $n$ has passed such a test, its probability of being prime is very high, nevertheless, its primality cannot be guaranteed. These methods make it feasible for large numbers to be tested for primality, although it is not possible to provide a certain answer.
  Tests that belong to this category are the APRCL test\(^2\), the Solovay-Strassen test\(^3\), the Lehmann test, the Lucas pseudoprinality test and the Rabin-Miller test\(^4\).

---

\(^{1}\)Although a deterministic test, its running time is probabilistic. For more information, readers are referred to [6].

\(^{2}\)There also exists a deterministic, but less practical version.

\(^{3}\)Also known as Euler’s pseudoprinality test.

\(^{4}\)Also known as Strong pseudoprinality test.
2 PRIME NUMBER GENERATION

2.1 Prime-Generating Sequences

Probably the most famous one has been proposed by Euler: \( a_n = n^2 + n + 41 \). This polynomial will produce an uninterrupted sequence of 80 primes for \( n = −40, −39, \ldots, 39 \) [3]. However, the sequence is symmetric and since each prime occurs twice, 40 primes are actually produced.

Another sequence is \( c_i = (c_0)^{c_{i-1}} - 1 \) [3]. Starting with \( c_0 = 2 \), it does produce a sequence of primes, but it is not known whether it produces just primes, because the numbers grow extremely rapidly:

- \( c_0 = 2 \)
- \( c_1 = 3 \)
- \( c_2 = 7 \)
- \( c_3 = 127 \)
- \( c_4 = 170, 141, 183, 460, 469, 231, 731, 687, 303, 715, 884, 105, 727 \)
- \( c_5 > 10^{51}, 217, 799, 719, 369, 681, 879, 879, 723, 386, 331, 576, 246 \).

The number \( c_4 = 2^{127} - 1 \) is actually the 12th Mersenne prime, that was proved to be prime by Lucas in 1876. It is very unlikely that the primality of the number \( c_6 \), or the ones that follow it, could ever be determined [3].

2.2 Modern Prime Number Generation Methods

It seems that most people are in favour of the following way of generating prime numbers:

1. Generate a random, odd number.

2. Test for primality with a method that is guaranteed to have a high hit rate.

Fairly quick methods for testing the primality of a given number \( n \) are the Solovay-Strassen method [5], the Lehman test [4] and the Rabin-Miller test [5], although the latter seems to be the most widely used one.

However, this method is still not guaranteed to produce a true prime number, although the probability of producing a pseudoprime is very small. Perhaps a better approach would be the following [6]:

1. Generate an odd integer \( n \).


   Use a combination of the Rabin-Miller test and the Lucas pseudoprIMALITY test.


Suppose that the method followed for generating a prime number has actually produced a pseudoprime \( n \). Although the probability of such a case to occur is very low, it is still worth looking at the consequences that it could possibly have. For most public-key cryptosystems this means jeopardising security. The risk of an encrypted message to be successfully cryptanalysed is ‘inversely proportional’ to the ease of factoring \( n \). Some numbers are easier to factor than others. For example, by using the Number Field Sieve (NFS), Fermat numbers are easier to factor than hard numbers\(^5\) [5].

\(^5\)A hard number is one that does not have any small factors and is not of a special form that allows it to be factored more easily.
3 POCKLINGTON’S THEOREM

3.1 Introduction
When the generation of a true prime is an absolute requirement, a way of achieving this is by using Pocklington’s theorem [6]. This states that:

Let \( p \) be an odd prime, \( k \) a natural number such that \( p \) does not divide \( k \) and 1 < \( k < 2(p + 1) \), and let \( n = 2kp + 1 \). Then, the following conditions are equivalent:

1. \( n \) is prime.
2. There exists a natural number \( a \), \( 2 \leq a < n \) such that

\[
a^{kp} \equiv -1 \pmod{n}
\]

\[
\gcd(a^k + 1, n) = 1
\]

An algorithm derived from the above theorem is the following [6]:

1. Choose, for example, a prime \( p_1 \) with \( d_1 = 5 \) digits. Find \( k_1 < 2(p_1 + 1) \) such that \( p_2 = 2k_1p_1 + 1 \) has \( d_2 = 10 \) digits, or \( d_2 = d_1 - 1 = 9 \) digits and there exists \( a_1 < p_2 \) satisfying the conditions \( a_1^{k_1}p_1 \equiv -1 \pmod{p_2} \) and \( \gcd(a_1^{k_1} + 1, p_2) = 1 \). By Pocklington’s theorem, \( p_2 \) is prime.

2. Repeat the same procedure starting from \( p_2 \) to obtain the primes \( p_3, p_4, \ldots, p_n \). In order to produce a prime with 100 digits, the process must be iterated five times, as shown in table 1.

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Number Of Digits</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( d_2 = 2 \cdot d_1 = 10 ) digits</td>
</tr>
<tr>
<td>2</td>
<td>( d_3 = 2 \cdot d_2 = 20 ) digits</td>
</tr>
<tr>
<td>3</td>
<td>( d_4 = 2 \cdot d_3 = 40 ) digits</td>
</tr>
<tr>
<td>4</td>
<td>( d_5 = 2 \cdot d_4 = 80 ) digits</td>
</tr>
<tr>
<td>5</td>
<td>( d_6 = 100 ) digits</td>
</tr>
</tbody>
</table>

Table 1: Iterations.

In the last iteration, \( k_5 \) should be chosen so that \( p_6 = 2k_5p_5 + 1 \) has 100 digits.

3.2 Implementation
In what follows, the existence of a multiple-precision integer library is assumed, which will enable the handling of large integers. Fast exponentiation and fast modular exponentiation techniques will be used, since the powers are expected to be quite large (interested readers are referred to [6]).
3.2.1 Problems Faced

When the algorithm was initially used, a few digits would suffice for generating a secure key. Nowadays, more computational power is available at relatively low cost and more efficient factoring algorithms have been discovered, thus creating a demand for larger numbers that will be harder to factor under the current circumstances.

The only difficult part in the implementation stage is the evaluation of $\gcd(a_{i+1}^{k_i} + 1, p_i+1)$, and more specifically, the calculation of $a_{i+1}^{k_i}$. Even if $a_i$ is a 1-digit prime, the size of $k_i$ is expected to be of approximately the same number of digits as $p_i-1$, since $p_i = 2^{k_i}p_i+1$. To give an approximation of the order of magnitude, the $37^{\text{th}}$ Mersenne prime is $M_{37} = 2^{3021377} - 1$ and has 909526 digits [6]. Supposing that $k_i$ was a 20-digit number, the calculation of $a_{i+1}^{k_i}$ would either be impossible, or would take far too long, even by using a fast exponentiation technique.

However, the above calculation would be feasible by using a fast modular exponentiation technique, provided that the condition can be re-written in an appropriate form. For example, proving that if

$$\gcd((a_{i+1}^{k_i} \mod p_i+1) + 1, p_i+1) = 1 \quad (3)$$

then also

$$\gcd(a_{i+1}^{k_i}, p_i+1) = 1, \quad (4)$$

would solve the problem, since $a_{i+1}^{k_i} \mod p_i+1$ can be calculated very efficiently.

Another issue worth mentioning is the procedure followed for choosing $a_i$ and $k_i$. For efficiency reasons, it is not worth calculating any power of $a_i$ for large values of $a_i$. For that reason, in the calculation of $a_{i+1}^{k_i}$ and $a_{i+1}^{k_i}$, $a_i$ will take the values 3, 5 and 7, cyclically. Hence, once a $k_i$ has been generated, the theorem’s conditions will be tested for each one of the values of $a_i$. If they are not satisfied, a new $k_i$ will be generated and the process repeated.

Choosing $k_i$ by algebraic methods is impossible because of the amount of numbers that are to be tested. The use of random numbers is preferred, as it will speed the execution up.

3.2.2 Auxiliary Functions

The algorithm presented in section 3.2.3 is an implementation of Pocklington’s theorem for generating a 100-digit prime. It is written in pseudo-C++ notation. Moreover, for simplicity reasons, the existence of the following functions is assumed:

- $\text{SetNewNumOfDigits}(d_{\text{old}}, d_{\text{new}})$: Checks the number of digits of $p_i$ ($d_{\text{old}}$) and sets the correct number of digits for $p_{i+1}$ ($d_{\text{new}}$).
- $\text{DigitsOf}(\text{num}, \text{dig})$: Checks if $\text{num}$ has $\text{dig}$ number of digits.
- $\text{FastModExp}(\text{base}, \text{exp}, \text{m})$: Performs $\text{base}^{\text{exp}} \mod \text{m}$, using a fast modular exponentiation technique.
- $\text{Gcd}(\text{num1}, \text{num2})$: Finds the greatest common divisor of $\text{num1}$ and $\text{num2}$, using Euclides’ algorithm [6].
- $\text{NextBase}(\text{base})$: Examines the current value of $\text{base}$ and returns the next one, cyclically.
- $\text{GenerateRandomK}($old, new$)$: Given an old-digit long $p_i$, it randomly generates a $k$, such that $p_{i+1} = 2^{k}p_i + 1$ will approximately have new digits.
3.2.3 Algorithm

//5 iterations are needed for a 100-digit prime
for(j=0; j<5, ++j){
  p1 = p2;

  k = 1; //Initialise
  primeFound = false;
  while(!primeFound){
    twop1 = 2 * (p1 + 1);
    SetNewNumOfDigits(d, d1);
    while(!primeFound){
      p2 = (2 * k * p1) + 1;
      kp1 = k * p1;
      if(DigitsOf(p2, d1)){
        a = 3;
        for(i=0; i<2; ++i){ // Since there are only 3 bases
          a_kp1 = FastModExp(a, kp1, p2);
          p2min1 = p2 - 1;
          //Check if a_kp1 == -1 mod p2
          if(a_kp1 == p2-1){
            primeFound = true;
            cout<<"Prime found: "<<endl<<p2<<endl<<endl;
            break;
          }
        }
        a = NextBase(a);
      }
    }
    k = GenerateRandomK(d, d1);
  }
}

3.3 Performance

This algorithm trades certainty for speed. Using the methods mentioned in section 3.2.1, one should be able to generate 100-digit prime numbers within a few seconds, or even less than that. The algorithm was implemented in C++, using the BigNum library for multiple precision integer representation [1]. When run on a Sun Sparc Ultra 10, the generation of a 100-digit prime ranged from 1 to 31 minutes, with an average of 15.86 minutes in 100 runs. This result was obtained by checking only the first condition, since the second one could not be implemented. Moreover, the program was experimentally modified so as to generate a 160-digit prime and it took about 1 day of continuous execution, before it managed to come up with one.

Up to date, there is no formal proof that the first condition implies the second, or vice versa. However, given the fact that if $p_i$ is not prime then $p_{i+1}$ will not be prime either, together with the results obtained, deduces that condition (1) (page 3) is a strong condition for primality testing. Therefore, since condition (2) (page 3) exhibits some implementation difficulty, it may be possible to replace it with a primality-proving method (the Elliptic Curve Method [2], for instance). The reason for doing so, is because once a number $n$ has passed condition (1), it is almost definite that it is a prime, since each number is generated so as to be of a certain form that has a high probability of being a prime. Furthermore, this is reinforced by the experimental results.

However, the algorithm’s performance in the second case raises an interesting point: --Does the implementation of Pocklington’s theorem have an ‘expiration date’? If, for instance, two 160-digit primes
are needed for generating a secure key in the near future, using this method is totally impractical, since it is a very time-consuming process.

3.4 Examples

The following two examples exhibit the generation of a prime, by showing the intermediate results as well.

3.4.1 First Example

Starting from $p_1 = 97711$, the following have been produced:

- $a_1 = 5$
- $k_1 = 22548$
- $p_2 = 4406375257$

- $a_2 = 5$
- $k_2 = 5672269218$
- $p_3 = 49988293466475878053$

- $a_3 = 3$
- $k_3 = 30278440082130267825$
- $p_4 = 3027135097065267631182877686789859089451$

- $a_4 = 5$
- $k_4 = 1813854257255414712781164661420477580766$
- $p_5 = 10981563766198237482064153956177259647573196407858955619248482635330_{-784142198933}^{-}$

- $a_5 = 3$
- $k_5 = 603977763629481446952$
- $p_6 = 13265237852658384156030973594451425884194920866937875781392644397088_{-20318686315226334078935741004433}^{-}$

3.4.2 Second Example

Starting from $p_1 = 97711$, the following have been produced:

- $a_1 = 5$
- $k_1 = 30036$
- $p_2 = 5869695193$

- $a_2 = 3$
- $k_2 = 2436120302$
- $p_3 = 28598567252438216573$

- $a_3 = 3$
- $k_3 = 23378902714462628788$
- $p_4 = 133720624313553973012472292260097007049$

- $a_4 = 5$
- $k_4 = 4848112241775702700627119663577344917472$
- $p_5 = 12965851914248613763710377243469059444879129989254423118286594309361_{-798214520257}^{-}$

- $a_5 = 5$
- $k_5 = 42141446863948470528$
4 CONCLUSION

Most public-key cryptosystems base their security on the properties of prime numbers. Knowing two large primes \( p \) and \( q \) (of 100 digits each, for example), means that their product, \( n \), can easily be calculated. Nevertheless, the reverse is very difficult to do, given the current factorisation techniques and existing computational power.

Nowadays, the most popular method for generating prime numbers is by generating a random, odd number, \( n \) and then testing its primality. Quite a few algorithms for testing the primality of a number are available, most of which are very fast, since they do not require any kind of factorisation. However, they trade certainty for speed (higher certainty usually increases implementation complexity), which means that there still exists the probability—although a very low one—of a pseudoprime to be produced. Producing a pseudoprime instead of a true prime reduces the degree of security on most cryptosystems.

One of the available ways of producing a prime with 100 percent certainty is to implement Pocklington’s theorem. On the one hand, the implementation is fairly simple. On the other, it has the drawback of quite a long execution time. This limits the scope of application to security-critical cases where time is of not much importance.
References


