Constructing Large Random Primes for RSA using Deterministic Methods: Maurer’s Algorithm
(Project SK-3)

15 December 2000

Michael Hammer
Issam Andoni
# Table of Contents

1. Abstract ......................................................................................................................... 1
2. Introduction ...................................................................................................................... 1
   2.1 Value of Public Key Cryptography .............................................................................. 1
   2.2 RSA Reliance on Large Prime Numbers ..................................................................... 2
   2.3 What Makes a Prime Strong ....................................................................................... 2
   2.4 Two Approaches of Finding Large Primes ................................................................. 3
3. Maurer’s Algorithm ............................................................................................................ 3
   3.1 Overview of Maurer’s Random Prime Procedure ......................................................... 3
      3.1.1 Exponentiate ........................................................................................................ 4
      3.1.2 Random .............................................................................................................. 4
      3.1.3 PrimeTest .......................................................................................................... 4
      3.1.4 TrialDivision ...................................................................................................... 5
      3.1.5 G_opt .................................................................................................................. 5
      3.1.6 GenerateSizeList .............................................................................................. 5
      3.1.7 CheckLemma1 ................................................................................................. 5
4. Outline of Technical Approach .......................................................................................... 5
   4.1 Objective of the Project .............................................................................................. 5
   4.2 Expected Results ........................................................................................................ 5
   4.3 Resources .................................................................................................................... 5
      4.3.1 Hardware .......................................................................................................... 5
      4.3.2 Software .......................................................................................................... 6
   4.4 Overview of Program Design ..................................................................................... 6
   4.5 Testing Procedures ..................................................................................................... 6
      4.5.1 Main Program Level ........................................................................................... 6
      4.5.2 Random Prime Procedure Level ......................................................................... 7
      4.5.3 Function Level ................................................................................................... 7
   4.6 Program Performance .................................................................................................. 8
      4.6.1 Speed ............................................................................................................... 8
      4.6.2 Memory Usage .................................................................................................. 8
      4.6.3 Design and Development Issues ........................................................................ 8
5. Summary of Program Results ........................................................................................... 10
   5.1 Review of Data Collected .......................................................................................... 10
   5.2 Analysis of Data ........................................................................................................ 11
   5.3 Degree of Agreement With Expected Results ............................................................. 11
   5.4 Practical Application of Results ................................................................................. 11
6. Conclusions ....................................................................................................................... 11
7. References ......................................................................................................................... 12

Appendix A: Original Source Code
Constructing Large Random Primes for RSA using Deterministic Methods: Maurer’s Algorithm

1 Abstract

Generating keys for cryptographic applications requires the ability to generate sufficiently large random prime numbers. The degree to which they are both large and random increases the security of the keys. The degree of certainty that the number really is prime should have a probability of 1.0 or fall short within an extremely small fraction. Two practical approaches were developed. Project SK-2 randomly chooses a number and then attempts to test with some degree of certainty that the number is prime using the Miller-Rabin method. This project implements Maurer’s deterministic method for constructing a prime number together with a certificate of primality. Maurer’s 1989 method uses a recursive algorithm based on Pocklington’s theorem published in 1914.

2 Introduction

Today, we live in an Information Age where the accuracy of information must be assured in order for the processes and supported systems to be reliable. Trust in the system forms the basis of financial networks and decisions made in many other situations. That assurance is needed both when information is in storage and when it is in transmission. In the past, trust in the messenger or trust in the difficulty of forgery of physical documents formed the basis of reliability. Also, the limited means of performing computation allowed simple systems of cryptography to be used.

When using computers and networks, the volatile nature of the medium required improved means of security. That resulted from the degradation due to improved computation as well as the ability to easily manipulate information stored or transmitted in “soft” form. Improved algorithms and hardware or software machines in the form of secret (symmetric) key technology partially resolved some of the necessary conditions for large-scale deployment of secure information systems. However, it did not solve the scaling problems of key distribution systems.

2.1 Value of Public Key Cryptography

It is possible to develop secure systems to provide confidentiality, authentication, integrity, and non-repudiation services based on secret key systems. However, the ability to create, distribute, and manage keys becomes problematic as the scale of the system becomes very large. Two aspects are of particular concern. One, for two parties to share a common key they must both be registered within the same system, or the process of distributing keys becomes more complex, perhaps even manageable. Second, if the key distribution center (KDC) is
compromised, the trust in the entire system addressed by those keys fails until the mechanism of trust by which the KDC distributes keys can be re-established. Because the secret key between the distribution system and the user may require manually reestablishing, perhaps through physical appearance or assurance means, that the person or device receiving the key is valid. In addition, the mechanical failure of the KDC will result in a denial of service unless additional protection measures are taken.

Public (asymmetric) key cryptography addresses both problems. First, the ability to communicate securely with any party in the system relies on the ability to openly distribute the public key by publishing it on the Web. Second, the failure of a Web site with the public keys would not compromise any user and would not deny any user the ability to continue using their existing private keys. Also, because the key-pair generating components do not have to be on-line and can be managed more securely off-line, the likelihood of failure and compromise is reduced.

2.2 RSA Reliance on Large Prime Numbers

The basis for public key cryptography relies on the complexity of reversing a mathematical calculation, such as from the difficulty of factoring large prime numbers. Unlike traditional encryption methods that rely on substitution and permutation operations to “scramble” the bits in a complex non-linear fashion, RSA relies on the use of an exponential modular operation. The base of the modular operation “N” is the product of two large prime numbers. The exponent used to decrypt (private key) is produced from the totient of N also produced from those two large primes. Therefore, it is extremely important to properly choose “strong” primes to create strong keys.

2.3 What Makes a Prime Strong

A strong prime is one that meets the current definition of “large,” which is now a minimum of 100 bits. It must also have the following characteristics:

- p-1 has a large prime factor r, where r-1 has a large prime factor t
- p+1 has a large prime factor.

The above protect against known attacks, such as the super-encryption attack, Pollard’s p-1 method, and Cyclotomic Polynomial Method. In Maurer’s algorithm, he builds primes based on ensuring that p-1 has at least one large prime factor for each level of recursive construction.

An additional requirement is to be sure that the set of primes chosen is broadly and randomly distributed across the possible set of prime numbers within the desired range defined by the bit length. That protects against the possibility that the same “strong” primes have a higher probability of being reused, which would result in a higher possibility of reconstructing keys using these “probable” primes.
2.4 Two Approaches of Finding Large Primes

There are two approaches to finding strong prime numbers. The first is to randomly choose a number and apply a probability test that has a high probability of indicating that the number chosen was in fact prime. One would then need to test the prime to be sure that it was a strong prime, although this latter requirement with a small risk could be skipped because most primes are strong. However, the additional time spent testing the prime would be worthwhile.

The second approach is to construct a prime by finding prime numbers of $p-1$ that are also relatively strong primes. This can be done recursively by dividing the problem space into successively smaller partitions that when reconstructed, produce a prime with the requisite strength. This is in essence the approach taken by Maurer.

3 Maurer’s Algorithm

One possible means of constructing strong primes could have been a pure bottom-up approach. That approach would start with a good size series of small prime numbers as factors and multiply to create a smaller set of larger prime factors, and so on until a single prime of the appropriate size resulted. At least one factor should be relatively large compared to the other primes used to construct the higher prime number. The difficulty with this approach would be that the resulting prime number may not be broadly distributed, but rather, there may be a higher probability of converging on the same smaller set of large primes.

Instead, Maurer started with a top-down approach that recursively divided the current level’s desired prime size using randomly chosen fractions until the lowest level’s prime size was small enough to randomly choose and verify. Thus the distribution problem was addressed. A bottom-up construction approach can then be used. The primes at each level are used as factors to build the next level prime. At the smallest level of recursion, non-constructed primes can be tested, by dividing each of them by all smaller known primes from a list. At each level of construction, Pocklington’s Theorem is used to validate that the resulting number is prime. The proof of primality consists of the successive proofs of primality at each level.

3.1 Overview of Maurer’s Random Prime Procedure

The basic recursive element of Maurer’s algorithm is the Random Prime procedure. That procedure consists of two parts. The first part is used when the upper limit on the range of the desired prime is small enough to randomly select and then test using a list of known primes. That iteration is then ended with the proven prime returned as one of the elements of the factor list and the next higher level.

The second part is used when the size range is still too large. At a high level the following actions are performed:

- A series (max 10) of fractions is randomly generated that add up to just under 1.0
A new desired size of primes less than half of the square root of the product of the
current range limits is generated and a limit is calculated to the number of prime
divisors needed during a later trial division

The new prime size is then partitioned according to the generated fractions

For each partition, a range +/- is used in the recursive call to another lower iteration
of RandomPrime

The returned prime is multiplied against a factor list (F)

When the product of all the factors has been computed, another range is generated
using the range limits and the factor product at that level and a random number (R) is
then generated within that range

Then a potential prime n=2*R*F+1 is generated

A Trial Division is used to provide a quick elimination of composite numbers, then a
check based on Pocklington’s Theorem is used

If that test is passed, then n is returned to the next higher level as the resulting prime.
If not, a new R is chosen, n generated, and tested again.

Note that the test of Pocklington’s Theorem demonstrates that if each prime factor is of
the form p=mF+1 and n=2RF+1, and F > sqrt(n) then n is prime. Alternatively, if F is odd and
F > R, then n is prime. The construction above is designed to choose R such that it is less than F.
Also, since F is a product of primes it will be odd.

3.1.1 Exponentiate

This function using the generated list of fractions from GenerateSizeList partitions the
current prime size into smaller values used to create the factor list.

3.1.2 Random

This function is used in three places to randomly choose a number within a defined range.
In the first part, it is used to choose the smallest prime used as a factor. In the second
part, it is used to select R and the value “a” used in CheckLemma1.

3.1.3 PrimeTest

This function is used to test the lowest level recursion of prime numbers generated, when
number sizes are within the computer’s limits.
3.1.4 TrialDivision

This function is used to test higher levels of prime numbers, when the number sizes are beyond the normal computer limits, using “big integer” routines.

3.1.5 G_opt

This function is used to determine how many known primes to try in TrialDivision. It is based on a fraction of the square of the number of bits in the number tested.

3.1.6 GenerateSizeList

This function returns an array of fractions whose total is less than, but near 1.0 that are used by Exponentiate to partition.

3.1.7 CheckLemma1

This function uses a base “a” to test first whether \( a^{n-1} \mod n = 1 \), then checks each factor \( q \) used in \( \gcd(a^{(n-1)/q} - 1 \text{ and } n) = 1 \). This function proves primality by Pocklington’s Theorem.

4 Outline of Technical Approach

4.1 Objective of the Project

The project objective was to develop a software implementation of Maurer’s method and measure times to generate primes of different sizes.

4.2 Expected Results

It is expected that the results will range from small fractions of a second for small prime sizes and increase in a non-linear fashion to possible tens of seconds for primes in the range of 1024 to 2048 bits.

4.3 Resources

The original intent was to run this on a PC; however, the difficulty in loading libraries for doing large number computations forced the use of a Unix platform in a lab at GMU.

4.3.1 Hardware

The original platform was Pentium II 233 MHz Processor, 64 Mbytes RAM. The Unix platform was a Sun W Ultra-4 Sparc machine, 2048 Mbytes RAM (at 24% shared utilization).
4.3.2 Software

The original operating system was Windows 95. The original compiler was Microsoft Visual C++ 6.0. The final operating system used was Sun OS Release 5.7. Standard C++ Libraries were used to make it more portable. However, using libraries specific to a Unix platform prevent the source code from being very portable.

Software Required: The NTL Libraries were used for functions to do computations on large integers. This will include doing addition, subtraction, multiplication, division, square root, powermod, number of bits, and random number generation for large numbers. We had to develop our own prime test and exponentiation routines.

4.4 Overview of Program Design

The main program is used to set seeds for the large random number generation, get input form the user on the desired range of the prime, measure the time it takes to generate a prime, and outputs the result. These actions are performed in a loop so that the user may repeat the prime generation any number of times.

The main input from the user is the number of bits in the range 8 to 2048, which are manually entered at the screen prompt. The corresponding numbers P1 and P2 will be generated for input to the RandomPrime procedure.

The output displayed to the screen is the generated prime number P and the time to generate that number T. For numbers less than 1024, a loop was used and an average time is calculated by dividing by the iterations in that loop.

The original intent was to run the program multiple times and record prime versus time in an Excel spreadsheet where the curve can be graphed for the report. Manual inputs provide early feedback regarding the time to perform a single prime calculation, which permits judicious choice of the series of input ranges to be used. The value “num” will be used to specify the number of times that a random number will be generated in that range to aid in calculating an average run time for that range.

The originality of this project is in writing a working program as the guidance provided for each of the eight functions leaves much work to the developer to produce an implementation. Functions will be divided evenly among team members.

4.5 Testing Procedures

The validity of the program must be tested at the following three levels:

4.5.1 Main Program Level

At the main program level, the ability to accept input, call the RandomPrime procedure, measure the time duration of that procedure, and generate output must be tested. A stub
for the procedure was used to perform multiple runs and determine the consistency and minimum time it takes to call and return from the procedure. The average minimum time to call the stub could be subtracted from the runs of the complete procedure, however the time was not measurable and therefore was not used.

4.5.2 Random Prime Procedure Level

At the RandomPrime procedure level, statements to display certain variables were used to follow the routine during run time and to determine where the procedure failed, when that occurred. Those values also allowed us to determine when the subroutines were not functioning properly or where the values produced were not in the desired range, given the intent of the procedure. In several cases constants were modified to get workable results. Furthermore, the application of the procedure to a known prime was used to test the validity of the overall logic. Maurer’s paper is very specific about the algorithm at this level, so a code walk-through and analysis will be discussed in the report.

4.5.3 Function Level

At the function level, each of the eight functions was tested separately using test main routines to call the functions one at a time to verify correct results. The following describes the general approach to evaluating the correct operation of each function:

- **Exponentiate:** This function was tested on smaller values with known results.

- **Random:** This function was tested many times and a histogram of results visually examined to verify that results are uniformly distributed.

- **PrimeTest:** This function was tested against known complex and prime numbers to determine whether the correct boolean value is returned.

- **TrialDivision:** This function will be tested against values of a and b with known boolean results.

- **G_opt:** This function is based on the simpler FastPrime g_opt described in Maurer’s paper. Results were compared to a calculator.

- **GenerateSizeList:** This complex function is based on Appendix I of the paper. A code walk-through will ensure that it correctly implements calculations. Display statements were used to check the values produced during operation.

- **CheckLemma1:** This complex function is based on Section 2 of the paper. A code walk-through will ensure that it correctly implements calculations. Also, the powermod operations were checked against our own program for results.
4.6 Program Performance

4.6.1 Speed

See Section 5.1 for our data on run times dependent on the requested key size.

4.6.2 Memory Usage

4.6.2.1 Stored File Size

The source files, excluding the library files occupy 58.7 Kbytes on the hard drive. The executable file occupies 465,718 bytes.

4.6.2.2 Run-Time RAM Requirements

The amount of RAM used depends on the number of variables used at any given time and can be modified by the needs of the operating system to swap data and instructions in and out of memory. We did not have a convenient way of measuring this on a shared computer.

4.6.3 Design and Development Issues

4.6.3.1 Library Difficulties

The ability to implement this algorithm for large key sizes depends on the availability of routines to do large number computations. By large, we mean that the size of integers exceeds the limit on integer size due to the design constraints of the given platform. If a platform has types and operators defined that can handle numbers of arbitrary size, then no additional libraries would be needed.

We originally specified the use of PGP, but found that certain key operations were not supported. We therefore looked at the Lidia library. However, our attempts at loading Lidia failed. In one case, it crashed Windows 95 and the computer had to be rebuilt after reformatting the hard drive. Ultimately, we switched to the Unix platform and found that the NTL library was able to be loaded and used.

4.6.3.2 Compromises in Procedure Design

In the Exponentiate routine, our library did not support fractional exponents, so we devised an approximation that used the higher order bits of the number and the normal size integer exponentiation to produce a mini-result. That result then was shifted back up to half the original bit size calculated from the downshifting to get the higher order bits. The result is a reasonable order of magnitude but may get worse as the number gets larger. This is one area where the implementation could be improved.
The `G_opt` routine was devised from guidance provided in the Maurer Fast Prime description. There was not real guidance provided for the primary Maurer description. This routine used the number of bits of the number to determine the maximum size of the primes used in trial division.

GenerateSizeList routine follows the method described in Appendix 1 of Maurer’s paper. However, instead of using a comparison of the last versus the remaining number, we used a constant threshold for the remaining to ensure that the sum of the fractions was within a specified tolerance of 1.0. Otherwise, there did not appear to be sufficiently large F produced.

When selecting the range to be passed down to the next recursion of `RandomPrime`, we had trouble with performing integer division and multiplication, in that the truncation of the result produced a range of zero. We had to modify the routine to ensure that the range was large enough to encompass a random prime. We also had to guard against the possibility that the lower bound would be a negative number.

Maurer’s paper also included a procedure called `FastPrime` that contains only 6 functions. If the implementation of the longer procedure proved too time-consuming, in either development or use, or lead to unreliable results (possibly due to inability to implement a function correctly) the alternate procedure could have been tried. Because the program seemed to be working for the most part this faster routine was not attempted.
5 Summary of Program Results

5.1 Review of Data Collected

A series of desired bit sizes for the generated random prime in the range from $2^8$ to $2^{2048}$ was selected to span from small to useful values of large prime numbers. The times to generate those prime numbers were collected and plotted. The nature of the resulting curve may give an indication of the likely times needed to generate numbers outside the selected range. Fitting the data points to an appropriate mathematically derived curve could be used to predict estimated times within some degree of accuracy.

<table>
<thead>
<tr>
<th>Prime Size in Bits</th>
<th>Iterations</th>
<th>Total Time</th>
<th>Average Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>1000</td>
<td>1</td>
<td>0.001</td>
</tr>
<tr>
<td>16</td>
<td>1000</td>
<td>2</td>
<td>0.002</td>
</tr>
<tr>
<td>32</td>
<td>100</td>
<td>1</td>
<td>0.010</td>
</tr>
<tr>
<td>64</td>
<td>20</td>
<td>1</td>
<td>0.050</td>
</tr>
<tr>
<td>128</td>
<td>10</td>
<td>1</td>
<td>0.100</td>
</tr>
<tr>
<td>256</td>
<td>1</td>
<td>1</td>
<td>1.000</td>
</tr>
<tr>
<td>512</td>
<td>1</td>
<td>13</td>
<td>13.000</td>
</tr>
<tr>
<td>768</td>
<td>1</td>
<td>94</td>
<td>94.000</td>
</tr>
<tr>
<td>1024</td>
<td>1</td>
<td>168</td>
<td>168.000</td>
</tr>
<tr>
<td>1792</td>
<td>1</td>
<td>754</td>
<td>754.000</td>
</tr>
<tr>
<td>2048</td>
<td>1</td>
<td>1218</td>
<td>1218.000</td>
</tr>
</tbody>
</table>

Graph of Results
5.2 Analysis of Data

A rule-of-thumb for determining viable run times would be that the time should not exceed human-scale patience (e.g. 2 minutes). Real-time applications would require generation in less than a few seconds, preferably in the sub-second range. Longer times would be viable for longer term static certificate usage, however, if generation times took on the order of hours, then the cost to meet demand could inhibit this technology and other methods might prove more economical.

5.3 Degree of Agreement With Expected Results

The results up to 512 bits were about what we expected. However, for larger prime numbers, the results took much longer than expected. We believe that this result was due to the increasing number of primes that had to be proved. It should be noted that we kept a number of display statements in the program to monitor results and see that it was still working properly. Also, there could have been some dependency on the number of other students doing work on the shared Unix platform. Finally, the speed of the program depended on the “luck” in selecting random numbers. Sometimes the program stalled, when an bad combination of numbers was produced. Sometimes the results came quicker than expected, but the numbers in the table above were representative of most runs of the program.

5.4 Practical Application of Results

We found that the program in its current form was not very stable and occasionally stalled at certain calculations. This appears to be due to the sensitivity to the selection of the ranges used at certain points in the program. Additional more complex controls of those ranges based on the ever changing size of the desired prime, depending on the current level of recursion would need to be devised. Maurer recognized this and included some suggestions, but there was insufficient time to try them.

If the time to generate primes in the size of 2048 bits and larger continues to be on the order of tens of minutes, then we do not see this method as better than the selection of primes using a probability test. In fact, we used a Miller-Rabin test to verify the primes and that test generally took only a second or two to complete. With sufficient testing to produce an extremely high probability of being prime, such numbers are likely to be more uniformly distributed.

6 Conclusions

This project provided an interesting exercise in understanding the difficulty of constructing large random prime numbers. It also provided insights into the limitations of some computers and programming environments as well as some of the techniques used to overcome those limitations. Given some additional time, we would recommend that future students implement their own routines for performing basic numerical calculations, or at least use a small set of routines that are fully exposed and not hidden within a large complex library.
7 References


[2] NTL: A Library for doing Number Theory (version 5.0c) found at: http://www.shoup.net/ntl/
Appendix A: Original Source Code