A study of Maurer's algorithm for finding provable primes in relation to the Miller-Rabin algorithm

Mathias Romme Schwarz (20041105) and Søren Grønnegaard Andersen (20043087)

December 17, 2007

Contents

List of Figures

1 Introduction

Many crypto systems require the use of very large primes. During the lectures, we have seen a variety of systems of which RSA[1] is the most prominent, with El Gamal as another important player. We have seen the Miller-Rabin algorithm for generating primes that are extremely likely to be prime numbers but are not guaranteed to be so. In the following report, we will refer to numbers generated by such algorithms as "probable primes". We will concentrate on algorithms that do guarantee to return a prime and thus we will refer to numbers generated this way as "provable primes".

In this report we will describe and implement both the Miller-Rabin and the Maurer Fast Prime algorithms as described in [6] and [5]. Furthermore, we will discuss and implement various speedups for both algorithms, and evaluate their effectiveness. We will mainly focus on Maurer's algorithm, and we will prove several theorems necessary to proving the correctness of Maurer's algorithm. Lastly, we will try to argue for a theoretical running time of each algorithm, and use our implementations for a head-to-head comparison between Maurer and Miller-Rabin. We will also use Trial Division to experimentally determine the optimal value of some constants used in both Miller-Rabin and Maurer.

The language of choice for implementing these algorithms is Java and as the sources are a bit too wordy for this report, they can be obtained from our website `http://www.daimi.au.dk/~schwarz/crypto/` instead.

1.1 Motivation for finding provable primes

When we produce a probable prime, we introduce a slight possibility that the number is in fact composite. There can be several serious drawbacks to this. Our crypto systems rely on special properties for primes, so if we are unlucky and pick a composite number, our crypto system may completely fail to work or it may be so seriously weakened that it is easy to break. Even if the possibility of such a failure is extremely small, it is still present, and it may be interesting to look at algorithms for finding provable primes.

The simplest example of such an algorithm is a method where we try to divide the number, n, by all $p \in Primes < n$ to see if any of them divides n. We may improve this by only dividing by all $p \leq \sqrt{n}$. This is of course not feasible for very large n, but as we will see later this is indeed the best choice for small n. This report will later determine some numbers for Miller-Rabin and Maurer versus this simple primality test.

2 The Miller-Rabin Algorithm

The currently most popular algorithm for finding primes for use in various cryptographic schemes, is the Miller-Rabin algorithm. This algorithm is a compositeness test: it guarantees that when it returns "Composite" for some number, that number is in fact composite. There is no such guarantee for primes, however - Miller-Rabin is probabilistic and may claim some integer is prime, even if it is not. Miller-Rabin is very accurate, though. Very strong bounds on its accuracy have been proven: Let $p_{k,t}$ be the probability that, when some odd integer of k bits passes t consecutive rounds of Miller-Rabin with different bases, the integer is not prime. Damgård, Landrock and Pomerance[4] proved that $p_{256,6} \leq 2^{-52}$.[2]. In [4] an equation is provided to calculate a bound for this probability. Depending on how t and k depend on each other, several different inequalities can be used. We will only state the one that requires $3 \leq t \leq k/9$ and $k \geq 21$. This captures most cases of Miller-Rabin tests since we for practical cases use large bit sizes and a small number of rounds. The inequality looks like this:

$$p_{k,t} < k^{3/2} 2^t t^{-1/2} 4^{2-\sqrt{tk}}$$

[1] [7] section 5.3
[2] [5], page 3

Using this we can find that $p_{1024,5} \leq 2^{-97}$. We will use this t and k when testing Miller-Rabin later on. Miller-Rabin is also fast. Our tests suggests that prime of around 500 bits takes less than a second to find.

2.1 The Math of Miller-Rabin

The Miller-Rabin test is built on the following fact from number theory.[3]

Theorem 1. *Let n be an odd prime, and let $n-1 = 2^s r$, where r is odd. Let a be any integer so that $gcd(a,n) = 1$. Then either:*

- $a^r \equiv 1 \bmod n$
- $a^{2^j r} \equiv -1 \bmod n$ *for some* j, $0 \leq j \leq s-1$

We will not prove this fact, merely note that it is true. It is worth noting that this theorem only says these properties hold for primes - it does not say they cannot hold for some composites as well. This fact suggests a way to attempt to check primality for some odd integer n: pick an integer a less than n. Compute $a^r \bmod n$. If this equals one, return "probably prime". If it does not, try computing $a^{2^j r} \bmod n$ for all j's between 0 and $s-1$. If this equals -1, then return "probably prime". If it does not - try another n.
This is, in fact, the essence of the Miller-Rabin algorithm, though Miller-Rabin tests with multiple $a's$, rather than just return "probable prime" once these conditions are met. We call the $a's$ bases, and each test of a base a round. The Miller-Rabin algorithm uses a number of other theoretical results as well to guarantee that numbers are composite if it returns "composite". Since this report focuses on Maurer's algorithm, we will not spend further time on these results.

2.2 Miller-Rabin Algorithm

We implemented Miller-Rabin in Java, and it can be found in the ProbablePrime class. Algorithm 1 is a pseudo code description of the algorithm. Line 4-5 obviously tests the $a^r \equiv 1 \bmod n$, but it is perhaps less

Algorithm 1 Miller-Rabin(int n, int *num_rounds*)

Require: n must be an odd integer $> 2, num\_rounds \in Z_+$
```
   (s, r) = find_sr(n − 1)
   for i = 0 to num_rounds do
3:   a = randInt(2, n − 2)
     y = a^r (mod n)
     if y ≠ 1 and y ≠ n − 1 then
6:     j = 1
       while j ≤ s − 1 and y ≠ n − 1 do
         y = y^2 (mod n)
9:       if y == 1 then
           return false
         j+ = 1
12:    if y ≠ n − 1 then
         return false
   return true
```

clear how Miller-Rabin tests our other equation, $a^{2^j r} \equiv -1 \bmod n$. This is done in the loop starting line 7. Line 8 inside this loop calculates $a^{2^j r} \bmod n$ in each iteration, however rather than check directly if this is -1,

[3] [6], chapter 4, page 138

Miller-Rabin checks if it equals 1. If it is, then n is provably composite.[4] Note that the check for congruence with -1 happens in the $y \neq n-1$ clause of the while-loop. To generate a prime using Miller-Rabin we can now simply pick a random odd integer and test using Miller-Rabin whether it is a prime. If it is not then we pick a new candidate and test this.

3 Trial division

Trial division is a simple form of primality test, in which the odd integer, say n, to check is divided by all primes less than some bound. If any of these primes divide n, it is not prime. If none do, then it might be. Trial division can be used to prove primality as well - simply set the bound to $\sqrt{n}$. This is exceedingly slow for primes larger than 20-25 bits (we will see this on a graph later), however, and thus not attractive in practice for large n. For smaller bounds, trial division can weed out many trivial composites extremely fast, which makes it very attractive as a weeder for a slower, more thorough test. As we will also see trial division can be a building brick in a larger algorithm such as Maurer which we will see later. We will not state an algorithm here for doing trial divisions as it should be obvious to the reader how to implement this. All you need is a list of all primes up to the bound and then you divide n with each of them.

4 Maurer's prime generation algorithm[5]

Most (if not all) efficient algorithms for finding provable primes use Theorem 5 (which we will discuss shortly) to some extent. The most well known method is the Maurer algorithm for finding provable primes. Algorithm 2 outlines how the prime is computed. Note that unlike Algorithm 1 this does not test whether a number is prime - it generates one. A parameter k is given to indicate the size of the desired prime such that $p \sim 2^k$. Before discussing the algorithm in detail, we discuss some number theoretical results needed for proving the correctness of Maurer's algorithm.

4.1 Necessary number theory theorems

To achieve the goal of creating an algorithm that efficiently allows us to generate provable primes we need a little number theory which we will discuss here. First we reiterate a few theorems from [7], as well as a few definitions.

Definition 1. *The order of an element g in some group G is the smallest number m for which $g^m \equiv e$, where e is the groups identity element.*

From this follows that in the groups we will be working with, that is Z_n^*, $g^m \equiv 1 \pmod{n}$.

Theorem 2. *If G is a multiplicative group of order n, and $g \in G$, then the order of g, called m, divides n.*

This is often referred to as "Lagrange's Theorem". We will omit the proof as it is beyond the scope of this report, but we will use this theorem often in the following. Additionally, we will need the following corollary:

Corollary 1. *If $b \in Z_n^*$, then $b^{\phi(n)} \equiv 1 \pmod{n}$.*

Proof. Follows directly from theorem 2, since Z_n^* has the order $\phi(n)$. □

[4] This is treated in slightly more detail in [6], chapter 4, page 139

4.1.1 Lucas' algorithm for primality testing[2]

Back in 1891 Lucas showed how to apply Fermat's little Theorem to create a primality test. The theorem looks as follows:

Theorem 3. *Let $n > 1$. If there for any prime factor q of $n-1$ is an integer a such that*

1. $a^{n-1} \equiv 1 \pmod{n}$
2. $a^{\frac{n-1}{q}} \not\equiv 1 \pmod{n}$

then n is prime.

Proof. [5] In order to show that n is prime, we will show that $\phi(n) = n-1$. Or, to put it differently, that $n-1 \mid \phi(n)$. We will prove this by contradiction. Suppose that $n-1 \nmid \phi(n)$. Then, since $n-1$ is composite (we only test odd n's), it must have a prime factorisation. Since $n-1 \nmid \phi(n)$, there must be some prime, q, lifted to some power, r, that divides $n-1$ but not $\phi(n)$. For this prime q we must have an integer a that satisfies property 1 and 2. Now let m be the order of a modulo n. Then by theorem 2, $m \mid n-1$, and $m \nmid \frac{n-1}{q}$. Since $m \mid n-1$, and $q^r \mid n-1$, $q^r \mid m$. Furthermore since $\phi(n)$ is the order of the group that is formed by mod n, this means that by theorem 2, $m \mid \phi(n)$. But, now $q^r \mid m \mid \phi(n)$, and a direct consequence of our only assumption was that $q^r \nmid \phi(n)$. Thus, n must be prime. □

4.1.2 Pocklington's theorem

This theorem was further strengthened by Pocklington. Theorem 3 requires that the whole number $n-1$ is factorized in order to make it possible to test all prime factors. We can strengthen this by only requiring something of the order $\sqrt{n}$ to be factored. This can be done using Pocklington's theorem:

Theorem 4. *$n-1 = q^k \cdot R$ where q is a prime such that $q \nmid R$. If there is an integer a such that*

- $a^{n-1} \equiv 1 \pmod{n}$
- $gcd(a^{\frac{n-1}{q}} - 1, n) = 1$

then each prime factor p of n has the form $p = q^k r + 1$, where r is some integer.

Proof. [6] Let p be any prime divisor of n. Let a be any integer satisfying $a^{n-1} \equiv 1 \pmod{n}$ and $gcd(a^{\frac{n-1}{q}} - 1, n) = 1$, and let m be the order of $a \pmod{p}$. By theorem 2 we know that $m \mid (p-1)$. Since $p \mid n$, $m \mid (n-1)$, since $n-1$ is the order of the group. But since $gcd(a^{\frac{n-1}{q}} - 1, n) = 1 \Rightarrow a^{\frac{n-1}{q}} \not\equiv 1 \pmod{p}$ for our specific p, it follows that $m \nmid \frac{n-1}{q}$. Since $m \mid n-1$, and $n-1 = q^k r$, then $q^k \mid m$ with some scalar, say s. We can now express m as $m = q^k \cdot s$. But, by definition 1, $m \mid p-1$ with some scalar, say t. So, $p-1 = q^k \cdot s \cdot t \Rightarrow p = q^k \cdot s \cdot t + 1$. But s and t are just some integers, which we might as well call r, so $p = q^k \cdot r + 1$ which proves our theorem. □

4.1.3 Using Theorem 4 as a prime test

We can now apply Theorem 4 and 3 to create our primality test. We now know how to prove primality for very specifically designed numbers. The one weakness is that we limit ourselves greatly in theorem 4. Only composites of the form $q^k \cdot R$ may be proven prime. We now seek to expand that result to the more general case where $n-1 = F \cdot R$, where F and R are some composites, $F > R$ and $gcd(F, R) = 1$. Thus, we no longer require F to be of the form some prime raised to some power.

[5] Proof due to [2]

[6] Proof due to [2], but we have fleshed out the arguments for clarity

Theorem 5. *Let $n-1 = FR$, where $F > R$ (note that this specifically means that $R < \sqrt{n}$), $gcd(F, R) = 1$ and the factorization $p_1, p_2, ..., p_m$ of F is known. If for all p_i there exists $a > 1$ such that:*

- $a^{n-1} \equiv 1 \pmod{n}$
- $gcd(a^{\frac{n-1}{p_i}} - 1, n) = 1$

then n is prime

Proof. Follows from theorem 3 and 4, remembering that theorem 4 required $q \nmid R$, we now simply require this is the case for all prime factors of F. We require $F > R$, since we need to ensure F is approximately the size of $\sqrt{n}$. □

4.2 Discussion of Maurer's algorithm

After having proved these important results, we now return to our description of Maurer's algorithm. The algorithm falls in two parts: one using trial-division, one using theorem 5. Algorithm 2 shows the pseudo code for the Maurer algorithm.

4.2.1 Simple trial division for $k < 20$

For small numbers where $k < 20$, the prime is found by simple trial division as this is faster than using the Maurer method. To prove primality we will of course test for all primes $p \leq \sqrt{n}$ where n is the number we are trying to prove primality for. If we find that n is composite we will pick a new n and start over with this. As we will see later for $k > 20$, algorithm 2 is called recursively with a smaller k, so this trial division has significance no matter what k we input.

4.2.2 Using Theorem 5 for larger k

If $k > 20$ we will do something more tricky. The idea is to generate some prime, n, of k bit, using primes of size at least $\sqrt{n}$. If we construct the F of Theorem 5 then we of course know the factorization. In this way we can use Theorem 5 to determine that the number n is a prime. First (lines 11-12 in algorithm 2) we set two constants c, m that are used to determine a trial division bound that we will use. Next (on lines 13-19) we will find a random r which we call the relative size[7]. To ensure that we find a prime that is chosen uniformly random from the set of primes, r must be as random as possible and essentially be distributed in the same way as the primes. It must hold that $r \geq 0.5$ for Theorem 5 to work (the intuition is that if q has a least half the number of bits of n then it is at least $\sqrt{n}$). To let r have a uniform distribution relative to 2^k we thus pick a value uniformly in the interval $[0, 1]$ and raise 2 to this value. If k is relatively small ($k < 2m$, where m is something we choose) we will just set $r = 0.5$. All in all we will have $r \in [0.5, 1]$. We will then generate a prime q of bit size $r \cdot k$ by calling the function recursively. As $r \geq 0.5$, q will always be at least the square root of the prime that we are generating right now. This is important as it makes Theorem 5 work since we have the factorization of at least the square root of the number given. Thus we generate a number n. We then trial divide n with all numbers up to B to weed out some composites in a cheap way before using Theorem 5 in lines 32-38 which will tell us whether we found a prime by applying Theorem 5.

4.3 Notes on primes generated by Maurer

The above description is not fully what Maurer suggested, however. He proposed 2 algorithms in [5]. The other is a bit more elaborate on choosing r (actually more than one r is chosen) and the primes that come from calling recursively. This has theoretical importance, but is not done in practice since it makes the

[7] The relative size of two integers is defined to be $\frac{log(q)}{log(n)}$

algorithm run much slower with a relatively little gain when it comes to ensuring that the generated primes are distributed uniformly. The idea is that we do not only select one prime factor p that is at least the square root of the prime we are generating. Instead we find multiple p's that when they are multiplied are at least as large as the square root of the number we are generating. The way of choosing the r's used is quite expensive and a little involved but Maurer shows how to do this[8] and he describes an algorithm very similar to Algorithm 2 that uses this. There is no need to state the full algorithm here, just think of it as generating $F = q_1 \cdot q_2 \cdot ... \cdot q_i$ where F is the F used in Theorem 5 where our algorithm generates $F = q$. Furthermore the $fast\_prime$ version will only be able to generate 10%[9] of all existing primes. The slower version will be able to generate more primes, but still not all.

5 Expected running times

Before showing the plots of the running times in practice we will have a look at the theoretical running times.

5.1 Running time for Maurer

The running time of Maurer's algorithm is very complicated to estimate. It involves many estimates based on the implementation of long integer arithmetic, and quite a bit of probability theory. However, the essence is that by implementing exponentiation via multiplication with the inverse modulus, the expected time for one exponentiation is $O(k^3)$, for a straightforward implementation of long integer arithmetic. Furthermore, it can be shown that the expected time for one division operation is $O(k \cdot log(k))$. By various properties, it can then be shown that the running time is $O(\frac{k^4}{log(k)})$ for a straightforward implementation of integer multiplication, and where k is the size of the desired prime in bits.[10]

5.2 Running time for Miller-Rabin

We will try to analyze the running time ourselves, but check with another source for the running time of Miller-Rabin. First, we attempt a bound ourselves. First of all, there are only 2 loops in the algorithm - line 2 and line 7 - plus the $find_{sr}$ method call. $find_{sr}$ is called but once per integer to check. Due to its implementation, its running time is $O(log_2(n-1))$[11]. The first loop obviously runs num_rounds times, the other a max of $s-1$ times. s is found by the $find_{sr}$ algorithm in such a way that $r^{-1} \cdot (n-1) = 2^s$, so s must be roughly the size of $log_2(r^{-1} \cdot (n-1))$. That gives us a running time of $O(num\_rounds \cdot log_2(r^{-1} \cdot (n-1)) + log_2(n-1)) = O(num\_rounds \cdot log_2(r^{-1} \cdot (n-1)))$, if the number we are testing is prime. The time to test a composite would be much less - usually trial division will reject it, if nothing else. However, the chance that some random integer n is prime is about $\frac{1}{log_2(n)}$[3], so we expect to try approximately $log_2(n)$ different numbers before we find one. Thus, the final running time is $O(num\_rounds \cdot log_2(r^{-1} \cdot (n-1)) \cdot log_2(n))$. An outside source[12] says that since Miller-Rabin uses modular exponentiation (line 8), the expected running time is $O(num\_rounds \cdot log^3(n))$. This corresponds very well with our own analysis, since $log_2(r^{-1} \cdot (n-1))$ should be somewhat larger than $log_2(n)$.

[8] See [5] Appendix 1

[9] See [5]

[10] Better performances can apparently be obtained using other implementations of integer multiplications. All this can be found in detail in [5]

[11] See ProbablePrime.java - we simply compute 2^s as long as it is smaller than $n-1$. Rather naïve, but fast.

[12] The best source we could find was [8] but it sounds reasonable in relation to our own results

Algorithm 2 maurer_fast_prime(int k)

Require: $k \in Z_+$
  if $k < 20$ **then**
    repeat
      $n \in [2, 2^k]$
      $isPrime = true$
      for all primes $p \leq \sqrt{n}$ **do**
        if $n \equiv 0 \pmod{p}$ **then**
          $isPrime = false$
    until $isPrime$
    return n
  else
    $c = 0.1$,$m = 20$
    $B = c \cdot k^2$
    if $k > 2m$ **then**
      repeat
        $s \in [0, 1]$
        $r = 2^{s-1}$
      until $k - rk > m$
    else
      $r = 0.5$
    $q =$ maurer_fast_prime($\lfloor r \cdot k \rfloor$)
    $I = \lfloor \frac{2^{k-1}}{2q} \rfloor$
    $succes = false$
    while $\neg success$ **do**
      repeat
        $R \in [I + 1, 2I]$
        $n = 2Rq + 1$
        $mayBePrime = true$
        for all primes $p \leq B$ **do**
          if $n \equiv 0 \pmod{p}$ **then**
            $mayBePrime = false$
      until $mayBePrime$
      $a \in [2, n - 2]$
      $b = a^{n-1}$
      if $b = 1$ **then**
        $b \equiv 2^{2R} \pmod{n}$
        $d = gcd(b - 1, n)$
        if $d = 1$ **then**
          $success = true$
    return n

5.3 Running time for Trial Division

It is easier to estimate the running time of trial division, though not trivial. Let $\pi(x)$ be the number of primes less than some number x. We have a very hard bound on the number of calculations needed if x is indeed prime - x must pass trial division by all primes less that $\sqrt{x}$. Thus the expected time to prove primality would be $\pi(\sqrt{x})$. An expected time to prove compositness is harder to estimate accurately, but it could certainly be no more than $\pi(\sqrt{x})$. (And it would almost certainly be much, much lower than that) Since "chance of composite" = "1-chance of prime", the expected running time of the entire algorithm where you choose a random odd integer and try to prove primality, is no worse that $O(\pi(\sqrt{x}))$. A decent estimate of $\pi(x)$ is $\frac{x}{log(x)}$[13] . Thus, our expected running time is $O(\frac{\sqrt{x}}{log(\sqrt{x})})$. For this running time to make sense, we must assume that all primes less than $\sqrt{n}$ are known in advance. One could simply save all these generated primes for later use, but such a list would quickly become extremely large for any realistic bit size, say 1024 bit, making Trial Division a poor choice in general.

6 Speed improvements

For practical use of the algorithms we discuss here, it is highly relevant to improve the speed of the algorithms as much as possible. Many articles have been written on this subject, and we will study what is being proposed in [1] and [6]. Some of these speed ups are of no practical use and some of them are. There can be many reasons why they fall out of use. One of them could be that an improvement is based on replacing one arithmetic expression with another that is faster to evaluate, but that changes in CPU architecture - or speed - means that there is no longer any gain. Another improvement may only work for small numbers which means that it falls out of relevance as the numbers we use become larger.

6.1 Our test setup

For testing we have used the Daimi camel10 machine, which is a 3Ghz Dual Intel Xeon machine with 1 GB. We do not use a lot of RAM for our programs so the very moderate amount of RAM in camel10 is not an issue here. As can be seen from the plots the first data point in each graph seems to be off the trend of the others. This happens on all the plots and it is due to the Java Hotspot compiler optimizing the program when it has run for a little while. This means that the first point is always showing unoptimized byte code being run and the following points show JIT compiled code being run.

6.2 Speeding up Miller-Rabin

Miller-Rabin is fairly slow when used alone, but it is possible to speed it up greatly using a simple addition to the algorithm. Using trial division will sort out many integers that are obviously not prime. This is a speeding improvement that has been proposed frequently. We made some testing to create a test see when to use Miller-Rabin and when to use trial division. Testing on different bit sizes, k, we find the optimal algorithm to use for k bits by plotting the trial division method and the Miller-Rabin method for finding 2000 primes in the interval $[2^5, 2^{25}]$ since we expect k to be found in this interval. Since we always have to test a significant amount of numbers before finding a prime, it is better to always use the faster one of trial division and Miller-Rabin. We plotted the running times in and it can be seen in Figure 1 on page 11. It is clear from this figure that $k = 19$ so our combined algorithm should use trial division to try to find a factor $q < \sqrt{2^{19}}$ before resorting to the Miller-Rabin test.

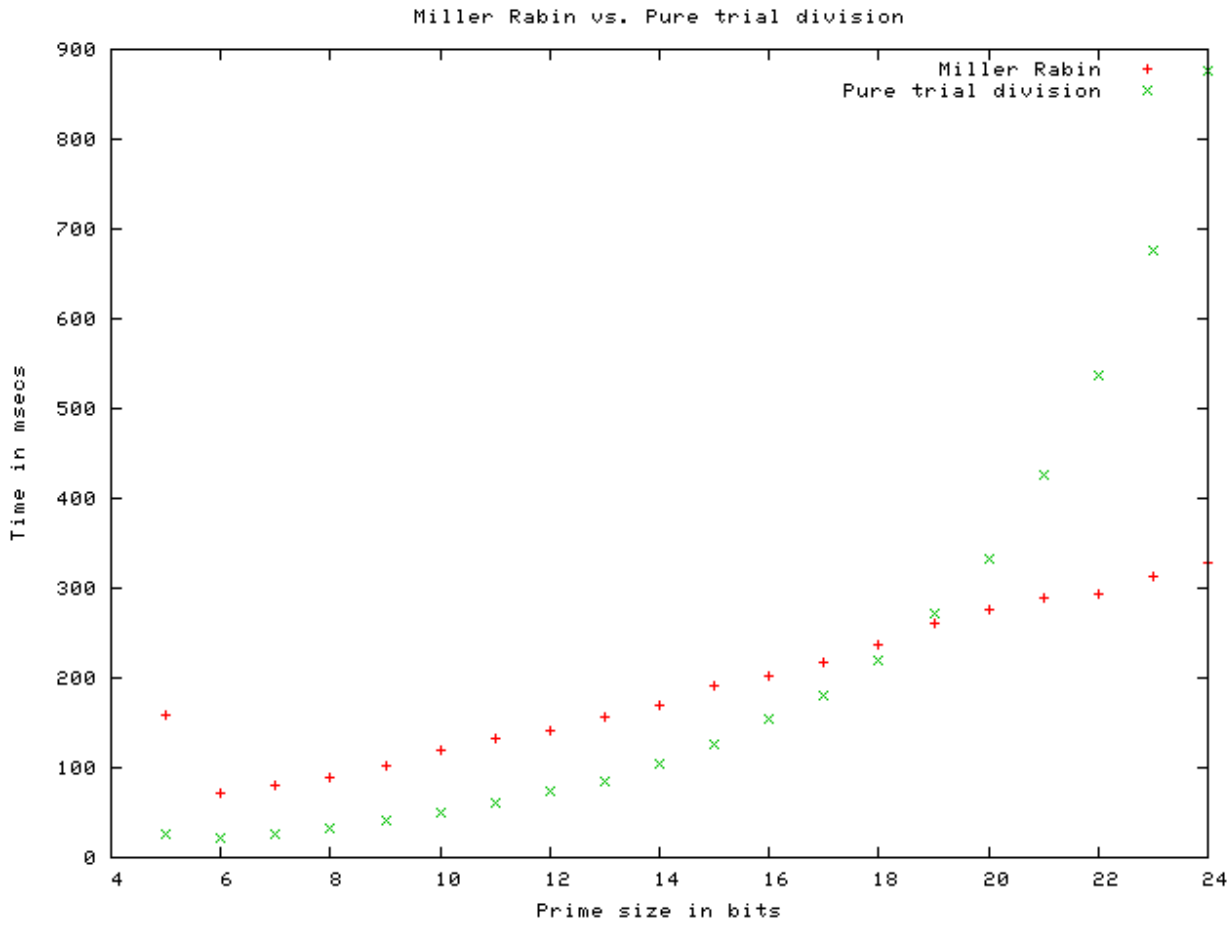

Figure 1: Plot comparing running times of Miller-Rabin to running times of trial division. It shows the times for 2000 primes of the given bit size to be found using the two methods

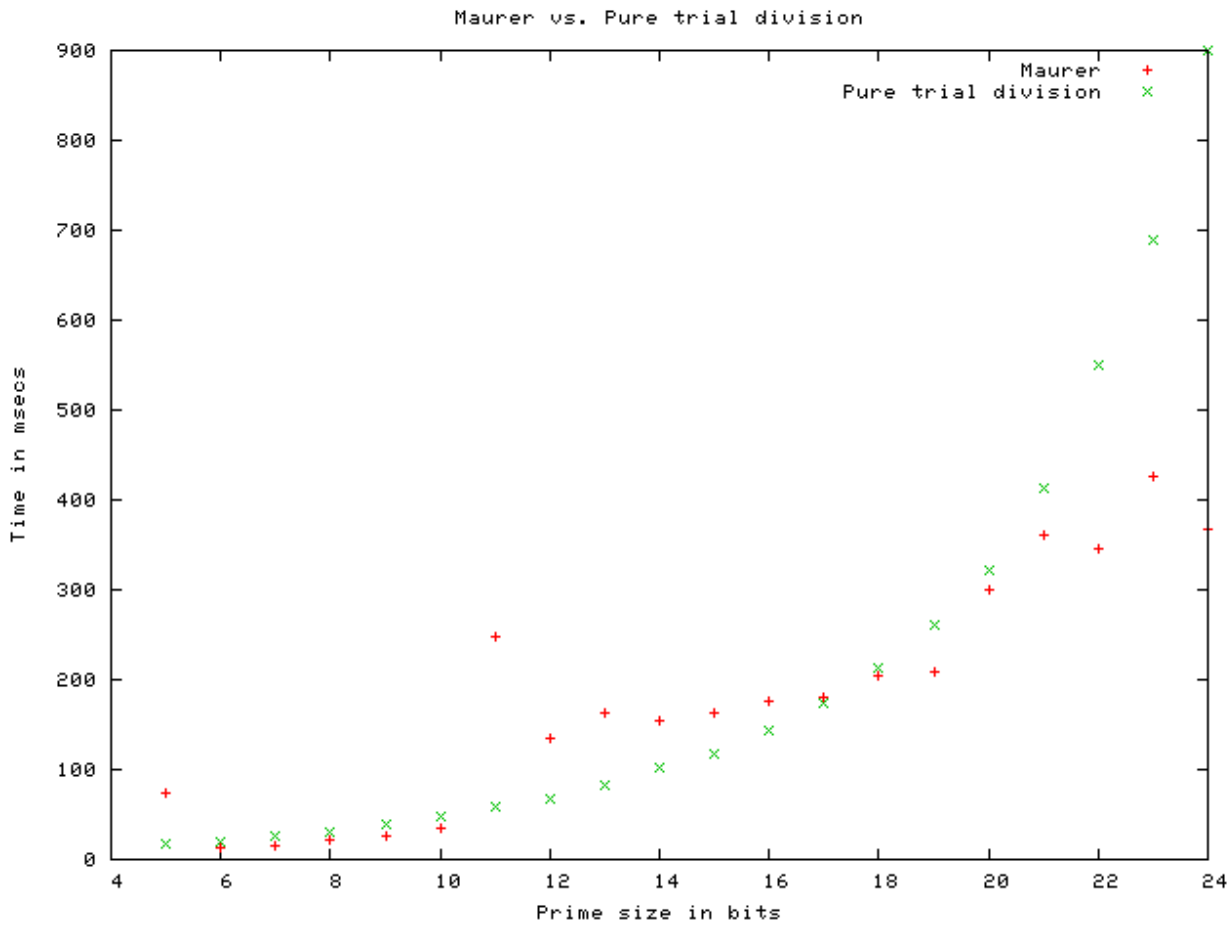

Figure 2: Plot comparing Maurer with the Miller-Rabin improvement to pure trial division. It shows the time for finding 4 primes of the given bit size. Maurer has to use some trial division so the bound is set as low as possible (10)

6.3 Speeding up Maurer

There is a number of things one can do to improve the running time of Maurer's algorithm - we have implemented some of them, and studied others. After calculating $n = 2 \cdot R \cdot q + 1$, we do trial division on it, up to its bound ($c \cdot k^2$). We did some practical testing on our code and we found that it can save several hundred iterations of the main loop, and thus several hundred multiply and gcd operations. Obviously, the bound cannot be too large since there is a point where trial division is slower than the original algorithm. Also, Maurer always finds the small primes needed using trial division. This means that all primes smaller than some bit size k are not constructed by building them from smaller primes but are found using trial division. Maurer[5] suggests that this k should be 19 and as can be seen on Figure 2 (page 12) this seems to be the right choice. It is harder to compare Maurer to trial division than doing the same comparison for Miller-Rabin since Maurer needs trial division to work. We have set the trial division bound as low as possible (this number seem to be 10, for smaller k the algorithm loops) to be able to do some comparisons. 19 also sounds reasonable since the trial division bound we found for Miller-Rabin was 19. Maurer should be slightly slower than Miller-Rabin but apparently not slow enough to justify a higher bound.

6.3.1 Speeding up Maurer using Miller-Rabin

We decided to look at a speedup of Maurer which consists of a single Miller-Rabin test with 2 as the base, as suggested by [1] and [6]. Miller-Rabin is slightly faster than Maurer and the idea is that for base 2 all multiply operations (expensive) can be replaced by bit shifting operations (cheap), thus making Miller-Rabin much faster than Maurer. We will then only continue with the Maurer algorithm if the number passes this simplified Miller-Rabin test. This requires us to replace all multiplications with bit shifting in our Miller-Rabin and this requires us to override the multiplication method on BigInteger. Before doing this we decided to test the BigInteger class to see if perhaps Java's class was already optimized for a base of 2. We performed 100000 single bit shifts of random 2048 bit numbers, as well as 100000 times picking a random 2048 bit number and multiplying by 2. As a baseline, we also performed 100000 multiplications between a random 20 bit number $\neq 2$ and random 2048 bit numbers. Our results were:

```
2 multiplied by random 2048 bit number: 1243 milliseconds
Random 2048 bit number bitshifted     :  994 milliseconds
20 bit random times 2048 bit random   : 1201 milliseconds
```

As is evident, bit shifting is in fact slightly faster. However, the amount of performance gained is in all respects negligible. We have done some testing and we estimate that we would probably perform approximately 2-5000 multiplications per prime found. The multiplications happen as a part of the *modPow* method which finds a given b, c, and n in calculations on the form $a = b^c \pmod n$. It is of course not feasible first to take b^c and the calculate a since b^c can easily be a larger number than we are able to store in the memory of a computer, so an algorithm is used that makes a substantial number of multiplications. With a difference of 250 milliseconds between bit shifting and multiplication for 100000 operations, that translates to a performance gain between 5 and 12,5 milliseconds per prime found, which is in the category of micro optimizations. This makes the bit shifting optimization irrelevant in practice. This is not the conclusion of [1], but one of the reasons for this may be that the full Maurer is considered there, and we only consider the faster version, algorithm 2.

6.3.2 Speeding up by only knowing the factorization of $\sqrt[3]{n}$

We can sacrifice some of the simplicity in the computations in Maurer to gain a speed up where we only have to know the factorization of $\sqrt[3]{n}$ where normally we would need to know the factorization of $\sqrt{n}$. We need this little theorem for the speed up:

[13] Hadamand, 1896 - though we looked it up in [3]

Theorem 6. *Let $n = FR + 1$ satisfy the conditions of Theorem 5, let R' be the odd part of R, and let $F' = \frac{n-1}{R'}$. Furthermore r, s are defined by $R' = 2F's + r$, where $1 \leq r < 2F'$. Suppose $F' > \sqrt[3]{n}$, then n is prime if and only if $s = 0$ or $r^2 - 4s$ is not a square.*

We will not prove this, but it can be used to generate primes where we only now the factorization of the cubic root as noted above. As can be seen this does not come without a drawback. We have to find this s and r and that requires some computations. We tried to implement this, but there is a bug in our implementation that sometimes makes the computation loop indefinitely. It does work in the sense that when it does return, it returns a prime (we test this using Miller-Rabin), but it is an order of magnitude slower than the standard Maurer we have implemented. We did not make a graph for this since it is not a fair comparison to measure a buggy implementation to a working one.

7 Experimental comparison of the improved algorithms

Now we have optimized each of the algorithms, it becomes interesting to compare the two and see how well they perform compared to each other.

7.1 How we tested

We let a computer do a lot of Miller-Rabin and Maurer calculations and we plotted the running times for these runs. For each algorithm we have plotted times for computations from numbers of 5 bits up to numbers of 1024 bits. It is of course possible to plot larger numbers and our program that generates these graphs take as input the desired maximum bit length. However, it takes a lot of time and the trend stays the same. Since there is a great deal of randomness involved we may be lucky to pick a prime as our test candidate the first time we try, but we may also pick a lot of composites and hence spend a lot of time. To get rid of some of this "luck noise" from the charts we generate 5 primes for each bit length for each algorithm and plot the sum of these times. For Miller-Rabin, we used a security parameter of 5 - that is, for a number to pass the test and be considered a probably prime, it would have to pass 5 rounds of testing with different bases. Using a higher security parameter could slow the algorithm slightly but almost all composite number are detected with security parameter 5, and therefore only few cases would yield the computation of another Miller-Rabin round and it would have little impact on the running time. The results would therefore be roughly the same if we had set the security parameter to 6 or 7, since the probability of any non-prime lasting more than a few rounds is extremely small.

7.2 Our implementation

Our implementation can be found on the web site as a zip of our project directory. It depends on the Apache logging framework Log4j and the Apache POI framework (specifically HSSF) so these should be present on compile time and on runtime. There is an ant file to build the system and everything can be compiled using 'ant compile'. Everything should be set up so that when the zip file is extracted, it is possible to build the system right away. In the 'report' folder there is a Makefile that will run all the tests needed to generate the graphs used in the report. Be aware that it takes several hours to generate these graphs. It can be done by typing 'make graphs'. This outputs the data points in csv format, Excel (.xls) format and the gnuplot format and png's are built from the gnuplot files. The command 'make' just builds the report itself with the constructed graphs.

7.2.1 The Java classes (/src)

The Java files implement the Miller-Rabin algorithm, the Maurer algorithm and methods for trial division. The Miller-Rabin implementation is located in the class called ProbablePrime and the Maurer is implemented

in the ProvablePrime class. We use the Java BigInteger class which makes the syntax notoriously hideous, but allows numbers of arbitrary size. To make a specific plot there is a Plot interface that must be implemented. Each of the plots have such a Plot class. The PlotOut class ties it all together by constructing a list of Plot object from the input that the user gives and runs all the plots. Output is handled by classes implementing the PlotStrategy interface. We have implemented three different PlotStrategy classes, namely for Excel, csv and gnuplot and they are all enabled by default. Finally there is a PrimeList class which is an infinite list of all primes that can be iterated as any other list. It will precompute a list of all primes < 1000000 if it cannot find a file on the disk containing these primes, so the first startup will take some time. Additional primes needed (for trial division) are calculated on the fly. The Improvement interface is implemented by classes that contains code snippets that can be run as a part of one of the algorithms, for example the TrialDivisionImprovement that can be put into Miller-Rabin. Finally the Condition interface can be implemented to give a condition of when trial division should stop. We have two Condition implementations, one that stops when the square root of the number is reached and one that stops when a specific bound is reached.

7.3 Plot of Miller-Rabin with trial division vs. pure Miller-Rabin

Figure 3 shows the running times of Miller-Rabin with and without trial division as a simple scatter plot. The graph, though not surprising, is interesting in more ways than one. First of all, it is not until we try to find primes of about 300-bits in size or more, that the two algorithms clearly differ in running time. And, it is not until close to 500 bits in size that the change becomes truly noticeable - here the pure Miller-Rabin would take around a second per prime, vs. less than a fifth of that with trial division improvement[14]. But the difference is very noticeable in realistic bit-sizes, making the trial division improvement mandatory - especially since it is so simple[15].

7.4 Plot of Miller-Rabin with trial division vs. Maurer

After having identified what worked and what did not work when trying to improve running times for the two algorithms, it becomes interesting to try to compare the two in terms of running time. There was pros and cons for both algorithms, Miller-Rabin could hand us a composite number instead of a prime and the $fast\_prime$ method of Maurer may not pick all primes with equal probability since we sacrificed this for some speed improvements in an earlier section. Figure 4 shows the two plots of the running times of both algorithms. We notice that there is a clear trend that Miller-Rabin is faster than Maurer. That is no surprise since that was what we expected to begin with. It also shows that the difference is not very big and it shows that many of the slow runs of Miller-Rabin (since it is probabilistic there will always be such slow runs) are indeed slower that the fastest runs of Maurer. That a lucky Maurer run is faster than an unlucky Miller-Rabin run is interesting since it means that we might just as well use Maurer $fast\_prime$ as Miller-Rabin.

8 Conclusions and closing remarks

We have studied different methods of generating primes, and looked into ways of speeding these methods up, and studied how well they perform. Of course trial division is not feasible for very large numbers, but we have seen that it works well for quite large numbers and is actually the best method for numbers smaller than $\sim 2^{19}$. On the question of probable versus provable primes a number of factors should be considered. First of all, though provable primes are of course nice to have, there are factors which make the Maurer algorithm

[14] As evident from the graph - remember, the time plotted is for 5 primes

[15] Note that all trial divisions were made with precomputed primes. This is feasible since one simply selects some upper-bound for testing, as we did

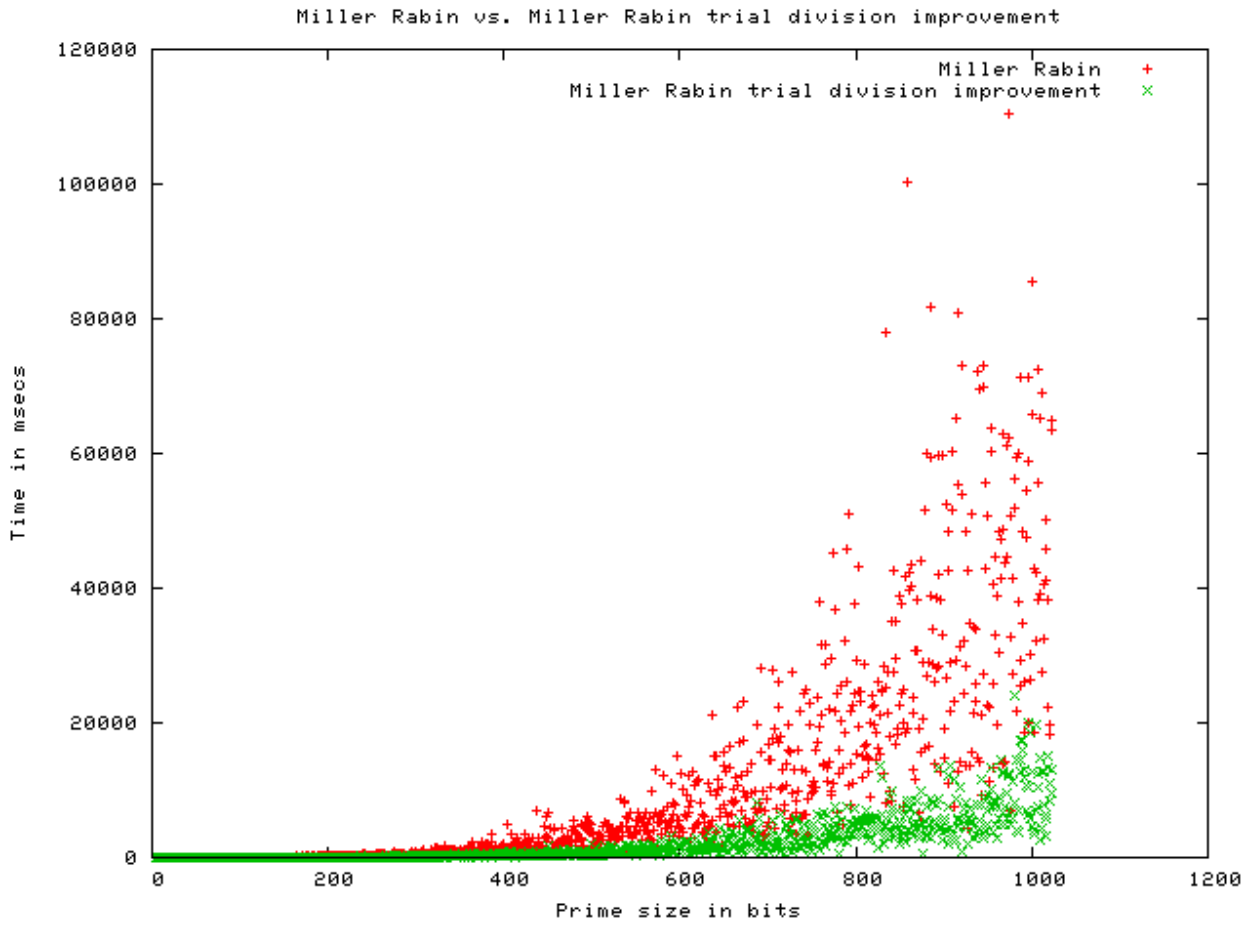

Figure 3: Plot comparing running times of Miller-Rabin to those of Miller-Rabin using the trial division improvement. It shows the running times for finding 4 primes of each bit size

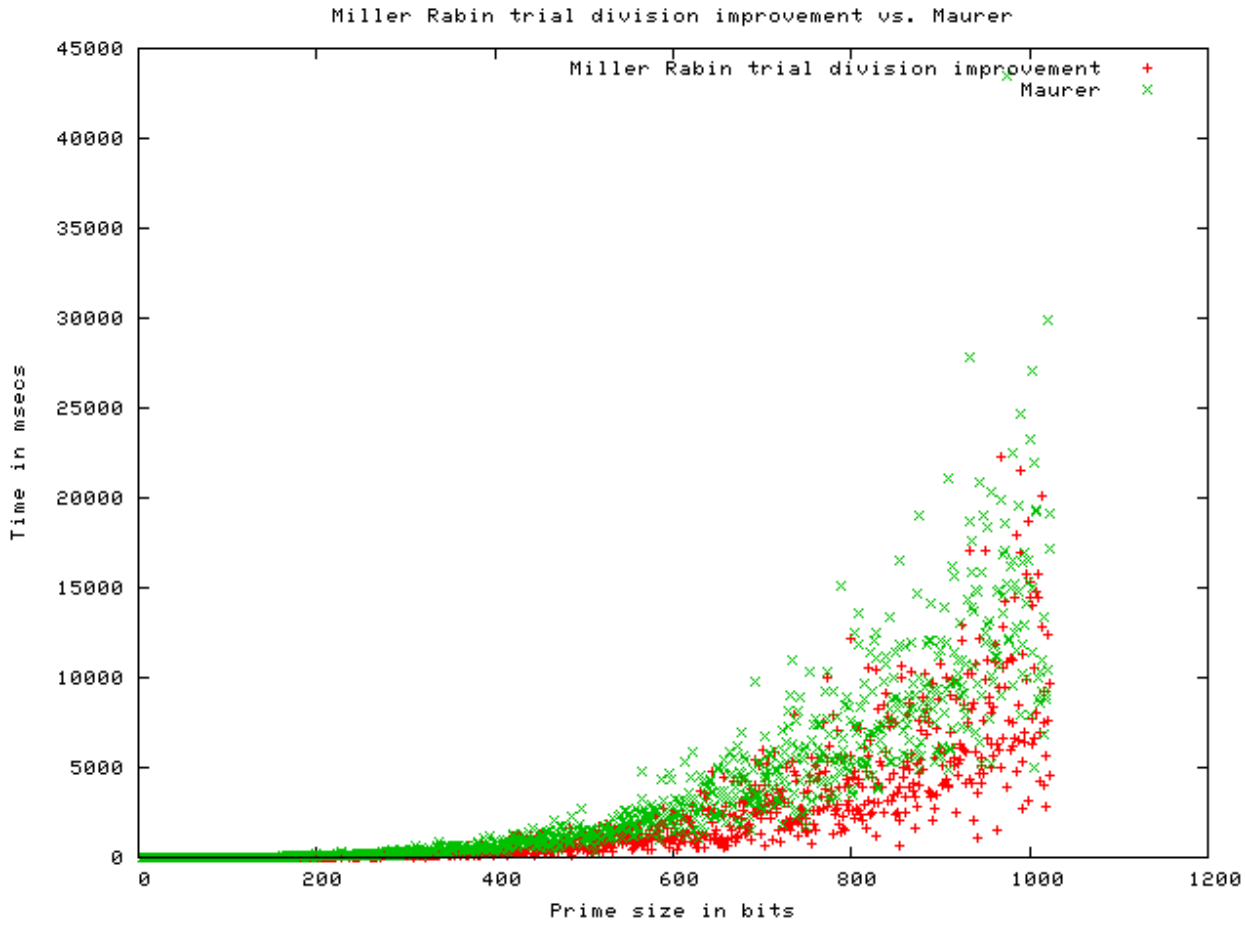

Figure 4: Plot comparing running times of Maurer to those of Miller-Rabin with the trial division improvement

less than perfect. First of all, the full algorithm is complex to implement, understand, and expand. This full algorithm that guarantees that the primes are evenly distributes is also a lot slower. The fast version we worked on is less complex, but can only find about 10% of the possible primes[5]. This is of course a problem in theory, but whether this would be problem in practice is hard to guess. It is feasible, however, that such a limit on the primes we can generate could be exploited by some adversary. Miller-Rabin, on the other hand, is much simpler both to understand and implement - as well as being roughly 20%-30% faster than Maurer around 1024 bits[16]. Regarding speed, it is important to remember than neither Miller-Rabin nor Maurer are by any definition slow. Both are very useful in practice, in terms of speed - Miller-Rabin takes about 4 seconds to generate a 1024 bit prime, and Maurer takes 5 seconds. What's perhaps more damaging to Maurer is the flexibility of Miller-Rabin - since we choose a number at random and test whether it is a prime the Miller-Rabin method can generate any prime. We can even guarantee the distribution to be uniform by simply using a random function that is uniform. Furthermore, as shown by I. Damgård, P. Landrock, and Carl Pomerance[4] the risk of the Miller-Rabin method returning a composite when it returned "probable prime" is extremely small. Many other things with equal probability can break our crypto system. It could be the case that our enemy simply guesses our encryption key without any significant knowledge. On the other hand we have seen various improvements to both algorithm and we have seen that Maurer that guarantees to return primes is only slightly slower than Miller-Rabin. However, at the end of the day what matters is the security and when weighting the fact that Maurer $fast\_prime$ can only generate around 10%[17] of all primes against the extremely small probability that Miller-Rabin will fail seems to put the advantage at Miller-Rabin and that is probably why Miller-Rabin is the most popular algorithm.

[16] See figure 4

[17] As state elsewhere the slow Maurer will be able to reach more primes, but still nor all

References

[1] Jørgen Brandt, Ivan Damgaard, and Peter Landrock. Speeding up prime number generation. In *ASIACRYPT '91: Proceedings of the International Conference on the Theory and Applications of Cryptology*, pages 440–449, London, UK, 1993. Springer-Verlag.

[2] Chris Caldwell. Finding primes & proving primality. http://primes.utm.edu/prove/prove3_1.html, 2007?

[3] Chris K. Caldwell. How many primes are there?, 2007. [Online; accessed 15-December-2007].

[4] Ivan Damgård, Peter Landrock, and Carl Pomerance. Average case error estimates for the strong probable prime test. *Mathematics of Computation*, 61(203):177–194, July 1993.

[5] Ueli M. Maurer. Fast generation of prime numbers and secure public-key cryptographic parameters. *Journal of Cryptology: the journal of the International Association for Cryptologic Research*, 8(3):123–155, Summer 1995.

[6] A. Menezes, P van Oorschot, and S. Vanstone. Handbook of applied cryptography.

[7] Douglas R. Stinson. *Cryptography - Theory and Practice*. Chapman & Hall/CRC, third edition, 2006.

[8] Wikipedia. Miller-rabin primality test — wikipedia, the free encyclopedia, 2007. [Online; accessed 13-December-2007].