Primality Testing Report

CS6150 December 9th, 2011

Steven Lyde, Ryan Lindeman, Phillip Mates

1 Problem Summary & Motivation

Primes are used pervasively throughout modern society. E-commerce, privacy, and psuedorandom number generators all make heavy use of primes. There currently exist many different primality tests, each with their distinct advantages and disadvantages. These tests fall into two general categories, probabilistic and deterministic. Probabilistic algorithms determine with a certain probability that a number is prime, while deterministic algorithms prove whether a number is prime or not. Probabilistic algorithms are fast but contain error. Deterministic algorithms are much slower but produce proofs of correctness with no potential for error. Furthermore, some deterministic algorithms don't have proven upper bounds on their run time complexities, relying on heuristics.

How can we combine these algorithms to create a fast, deterministic primality proving algorithm? Is there room for improvement of existing algorithms by hybridridizing them? This is an especially interesting question due to the fact that AKS is relatively new and unexplored. These are the questions we will try to answer in this report.

2 Previous Work

For this project we focused on three algorithms, Miller-Rabin, Elliptic Curve Primality Proving (ECPP), and AKS. What is the state of the art for these different implementations?

2.1 ECPP

Originally a purely theoretical construct, elliptic curves have recently proved themselves useful in many computational applications. Their elegance and power provide considerable leverage in primality proving and factorization studies [6]. From our reference [6], the following describes Elliptic curve fundamentals.

Consider the following two or three variable homogeneous equations of degree-3:

$$ax^3 + bx^2y + cxy^2 + dy^3 + ex^2 + fxy + gy^2 + hx + iy + j = 0$$
$$ax^3 + bx^2y + cxy^2 + dy^3 + ex^2z + fxyz + gy^2z + hxz^2 + iyz^2 + jz^3 = 0$$

For these to be degree-3 polynomials, we must ensure at least a, b, c, d, or j is nonzero. Let us also assume that the polynomial is absolutely irreducible for the algebraic closure of F. An interesting property of the three variable equation is that the projective solutions are almost exactly the same as the affine solutions of the two variable equation. In particular, a solution (x, y) of the two variable equation is the same as the solution $(x, y, 1)$ of the three variable equation. Also, any solution (x, y, z) with $z \neq 0$ may be identified with the solution $(x/z, y/z)$ of the two variable solution. Furthermore, solutions (x, y, z) with $z = 0$ do not correspond to any affine solutions, and are called the "points at infinity" for the equation. This fact enables us to focus entirely on two variable equations. But the two variable equation above is cumbersome, so it is profitable to consider a change in variables that sends solutions with coordinates in F to like solutions, and vice versa for the inverse transformation.

For example, consider the Fermat equation with an exponent of 3, that is, $x^3 + y^3 = z^3$. Assuming we are considering solutions in a field F with characterstic not equal to 2 or 3. Letting $X = 12z$, $Y = 36(x - y)$, $Z = x + y$, we have the equivalent equation: $Y^2Z = X^3 - 432Z^3$. The projective curve is considered to be "nonsingular" (or "smooth") over the field F if there is no point on the curve where all three partial deriviatives vanish. In fact, if the characteristic of F is not equal to 2 or 3, any nonsingular projective equation with at least one solution may be transformed by a change of variables to the standard form: $y^2 = x^3 + ax + b$, $a, b \in F$, where the one given solution of the original equation is sent to (0,1). Further, it is clear that a curve given above has just this one point at infinity, (0,1). Such a form for a cubic curve is called a Weierstrass form. If we have this type of curve and the characterstic of F is not 2 or 3, then the curve is nonsingular if and only if $4a^3 + 27b^2$ is not 0. We use this fact throughout the various ECPP implementations to validate the random curve values of a, b chosen. The tremendous power of elliptic curves becomes available when we define a certain group operation, under which $E(F)$ becomes, in fact, an abelian group.

1. $-O = O$;
2. $-P_1 = (x_1, -y_1)$;
3. $O + P_1 = P_1$;
4. if $P_2 = -P_1$, then $P_1 + P_2 = O$;
5. if $P_2 \neq -P_1$, then $P_1 + P_2 = (x_3, y_3)$, with $x_3 = m^2 - C - x_1 - x_2$, and $-y_3 = m(x_3 - x_1) + y_1$, where the slope m is defined by $m = \frac{y_2 - y_1}{x_2 - x_1}$ if $x_2 \neq x_1$ or $m = \frac{3x_1^2 + 2Cx_1 + A}{2y_1}$ if $x_2 = x_1$.

Using these group operations we can perform addition/subtraction and multiplication operations. The interesting geometrical properties of these operations allows us to detect when a given point on the elliptical curve proves that n is prime if q can be proven prime because when an illegal curve operation is encountered, it is exploited to find a factor of n. This idea of what we might call "pseudocurves" is the starting point of H. Lenstra's elliptic curve method (ECM) for factorization. With this background, the details of three ECPP implementations are now explained.

Currently there are at least 3 ECPP implementations with varying runtimes:

1. **Goldwasser-Killian primality test** - developed with the intention of providing a polynomial-time complexity for proving most if not all prime numbers with an expected number of operations fixed at $\Theta(\log^k n)$ for an absolute constant k. The idea was to find curves with orders that have large enough "probably prime" factors and recurse on the notion that these factors should in turn be provably prime. In practice, the algorithm iterates through these factors getting a chain of inferences with the last number q so small it can be proved prime by trial division. If some intermediate q is composite, then one can retreat one level in the chain and apply the algorithm again. This not only provides a rigorous primality test but also generates a certificate of primality which can be provided alongside the the original n. Checking the certificate takes significantly less time than generating the original certificate and can be used by anyone who wishes to verify that n is prime [6].

2. **Atkin-Morain primality test** - developed to overcome the point-counting step in the Goldwasser-Kilian ECPP algorithm. Their idea was to find either "closed-form" curve orders, or at least be able to specify the curve orders relatively quickly. Their eventual approach was to find curves using complex multiplcation. The key difference is that they search for appropriate curve orders *first*, and then construct the curresponding elliptic curve. The algorithm does have one flaw, there is no guarantee that it will successfully find u and v even if they exist. When this happens the algorithm will continue to the next discriminant D until it is successful or times out at some maximum discriminant D or iteration counter. Compared to the Goldwasser-Kilian approach, the complexity of the Atkin-Morain method has been heuristically estimated to run with $\Theta(\log^{5+\epsilon} N)$ operations to prove N prime. The added difficulty in getting an exact running time comes from the fact that the potential curve orders that one tries to factor have an unknown distribution [6].

3. **Morain fastECPP primailty test** - developed by Morain to improve the original Atkin-Morain primality test based on an asymptotic improvement due to J. Shallit that yields a bit-complexity heuristic of $\Theta(\log^{4+\epsilon} N)$ to prove N prime. The basic idea is to build a base of small squareroots, and build discriminants from this base. In this way, the choosing of discriminant D can be enhanced, and the overall operation complexity of the original Atkin-Morain algorithm can be reduced from 5 to 4 [6].

The Goldwasser-Killian algorithm consists of the following 5 steps which are ([6]):

1. Choose a pseudocurve over Z_n
2. Assess curve order
3. Attempt to factor
4. Choose point on $E_{a,b}(Z_n)$
5. Operate on point

The first step consists of choosing a pseudocurve at random by picking random values for a, b that satisfy the $4a^3 + 27b^2$ property mentioned earlier. The second step involves calculating the integer m that would be the number of curves of the order Z_n if n is prime. If this fails, then n is composite. The third step is to attempt to factor $m = kq$ where $k > 1$ and q is a probably prime exceeding $(n^{1/4} + 1)^2$, but if this cannot be done in some timely fashion we fall back to the first step and start over. The forth and fifth steps involve picking random points to perform operations to find a point that results in an "illegal" operation performed or the infinite point (0,1) being returned. If an "illegal" operation is discovered it proves that n is composite, otherwise one additional operation is performed to conclusively prove that n is prime if q can be proven to be prime. The algorithm then sets $n = q$ and repeats the same steps until such time that q is small enough to prove it is prime using some brute force method like a seive test up to $\sqrt{q}$.

The primary difference between the Goldwasser-Killian algorithm and the Atkin-Morain implementation is largely the order of the steps used. The main issue is that the algorithm is noticeably sluggish due to the assessing curve order step which involves point-counting to obtain the value of m. Atkin found an elegant solution to this impasse by changing the technique by which the curves are found described next.

The Atkin-Morain algorithm also attempts to find a curve of the order Z_n but does so relatively quickly compared to the Goldwasser-Killian algorithm. The Atkin-Morain algorithm consists of 5 steps which are ([6]):

1. Choose discriminant
2. Factor Orders
3. Obtain curve paramaters
4. Choose point on $E_{a,b}(Z_n)$
5. Operate on point

The key difference is that the Atkin-Morain algorithm finds the appropriate curve orders *first*, and only then constructs the corresponding elliptic curve. The key to doing this is the first step which consists of finding a suitable discriminant D. As it turns out, each discriminant has multiple corresponding curve orders m that are equivalent to the assessing curves step used in the Goldwasser-Killian algorithm. It is fairly easy to find a discriminant D because it will always yield a $Jacobi(\frac{D}{n}) = 1$ if the discriminant is not suitable for producing a curve order m for n. But, it is not fool proof because of a $Jacobi(\frac{D}{n}) = 0$ may or may not yield a curve order m that works for n. But it is significantly faster than the assess curves step used in the Goldwasser-Killian algorithm. Once a discriminant is found, step 2 involves choosing from among the several related curve order m's available to find one that we can factor to find a suitable q similar to the attempt to factor step used in the Goldwasser-Killian algorithm. After spending too much time in this step both algorithms will revert back to the first step and start over. It is also important to note that if n is composite, no suitable curve might be found. It is therefore, highly advisable that Miller-Rabin be used to determine if n is composite before proceeding with the first step and therefore save time in proving n is prime. The third step in the Atkin-Morain method involves choosing suitable parameters for a, and b. There might be several appropriate parameters for a, and b so this step might loop several times trying different suitable parameters for a and b. The forth step involves randomly choosing points to test on the curve to again find either "illegal" operations or the infinity point (0,1) exactly as the Goldwasser-Killian algorithm did. Once a suitable point is found (and several points may be tried in the process) it can be shown that n is prime if q can be proven prime. The algorithm then proceeds to set $n = q$ and starts over with step one until q is sufficiently small to conclusively prove that q is prime using some brute force technique like a sieve test mentioned above. Finally, Morain has continued to find improvements to the algorithm, authoring the "fastECPP" technique detailed below.

Because of the success of the ECPP algorithm in proving some spectacularly large numbers (over 15000 decimal digits in size in one case) there has been a large amount of research into improving the ECPP algorithm. The fastECPP algorithm developed in July 2004 by Morain is based on an asymptotic improvement due to J. Shallit that reduces the running time from $\Theta(\log^{5+\epsilon} N)$ to $\Theta(\log^{4+\epsilon} N)$. The basic idea is to build a base of small squareroots and build discriminants from this basis. The Atkin-Morain method might require $\Theta(\log^2 N)$ discriminants D to be tried before a finding a suitable D. Instead, one may build discriminants of the form $-D = (-p)(q)$, where p, q are primes each taken from a pool of size only $\Theta(\log N)$ instead. This greatly improves the choose discriminant step used in the Atkin-Morain algorithm.

2.2 AKS

AKS is a relatively new deterministic primality proving algorithm that runs in polynomial times $O(\log^6(n))$ [7]. The idea presented by [2], is to make use of a generalization of Fermat's Little Theorem: n is prime iff the following holds for $a < n$ such that $gcd(a, n) = 1$: $(x + a)^n \equiv (x^n + a) \mod n$. This is too computationally expensive, so the inventors of AKS use a trick: pick a special r value such that the following holds $(x + a)^n \equiv (x^n + a) \mod (x^r - 1, n)$

The algorithm is as follows from [2]:

1. If ($n = a^b$ for $a \in \mathbb{N}$ and $b > 1$), output COMPOSITE.
2. Find the smallest r such that $o_r(n) > \log^2 n$.
3. If $1 < gcd(a, n) < n$ for some $a \leq r$, output COMPOSITE.
4. If $n \leq r$, output PRIME (only relavent when $n \leq 5,690,034$)
5. For $a = 1$ to $\lfloor \sqrt{\phi(r)} \log n \rfloor$ do
   if $((x + a)^n \neq x^n + a \mod (x^r - 1, n))$, output COMPOSITE.
6. output PRIME.

Literature states that the main bottleneck of AKS is in step 5, testing the congruence $(x - a)^n \equiv (x^n - a) \mod (x^r - 1, n)$ where n is the input number, a ranges from $1...2\sqrt{r} \log n$, and r is a number between $2 \leq r \leq \log^6 n$. This is where a lot of the literature focuses optimization attempts. For instance, [5] describes a way to conduct this operations using Binary Segmentation. Binary Segmentation is a way of encoding the polynomial into a large integer and that enables polynomial multiplication via large integer multiplication. This relatively efficient and simple because it leverages fast multiplication of large integers.

Another approach to this, proposed by [9], is to not fully expand the polynomial before reducing it and perform modular arithmetic while computing the coefficient of each term. In order to reduce a polynomial of degree $r - 1$ that has been squared, one subtracts the coefficients with degrees between r and $2r - 2$ by r and add them to the existing coefficients of that degree. The authors of this paper boast a dramatic speedup over their original naive polynomial division algorithm.

[5] proposes yet another way to compute these polynomial exponentiations by using a windowed power ladder, which is essentially a more efficient variant of successive squaring.

[3] noticed that the polynomial congruence is satisfied for some composite numbers chosen for n. It was observed that all of these composites are Carmichael Numbers and r's divide $n^2 - 1$. Thus, r can be selected differently such that only one congruence needs to be checked: $(x - 1)^n \equiv (x^n - 1) \mod (x^r - 1, n)$. This relies on an unproven conjecture given in [10].

2.3 Miller-Rabin

The Miller-Rabin algorithm is quite simple and has a surprisingly fast runtime, the catch being that it is probabilistic and cannot produce a certificate of primality. The de facto algorithm for creating RSA keys, Miller-Rabin tests the primality of n by randomly choosing an $a \in \mathbb{Z}/n\mathbb{Z}$ and checking if it is a witness for the compositeness of n. This is done k times, ensuring with a chance of $1 - 1/4^k$ that the result isn't a false positive, meaning the number is prime when in fact it is not.

The curious might wonder, what is a witness of compositeness?
A witness a to the compositeness of n is a value for which one of the following holds:

$$\begin{cases} a^t \equiv 1 \mod n \\ a^{2^i t} \equiv -1 \mod n \; for\ some\ i\ with\ 0 \leq i \leq s-1 \end{cases}$$

where $n - 1 = 2^s t$ and t is odd

3 Our Implementation

3.1 Miller-Rabin

Our implementation of Miller-Rabin doesn't deviate much from the classic algorithm described in Algorithm 3.5.6 in [6]. We decided to mimic the gmp implementation of the algorithm in one respect, if the number is below a threshold, it brute force proves it is prime. For instance, in our impelementation if $n < 1,000,000$ then we iterate through values less than $\sqrt{n}$ and check for divisibility. In comparing our implementation against GMP's mpz_probab_prime_p we saw similar runtimes.

3.2 ECPP

Our implementation of ECPP for the most part our implementation follows the Atkin-Morain method described in Algorithm 7.6.3 of [6]. Another reference we used was the Atkin-Morain implementation found in the open source GMP-ECPP [1].

The first step involves choosing a discriminant, the reference program [1] starts with the discriminant $D = 1$ and increments up from there and eliminates only those D values that don't satisfy the following:

1. D mod 4 != 0
2. D mod 4 != -3
3. D mod 4 != 1

If the discriminant passes the Jacobi test mentioned earlier, then it proceeds with the second step of finding an appropriate curve order m. To do this it must first find a suitable u and v value. The reference program can do this by computing either a Weber curve or a Hilbert curve (the default is to compute a Hilbert curve). This process can take quite a long time for some discriminants since it must compute all the coefficients of the Weber or Hilbert curve first before finding a suitable root for the Weber or Hilbert curves. We chose to implement an alternative approach, as outlined by [6], which uses precomputed list of r and s values for a finite list of 28 discriminant D values. This was one of the first innovations used in our program versus the reference program. I fully intended to extend this table to over 600 discriminants by using the reference program's Hilbert curve computing program to produce a table of Hilbert curve coefficients. To give the reader an idea of how long it takes to produce these values, the reference program ran on our machine for over 12 days without stopping to produce all the Hilbert curve parameters from $D = -7$ to $D = -1200$.

Another key step to the second step in the Atkin-Morain algorithm involves computing a square root modulus some value n. This component is used throughout the Atkin-Morain algorithm and can be implemented in a few different ways. The first way involves determining if the value p provided (usually $p = n$) can be shown to satisfy one of the following:

1. $p \equiv 3, 7 \mod 8$

2. $p \equiv 5 \mod 8$
3. $p \equiv 1 \mod 8$

The $p \equiv 1 \mod 8$ case is the more general case and involves randomly choosing a value d and using it to compute the square root modulus some number. It was surprising to discover in testing this more general case that the reference program had implemented this wrong and always fell back on a completely different algorithm (that used floating point arithmetic we intentionally avoided) to find the square root modulus some number solution. Upon completing this algorithm and adding the finite table of discretes and r and s values our ECPP implementation became capable of proving the primality of numbers with less than 27 decimal digits.

Next we improved the third step in the algorithm, which consists of factoring the curve order m to some suitable value q that was probably prime. This is done using the LenstraECM algorithm, which is also utilized in the reference program. We believe our implementation of LenstraECM is more robust than that of the reference program because the they only choose one small random value ($a <= 2^{32}$) when finding a curve to use for finding factors in m. Our implementation follows [6]'s recommendation and chooses three random values ($x, y, a <= n$).

This significantly improved our ability to handle prime numbers greater than 27 decimal digits. In implementing the 4th and 5th steps in the Atkin-Morain algorithm, it was observed that the reference program was computing the curve parameters a and b incorrectly. Our implementation tests several different points (x, y) against each curve parameter a and b available before trying another point. The reference application tries multiple times using the same point on the same curve parameters a and b before trying to use another curve parameter. What we found is that the reference application wasted time without getting any results by doing this. Our algorithm improves on this by only trying the point (x, y) once for each curve parameter choice available before choosing another point to try, improving our chances of finding a solution faster than the reference program. Even after trying more than 100 points for each curve parameter, a solution proving that n is prime if q is cannot be found. The reference application increases the time spent in the LenstraECM algorithm and starts over with discriminant $D = 1$ again. We kept this similar technique for our implementation since we only have 28 discriminants to choose from.

Even with this addition, if n is sufficiently small, our implementation can fail to find a suitable discriminant or curve order. In these cases we decided to use the AKS algorithm to test if n is prime. Since each iteration through the Atkin-Morain algorithm sets $n = q$ it can be shown that the AKS algorithm might be used on occasion. Since the sieve test (test every prime number from 2 to $\sqrt(n)$) makes an effective test for small values for n it was decided to add a sieve test for all $n <= 2^{32}$. This limits our use of AKS to only those values of $2^{32} < n < 2^k$. Where k is likely some value less than 89. All of these improvements significantly improved the running time of our implementation over the reference implementation. It does not however come close to matching the running time of the fastECPP implementation. Therefore further improvements can be made to our implementation to improve our results.

3.3 AKS

Our implementation of AKS is in C++ and uses GMP, as well as MPFR for floating operations such as computing $\log^2 n$.

For steps 1 through 4 our implementation is fairly straight forward. Step 1 we use mpz_perfect_power_p to check if n is of the form a^b where $a \in \mathbb{N}$ and $b > 1$. For Step 2 we have a doubly nested loop that increments r by 1 starting at $\log^2 n$, looping through k values between 1 and $\log^2 n$ until the equation $n^k \equiv 1 \mod r$ is satisfied. In Step 3 we loop through possible a values less then r and ensure that $1 < gcd(a, n) < n$ holds. Step 4 we check if $n \leq r$ for the cases that $n \leq 5,690,034$.

Now for Step 5, where the majority of the complexity and computation is introduced. We uses an array of coefficients to represent our polynomials. Since everything is modulo $x^r - 1$ we know that there will be r elements in the array. This representation makes it simple to compute polynomial multiplication modulo a number. In essence it is a doubly nested for-loop that multiplies two coefficients from indices i and j and places the result in $t = i + j \mod r$. With this array representation we utilize a naive successive squaring algorithm to do the polynomial exponentiation.

Unfortunately, while running our experiments, but after submitting our code, we discovered that our AKS implementation has bugs. Something was causing it to, on rare occasions, decide that a composite number was in fact prime. Upon further inspection we found that our successive squaring had wrong loop parameters and our loop to check if $(x+a)^n \neq x^n + a \mod (x^r - 1, n)$ holds didn't check all values.

We fixed the implementation, but without much time to optimize it, we were left with very poor running times. The bugs rarely reared their ugly heads, so we doubt they had an effect on ECPP, especially because we run Miller-Rabin before AKS.

Due to the lack of time, we only collected experimental data for our buggy implementation of AKS. We were able to run a few numbers through the fixed version and compare its performance to the buggy version. This can be seen in Figure 4 and is further discussed in the Experimental Results section.

3.4 Hybrid

Cheng's paper [4] shows that one iteration of ECPP can be used to produce a smaller number to be checked in AKS. He proved that on such iteration would take the running time of AKS down to $O(\log^4 n)$. To our knowledge, no program implementation uses ECPP in conjunction with AKS, hence there is room for novel work here. We tried his idea and document it below in our experimental results section.

4 Experimental Results

We created 3 sets of numbers that are then inputted to various implementations: a set of 600 composite numbers ranging from 1 to 200 decimal digits (1 to 664 binary digits), a set of 458 primes ranging from 1 to 200 decimal digits (1 to 508 binary digits), and a set of 17 prime numbers, each with 10 decimal digits, that cause our ECPP implementation to run out of discriminants and run AKS. We run these 3 input sets using 5 implementations: our ECPP, our (buggy) AKS, our Miller-Rabin, the Atkin reference [1], and Morain's fastECPP closed source binary [8]. All of these tests were run on the CADE machines, what follows is our results and analysis.

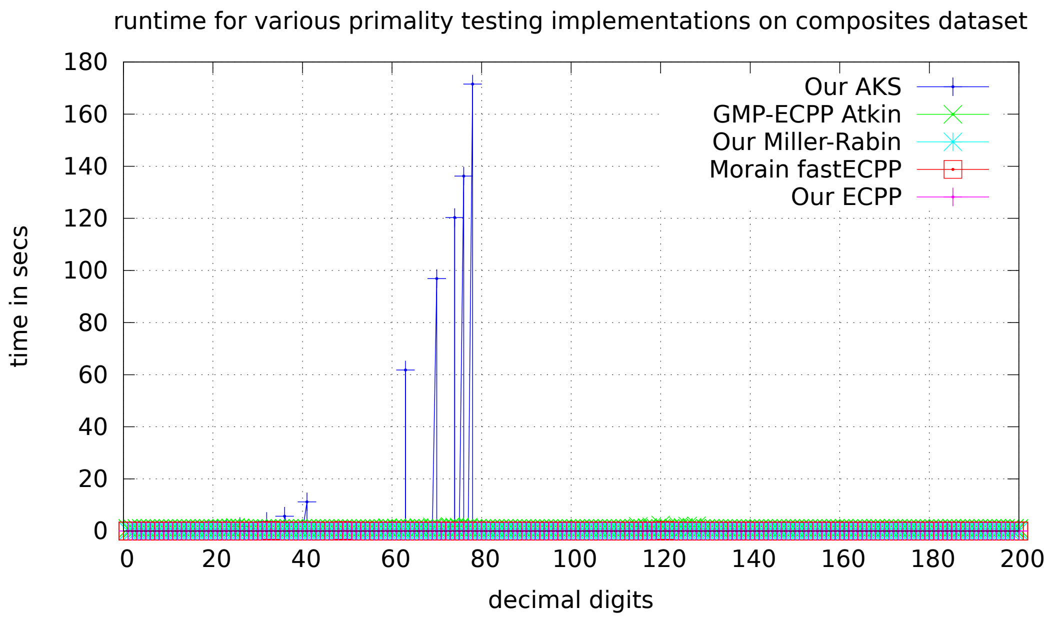

Figure 1: Composites

Since all of the implementations except AKS use some form of Miller-Rabin to filter out composites, the running times of those implementations on a set of composite numbers looks fairly uniform and fast. AKS's runtime complexity on the other hand exhibits some interesting features. In Figure 1 one can see all the implementations that make use of Miller-Rabin are extremely fast at determining composites. Most of the time AKS runs quite fast, taking less than a second. Regardless, there are several numbers that cause AKS to take much longer, that is, almost as long as proving a prime of a similar digit length. These spikes are most certainly cause by the way primes are proved in AKS. If the algorithm makes it all the way to step 5, in the worst case it will have to loop through $\lfloor \sqrt{\phi(r)} \log n \rfloor$ values of a. This worst case probably doesn't occur often for composite numbers, so one can deduce that the spikes in Figure 1 represent "hard" composite numbers for AKS.

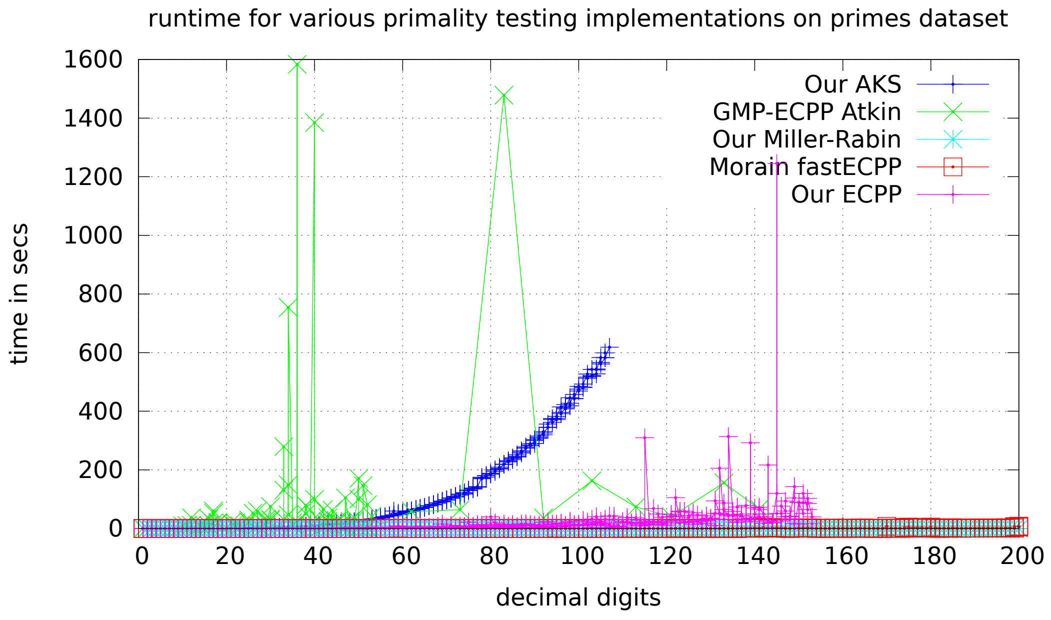

Figure 2: Primes

The main result of our experimentation can be seen in Figure 2, where we run the various implementations over a set of primes ranging from 1 to 200 decimal digits. Please note that some are cut off early or downsampled due to runtime constraints. As one can see, Morain's fastECPP blows every other test out of the water, proving a 200 decimal digit number prime in under a second. AKS exhibits a nice curve that is much slower than the rest of the algorithms. Our ECPP implementation for the most part is quite fast but has some large variation as the numbers grow in size. This is in part due to the randomization in ECPP, but it is also possible that our implementation ran out of hard coded discriminants and is falling back on AKS. We were only able to run GMP-ECPP on a subset of the numbers, but it shows much more varied runtimes when compared to our implementation.

Our ECPP implementation defaults to AKS if it runs out of discriminants. In order to test how this hybridization compares to GMP-ECPP and fastECPP we collected a few primes that cause our ECPP to use AKS ran them on the implementations. Figure 3 is the result, and it appears that some of those primes caused GMP-ECPP to take longer than usual. This is probably because GMP-ECPP computes Hilbert curves on the fly, and the numbers that cause our implementation to run through all 28 hard coded discriminants probably causes GMP-ECPP to spend a long time generating curves.

Upon discovering and fixing the bugs in AKS, we wanted to compare our buggy and fixed implementations. Figure 4 demonstrates the massive slow down incurred by correctly multiplying and checking polynomial congruence.

Lastly, we thought we would gather some experimental results for [4]'s proposed one round of ECPP to speed up AKS. As expected, Figure 5 demonstrates a nice speedup in over traditional AKS, which is granted to us by only having to prove that a q corresponding to n is prime. Unfortunately we were only able to run this with

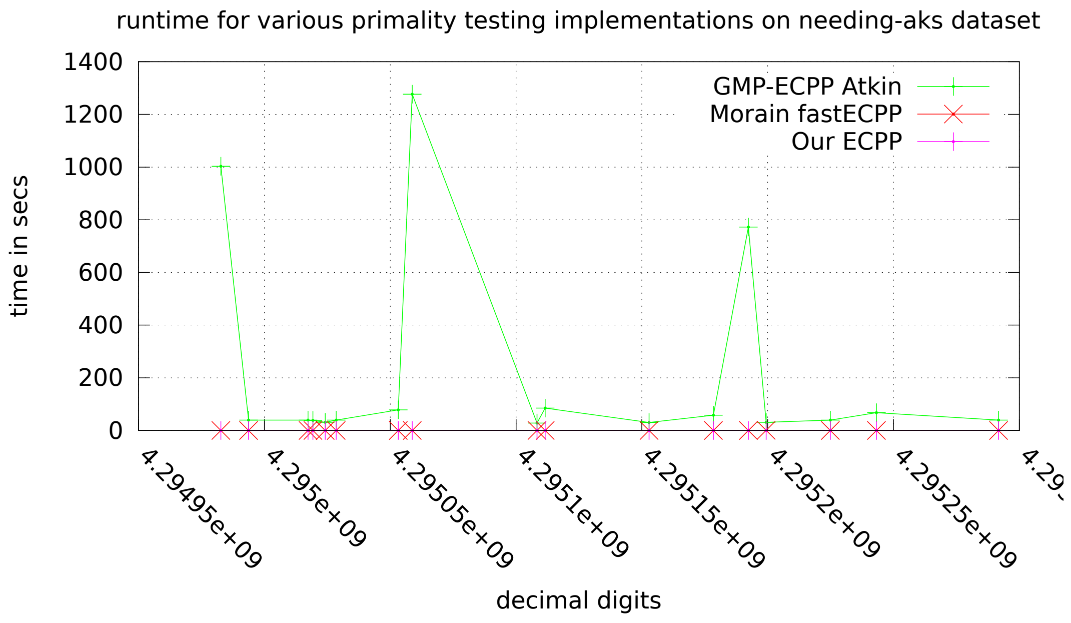

Figure 3: Numbers that require our ECPP implementation to run out of discriminants

our buggy implementation of AKS.

5 Conclusion

Primality Proving is a complicated field in which many algorithms exist, all with different advantages and disadvantages. We chose to focus on the Atkin-Morain version ECPP due to its speed. In the spirit of novelty, we also decided to implement AKS and Miller-Rabin and see what opportunities are possible with the hybridization of these algorithms.

Our ECPP algorithm holds up against comparable implementations [1] and even makes improvements in some instances. In addition we managed to make use of AKS in our ECPP implementation when we run out of hard coded discriminants. The discovery of bugs in our AKS implementation makes this choice questionable, and we will need more time to see if a correct (slower) implementation of AKS makes a good fit for this function. Regardless we make no attempt to dethrone the state of the art: [8].

While researching AKS we explored many different ideas in literature but were only able to implement one technique for polynomial exponentiation: a successive squaring method. Since we were unable to find good reference implementations it was difficult to fully analyze our implementation. The bugs furthermore introduced questions since the runtime between the broken and fixed implementations was so drastically different.

We were able to test out Cheng's formulation of one round of ECPP and one iteration of AKS [4]. This can be seen in Figure 5, and proves a nice experimental result to a theoretical formulation.

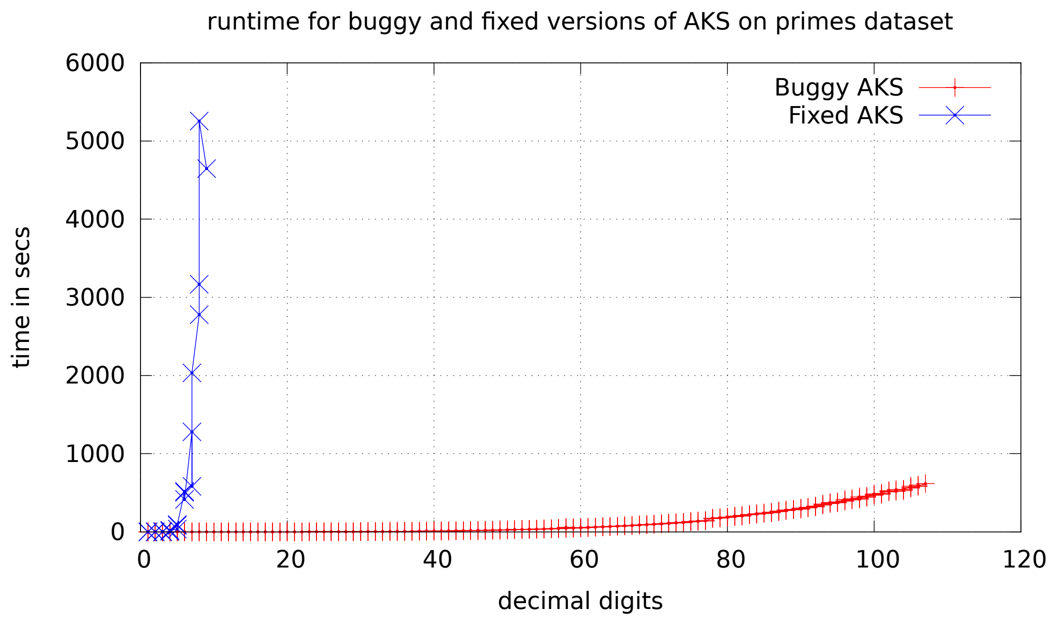

Figure 4: Comparison of buggy and correct AKS implementations

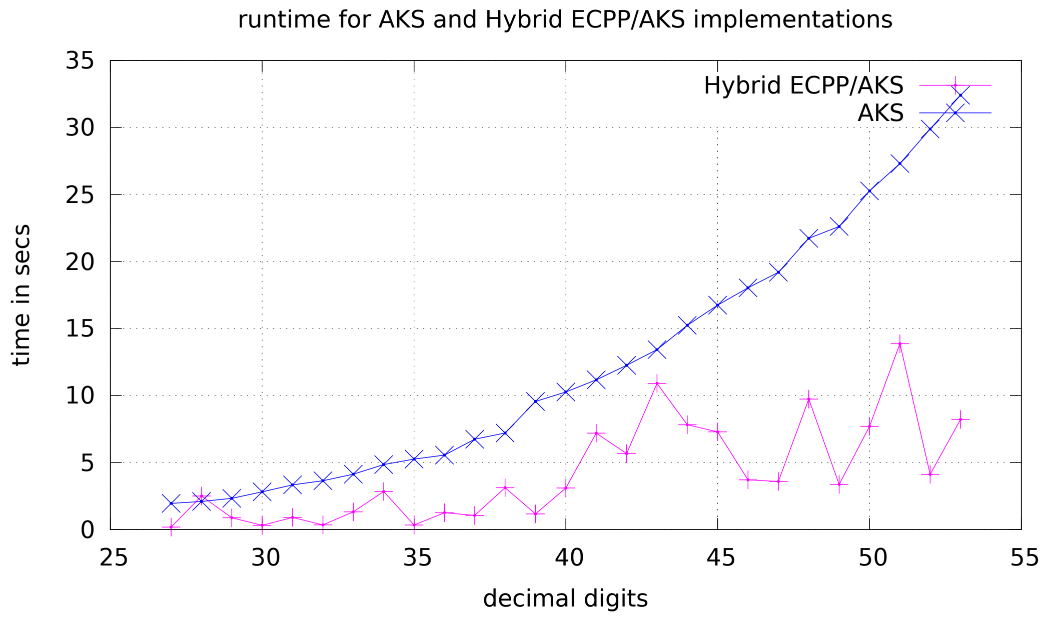

Figure 5: Hybrid AKS/ECPP compared to AKS

References

[1] Gmp-ecpp 2.37. `http://sourceforge.net/projects/gmp-ecpp`.

[2] Manindra Agrawal, Neeraj Kayal, and Nitin Saxena. Primes is in p. *Ann. of Math*, 2:781–793, 2002.

[3] Vibhor Bhatt, G K Patra, Vibhor Bhatt, and G. K. Patra. Analysis on implementation and its further improvements of aks class algorithms, 2003.

[4] Qi Cheng. Primality proving via one round in ecpp and one iteration in aks. In *Advances in Cryptology CRYPTO 2003*, pages 338–348. Springer Verlag, 2003.

[5] R. Cr and J. Papadopoulos. On the implementation of aks-class primality tests, 2003.

[6] Richard Crandall, Carl Pomerance, Richard Crandall, and Carl Pomerance. Prime numbers: a computational perspective. second edition, 2005.

[7] H. W. Lenstra and Carl Pomerance. Primality testing with gaussian periods. Technical report, 2003.

[8] F. Morain. Implementing the asymptotically fast version of the elliptic curve primality proving algorithm. *Math. Comp.*, 76:493–505, 2007.

[9] R G Salembier and Paul Southerington. An implementation of the aks primality test. *Computer Engineering*, 2005.

[10] Arnold Schnhage. Asymptotically fast algorithms for the numerical muitiplication and division of polynomials with complex coefficients. In Jacques Calmet, editor, *Computer Algebra*, volume 144 of *Lecture Notes in Computer Science*, pages 3–15. Springer Berlin / Heidelberg, 1982.