Elliptic Curve Factorization in C, Final Project

by Jeffrey Blakeslee

The Elliptic Curve Factorization method (ECM) developed by H. Lenstra, is used to factor composite integers in the range of 50 digits. The algorithm attempts to process numerous multiplications over an elliptic curve group, failing when the addition of two points cannot be calculated in the group. This occurs since the operations are not performed over an actual group, but a pseudo-group defined by the composite integer that we attempt to factor.

This paper will give a brief explanation of Pollard's p-1 algorithm, an explanation of the ECM algorithm, and then describe my process of developing an implementation in C. It will be beneficial to present Pollard's p-1 algorithm first, in order to provide a sampling of the ECM; it is very similar, but slightly less complex.

Pollard's p-1 relies on Fermat's Little Theorem: $a^{p-1} \equiv 1 \bmod p$. The method utilizes the fact that for any k, $a^{k(p-1)} \equiv 1 \bmod p$ which when we consider p a divisor of n provides for $GCD(a^{k(p-1)} - 1, N)$ to be a factor of N. By selecting some bound B, which we hope is p-smooth, we simply calculate, $m = B!$, and then $GCD(a^m - 1, N)$, providing a factorization of N when the 1 < GCD < N.

The ECM uses the same general concept, however the calculation is B!*P, for some point P, rather raising to a power. The further difference with ECM, which actually accounts for much of the benefit over p-1, is that many random curves can be used for this operation. The algorithm proceeds as follows. Select a random curve and random point on the curve: $y^2 = x^3 + ax + b \ (\mathrm{mod\ N}),\ \mathrm{P{=}(0,1)}$

This is done by choosing a random point first, then solving for a and b, which gives the curve. We then calculate B!*P, unless during an addition, an inverse calculation fails, providing a factor of N. If all of the additions succeed, select a different point and curve and repeat the calculation of B!*P. Given two points, the (x_1, y_1) (x_2, y_2) inverse calculation can fail during the calculation of the slope when $x_1 \neq x_2$ and $y_1 \neq y_2$ or when the two points are the same.

My process of developing a software implementation of the ECM, began in Python. I implemented the inverse calculation, the GCD, Elliptic curve addition, and the factorial calculation in Python to factor N=455839. This allowed me to develop an understanding of how to reduce the factorial code down into a function. Once I started trying to factor a number as large as 9801097, it became too slow. I move to C and used GMP, which has an inverse_mod and GCD function built-in. Once I had this working in the same form as my Python code, it was apparent how much more quickly this code worked. The code factored the 34 digit number, 96061208566745701160740922300007191, in about 4 minutes, utilizing a bound of 5000, and 90 curves. Next, I added Pthreads which allow the program to search multiple threads at once. The first problem encountered here was that the stack size allocated per thread wasn't sufficient for the memory needed to store the array of points. Using the Pthread code, I was able to double the size of the stack per each thread, and obtained the factorization of 960612085667456598697625223710010171533, a 38 digit number using a Bound of 11000, and 36 curves, in about 10 seconds.

In conclusion, the project gave me the opportunity to gain a much better understanding of elliptic curves, as well as to gain experience using GMP and Pthreads. In the future, I would like to continue to make improvements to this program;

like implementing the addition operation in the projective plane, as well as attempting to find an optimal Bound and number of curves that will be more generally applicable.

Bibliography

R. Crandall and C. Pomerance, *Prime Numbers, a Computational Perspective,* Springer, 2005.

W. Trappe and L. Washington, *Introduction to Cryptography with Coding Theory,* Pearson, 2006.

http://en.wikipedia.org/wiki/Elliptic_curve_method