Sieving the Seven Seas

By Danielle Aloicius, Joshua Finer, Annette Paciorek, and Xiang Xu

Abstract

We become the latest in a rich history of mathematicians aiming to grasp as much as we can about the subtleties of factoring large composite numbers. We humbly recount the methods of those who have paved our way and our successes and failures implementing these methods, with a focus on the quadratic sieve factoring algorithm as we understand it. We intercept a message and thwart an evil enemy.

Preface: Overview of RSA

We have spent the past month obsessed with recovering a message encrypted using the RSA mechanism. RSA is so named for its inventors Ron Rivest, Adi Shamir, and Len Adelman. In general terms, Alice is able to send an encrypted message to Bob if she knows his public key of the form (e, n), where n is some composite number difficult to factor (we will later explore just what this means in greater detail) and e is a public exponent. Alice takes her message, some number g and calculates $m = g^e \bmod n$ and sends m to Bob. Bob's private key is $d = e^{-1} \bmod \varphi(n)$. The security of RSA rests in the perceived difficulty of determining d. If an eavesdropper can factor n, then this eavesdropper can calculate $\varphi(n)$ and d in turn. $\varphi(n)$ has proven difficult to compute for large n if the the factorization is not known. A variety of attacks exist against RSA other than factoring; we describe some of these. We also particularize the exact parameters of our challenge and the approach that was most appropriate for us.

The Intercept

An Exercise in Ekphrasis, Inspired by Dante's Inferno

Midway in our term's journey, we had a say
in our choice of project and woke to find ourselves
in a group of four, tasked to break an RSA

cryptosystem! We never saw so dear
an intercept to us, so important a message!
The convoluted which must be made clear.

Death could scarce be more bitter than failure!
But since we learned very much, we will recount
all that we found revealed about factoring large integers.

How we came to it, we will thoroughly relay,
so consumed and deprived of sleep we had become
when we attempted to break an RSA.

On April First, 2014, we intercepted a critical message from our Great Foe. As The Enemy works in mysterious ways, he encrypted his message using the RSA method with a modulus of 495960937377360604920383605744987602701101399399359259262820733407167 which we call *n* and an exponent of 31, called *e*. It came to us as 19705178523446373241426321455642097240677633038639787310457022491789, or what we will refer to as *m*, and offered the promise of a communication so valuable it would change our lives forever. Thus motivated to retrieve the decrypted message, we heeded the advice of Dr. Kent Boklan, who in his infinite wisdom once said, "once a method of encryption has been identified, the most potent weapon in a cryptanalyst's arsenal is to exploit mistakes and lapses." [1]. Hoping to do just that, we parsed through Dan Boneh's paper "Twenty Years of Attacks on the RSA Cryptosystem." As the title indicates, Boneh outlines vulnerabilities in the RSA cryptosystem and how one may protect against exploitation of these vulnerabilities, many of which arise from poor decisions in implementation.

Dan Boneh groups various historical attacks into five main categories, which are factoring; elementary attacks; low private exponent; low public exponent; and implementation attacks. Elementary attacks are most effective against foolish users of RSA, such as those who send the same message *e* times, or who share a modulus *n*. Our foe has shown no signs of such recklessness, and thus proves doubly nefarious. Elementary attacks, therefore, do us no good in this instance. Low private exponent attacks, though interesting, also do not apply. Because $ed \equiv 1 \bmod \varphi(n)$, and $e = 31$, we suspect *d* is considered large, specifically $d > 1/3\ (n)^{(1/4)}$, though we do not know the sizes of the factors of *n*. In this situation, we *do* know that since $e = 31$, low public exponent attacks would be more useful for our purposes. The only applicable low exponent attack Boneh summarizes involves LLL lattice basis reduction. We smiled awkwardly at Coppersmith's Theorem and put it in the "To Research" pile. Lastly on Boneh's list were implementation attacks. Timing attacks are quite interesting. Paul Kocher showed that one can recover *d* by measuring "the time it takes a smartcard to perform an RSA decryption (or signature)" [2]. Furthermore, one can even measure a smartcards' power consumption to determine *d*. To our chagrin, the Evil Encryptor was able to effectively shield this information from our operatives. Our two best options, therefore, were factoring and low public exponent attacks. Having had some experience factoring integers, we opted to attempt a factorization of *n*. The majority of this paper will be a documentation of those attempts.

Factoring

Our first action was to test the primality of *n*. If *n* is prime, then *n* is easy to factor, namely, $n = n \cdot 1$. We used the Miller-Rabin primality test, which returned that *n* is composite. Miller-Rabin is a *probabilistic* test for primality, but if Miller-Rabin indicates a number is composite, the number is indeed composite. Therefore we were completely confident that *n* was not prime and, following this, our factoring efforts began in earnest.

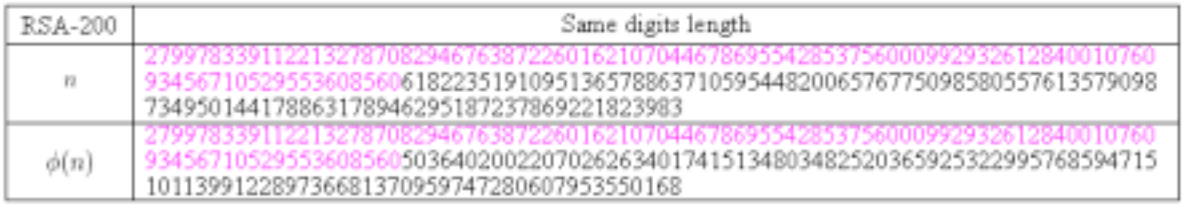
As a first approach, we chose to implement trial division and the Pollard-Rho factoring algorithm concurrently. Knowing nothing of the size of the factors of *n*, it was plausible that *n* may have had factors small enough that trial division would be the fastest algorithm to pull them

out. Our crude version of trial division successively attempted to divide n by consecutive odd numbers, and returned a factor of 809 in a fraction of a second. The Pollard-Rho algorithm involves iterations of a quadratic polynomial $f(x)$. These values of $f(x)$ form a pseudo-random sequence, to which we may apply the general idea of the birthday paradox, that "if elements are chosen randomly from a set of size n, with replacement, a repeat is likely after $O(\sqrt{n})$ choices" [3]. This repeat value will indicate a cycle, and allows us to forego superfluous calculations. If, however, the greatest common divisor of two values of $f(x)$ is ever greater than 1, then a prime p that divides this greatest common divisor is also a factor of the composite input. Running Pollard-Rho with the polynomial $x^2 + 2$ at initial values 2 and $f(2)$, a factor of 809 was returned after thirty-five iterations. Even printing values of the polynomial and of the greatest common divisor at each iteration, the computation terminated in less than one second. We divided n by 809, and the result was, successfully, a whole number. Specifically,
$n_1 = n / 809 = 613054310726032886180943888436325837702226698886723435429939101863$.
Here we note that n is 69 digits in length and n_1 is 66 digits long. We employed the Miller-Rabin primality test once again, to determine whether or not n_1 is composite. Having confirmed the primality of 809, if n_1 were prime as well, then we would easily be able to recover
$\varphi(n) = \varphi(809) \cdot \varphi(n_1) = \varphi(808) \cdot \varphi(n_1 - 1)$, and use this $\varphi(n)$ to obtain $d = e^{-1} \bmod \varphi(n)$, the critical value for the decryption of the message. By chance or by design (design of the surreptitious Encryptor, no doubt), n_1 was confirmed composite by Miller-Rabin, thereby putting $\varphi(n_1)$ further beyond our reach than simply calculating $n_1 - 1$, and what fun would that be, anyway?

Having pulled out a factor of 809, our task at hand was determining φ of the remainder, now n_1, which we chose to calculate by factoring n_1. Hoping that Pollard-Rho would bring us success once more, we started running the algorithm on April 3, and ended on April 25 without result. This program did not run continuously, but was stopped occasionally to change the values of the polynomial and tinker with the program. Similarly we ran trial division for a time as well, and determined that the remaining factors were no smaller than 9 digits in length. At worst case scenario, we estimated trial division would take somewhere on the order of 10^{32} years. As a comparison, the sun's lifespan is about 10^{10} years, and we are currently about halfway through that lifespan. In fact, trial division would outlive the entire universe (probably).

φ **Folly**

Just before we fully committed to the factorizing the modulus, one last factorization-free approach presented itself. This approach hoped to capitalize on a characteristic specific to the encryption process. The first step in the encryption scheme utilized by our Adversary is the conversion of a written message to numbers. The letter-to-number relation our enemy used takes A to 1, B to 2, ..., Z to 26. We observed that, as a result, zeroes appear in a numerical representation only if the original message had a J or a T, *i.e.,* only as 10 or 20. Furthermore, assuming that the message is a typical English sentence, certain sequences of numbers are unlikely to appear. These unlikely sequences include strings of numbers like "999," "888," and other triples, which would represent strings of letters in the original message like "iii", "hhh" and so on up to "333". "111" and "222," on the other hand, were quite plausible, as "111" may represent “ak” or “ka” and “222” might be the numerical representation of “bu” or “ub”. Thus a potential attack would involve calculating possible values of φ (n_1), multiplying these by φ(809), namely 808, to obtain a range of values of φ(n), and then calculating values of d for each of these potential φ(n)’s. Then raising the intercept m to the power of d mod n, we thought we might discard the decrypts that contain strings of numbers unlikely to occur in the original message. Because n_1 is composite, we know it is the product of at least two primes p and q, with p, $q > 2$. φ(n_1), therefore, has factors φ(p) and φ(q), more specifically p - 1 and q - 1. Since p and $q > 2$ are both odd, p - 1 and q - 1 each have a factor of 2, and so it follows that 4 | φ(n_1), and φ(n_1) ≡ 0 modulo 4 [7]. An obvious stumbling block is the sizable range of possible values of φ(n_1). Liu and Ye observe that for moduli of 200 digits and more, n and φ(n) share roughly half of their leading digits in common. They demonstrate an example in the table below:



RSA-200	Same digits length
n	27997833911221327870829467638722601621070446786955428537560009929326128400107609345671052955360856061822351910951365788637105954482006576775098580557613579098734950144178863178946295187237869221823983
$\phi(n)$	27997833911221327870829467638722601621070446786955428537560009929326128400107609345671052955360856050364020022070262634017415134803482520365925322995768594715101139912289736681370959747280607953550168

[7]

Before calculating more precise upper and lower bounds of φ(n_1) [10], and assuming this pattern holds for 66-digit numbers, our initial estimate of the search space contains about 10^{33} possible values of φ(n_1), and therefore 10^{33} possible decrypts. Nevertheless, curious to see how many of these decrypts would actually conform to our rules (no instances of “30”, “40” etc, nor 999, 888,...) we wrote a fairly straightforward program and used to it to test a small range of φ values to get an idea of how plausible this attack could be. We initialized our possible φ value was the first v such that $v < n_1$ and $v \equiv 0$ modulo 4.
The program multiplies v with 808
Calculates $d = e^{-1}$ mod n
Calculates m^d mod n
Prints m^d mod n if it conforms to the rules
Sets $v = v - 4$ and repeats.
We ran this program starting at
v = 613054310726032886180943888436325837702226698886723435429939101860 and our program ran through a short interval printing more than one possible decrypt per second. It was immediately apparent that this method was a Fata Morgana and not sensible enough to warrant further research. We do note, however, that perhaps this attack may be more useful against a more exploitable letter-to-number relation and more refined bounds on φ. If this ever were to be realized as a viable attack, a simple safeguard would be a pre-permutation of the letters, so that A does not map to 1, B does not map to 2, ..., Z does not map to 26. This would prevent attackers from being able to apply a sort of language recognition attack of this nature.

Ancestors of the Quadratic Sieve

The quadratic sieve can trace its ancestry back to Fermat's difference-of-squares technique, which aims to represent n, the number to be factored, as a difference of two squares. If one knows $n = x^2 - y^2$, then it follows that $n = (x + y)(x - y)$. This method works well if the factors of n are reasonably close to $\sqrt{n}$, "but in its worst cases, the difference-of-squares method can be far worse than trial division" [9]. Generalizing to congruences, if

(1) $x^2 \equiv y^2$ modulo n

and if $x \not\equiv \pm y$, then $(x + y, n)$ and $(x - y, n)$ are proper factors of n [14]. About 50% of the time, the greatest common denominator of $(x \pm y, n)$ will trivially be 1 or n, so factoring methods motivated by this congruence "focus on finding multiple $[x, y]$ pairs so success is probabilistically ensured" [16]. The quadratic sieve factoring algorithm is a highly refined algorithm based upon this essential idea. As we approached the quadratic sieve method from a place of understanding of the method of Fermat, we programmed a prototype of the algorithm, which was actually an implementation of Dixon's method. We describe this prototype as follows.

Our first action was to implement a Sieve of Eratosthenes[1] in order to generate a list of primes. In 1.1 seconds, our Sieve of Eratosthenes generated a list of primes up to 100,000 and functions as a deterministic primality test. It is a more practical method of generating primes on this interval than Miller-Rabin for instance, which is comparatively slower and probabilistic to boot. Our next consideration was choosing a smoothness bound for the factor base. Smooth numbers are a critical concept to most factoring algorithms and several other applications of number theory. A number is said to be B-smooth if all of its prime factors are less than or equal to B. Carl Pomerance discusses the usefulness of smooth numbers in some specific algorithms in his paper "The Role of Smooth Numbers in Number Theoretic Algorithms," and we will give a brief review here in the context of the quadratic sieve. The concern of the quadratic sieve is seeking solutions to $x^2 \equiv y^2$ modulo n. Pomerance poses and proves a lemma that gives a bound N on the smallest nonempty subset of $[1, x]$ such that the product of N-many randomly selected numbers from the interval $[1, x]$ is a square with probability 1 [11]. Furthermore, all those N many numbers are $L(x)^{\frac{\sqrt{2}}{2}}$ - smooth, where $L(x) = \exp(\sqrt{\ln x \ln \ln x})$. The factoring methods we explore hinge upon finding products that are squares. By Pomerance's lemma, we can greatly reduce the interval in which we search for these relations, and be confident that they exist, and this subset of smooth numbers is called our factor base [5][12]. We therefore choose a smoothness bound to search through a subset of integers up to n to find solutions to $x^2 \equiv y^2$ modulo n. In our rudimentary precursor to the quadratic sieve, we chose a smoothness bound of 8,000 which we arrived at after some experimentation. We used the Sieve of Eratosthenes to determine a factor base of all 1,007 primes up to 8,000. Note that we did not discard non-quadratic residues in this preliminary algorithm. The issue of quadratic residues became a crucial optimization as our method evolved and we shall discuss it presently. Having established a factor base, we then search for numbers of the form $(i \sqrt{n} + j)^2 \equiv 0 \bmod n$, where $x = (i \sqrt{n} + j)$ and

[1] algorithm for making tables of primes [15]

indices i and j range over values of 1 - 100 and 1 - 10,000, respectively[2]. For each i, j, we tested to see if x^2 is smooth over our factor base. We did so by dividing by every element in our factor base that was a factor of x^2. If, after dividing by all primes in the factor base, our x^2 had come to equal 1, then we determined it was B- smooth and we saved the pair $\{x, p \cdot q \cdot ... \cdot r\}$ to a list, where $p \cdot q \cdot ... \cdot r$ is the prime factorization of x^2 and primes $p, q, ... , r$ are in the factor base. If, after dividing by all primes in the factor base, x^2 had not reduced to 1, then we discarded it and any partial factorization we may have obtained. From our list of pairs, we selected the prime factorizations $p \cdot q \cdot ... \cdot r$ and constructed a matrix A of exponent vectors modulo 2. For instance, if the prime factorization of some x^2 were of the form $p^3 \cdot q^4 \cdot r^2$, our exponent vector would be $\{3, 4, 2\}$, and would become $\{1, 0, 0\}$ mod 2. The goal is to find a linear combination of exponent vectors equal to 0, because that would indicate all even powers of primes, which is exactly a perfect square. This would correspond to finding the y^2 on the right-hand side of (1). The null space of A^T gives us exactly those linear combinations. We then iterate through the null space and retrieve the appropriate pairs $\{ x, p \cdot q \cdot ... \cdot r\}$, and obtain $x^2 \equiv y^2$ where y^2 is the product of the prime factorizations of x^2. With this congruence we can check the greatest common divisor. If the greatest common divisor = 1 or n, then $x \equiv \pm y$. If $1 <$ gcd $< n$, then the gcd is a nontrivial factor of n. If not, we continue to the next solution in the null space and test it.

Optimizing the Quadratic Sieve

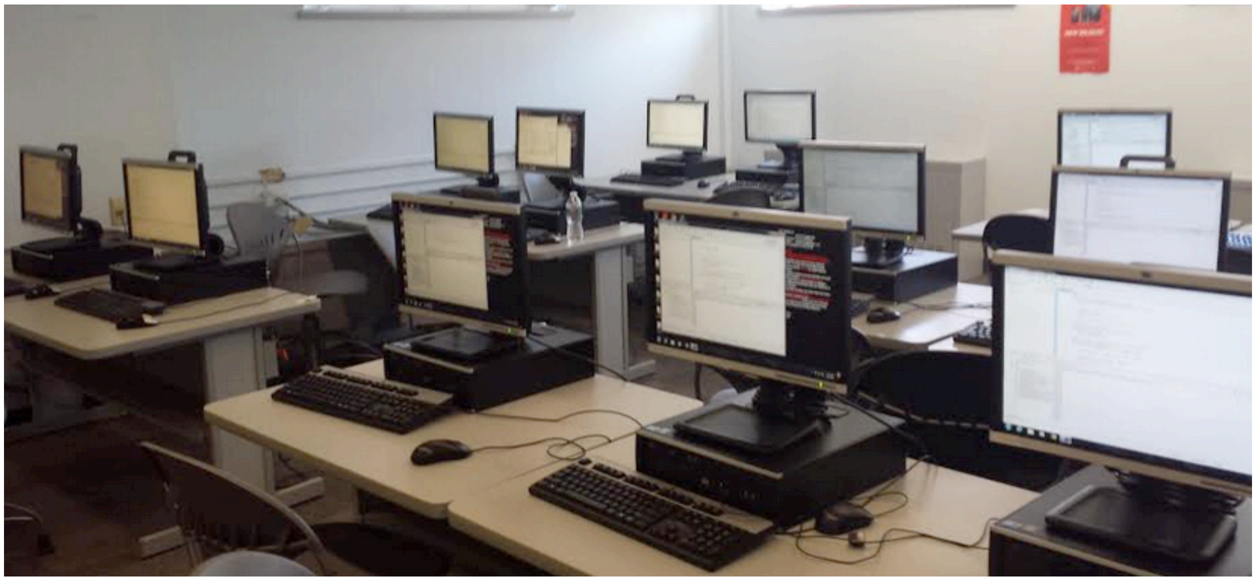
This factoring method allows us to factor numbers 20 digits with factors close in size in length in less than half an hour, but there were several issues which we needed to address as we were scaling it up. The factor base we used is bulky, for it contains prime numbers that will never divide the polynomial $Q(j) = (i\sqrt{n} + j)^2 - n$. Simply put, "only those odd primes p with $(n / p) = 1$ can divide [$Q(j)$]" [8]. Here (n/p) is Euler's Criterion [6], and we may use it to weed out primes that are useless to us by discarding all p such that $(n/p) = -1$. In general, for smoothness bound B, we can estimate that the size of our factor base containing only quadratic residues will be about $B / 2 \log B$ [8]. Another disadvantage of our preliminary algorithm is that it didn't sieve at all, given that our prototype was an algorithm of Dixon's factorization method [16]. Herein comes the actual *sieve* part of the quadratic sieve. We simplify our polynomial to the form $Q(i) = (\sqrt{n} + i)^2 - n$, and observe that $(i\sqrt{n} + j)^2 - n \equiv 0 \bmod p$ only if $p \mid (i\sqrt{n} + j)^2$. If p divides $Q(j)$, then p divides $Q(j + p)$, $Q(j + 2p)$, ..., $Q(j + kp)$ for $k \in \mathbb{Z}$. Thus we know division occurs at intervals of length p. In our previous algorithm, we were attempting to divide along each number in the entire interval. The application of Euler's Criterion to our factor base and the implementation of an actual sieving mechanism were the two most significant modifications to the preliminary algorithm, and demanded further alterations to our code to accommodate new issues that arose. For instance, sieving in our new program required an algorithm to compute square roots of n within a modulus. For this purpose we chose to implement the Shanks-Tonelli algorithm. With these adjustments, our new algorithm processed pairs of numbers, which we called the Left values and the Right values, or L_i and R_i, such that $L_i = \lfloor\sqrt{n}\rfloor + i$;

[2] n is the number we wish to factor. We do not yet use n_1 in the polynomial because we were not ready to attempt a factorization on that scale.

$R_i = (\lfloor\sqrt{n}\rfloor + i)^2 - n$; we tested each R_i to see if it was smooth over the factor base, which we did by sieving as opposed to the trial division in the previous version of our program. For any prime p in our factor base, $p \mid R_i$ only if $(\lfloor\sqrt{n}\rfloor + i)^2 - n \equiv 0 \bmod p$, which is equivalent to stating that $(\lfloor\sqrt{n}\rfloor + i)^2 \equiv n \bmod p$. Taking the square root of both sides yields $\lfloor\sqrt{n}\rfloor + i \equiv \sqrt{n} \bmod p$. Here we used Shanks-Tonelli to find the two roots of n, namely $r_1 \equiv \lfloor\sqrt{n}\rfloor + i_1$ and $r_2 \equiv \lfloor\sqrt{n}\rfloor + i_2$. Rearranging both congruences yields $i_1 \equiv r_1 - \lfloor\sqrt{n}\rfloor$ and $i_2 \equiv r_2 - \lfloor\sqrt{n}\rfloor$. These two i values indicate locations such that $p \mid R_{i1}$ and $p \mid R_{i2}$. We also know that p divides R at locations of the form i_{1+kp} and i_{2+kp} for $k \in \mathbb{Z}$. Thus we are able to pinpoint locations based on the arithmetic progression of p and spare ourselves unnecessary computations. If the value of L = 1 after dividing by all primes, then we save it and collect all such L. We are not dividing by higher powers of primes, however, so some numbers that are smooth over the factor base won't be fully reduced to 1 by our algorithm, and values of L such that L > 1 are discarded. Thus we felt another modification was in order so that we may catch some of these products of powers of primes. We chose a threshold t such that t is greater than the largest prime in our factor base but smaller than the next largest prime, namely $t = 752250$, one less than the first prime outside of our factor base. If the value of L < t after dividing by all primes in our factor base, then we know L is smooth over our factor base and we can accommodate it as a product of higher powers. Some other fine-tuning included pre-computing the factor base and solutions to Shanks-Tonelli. Also, further optimizations that impacted the running time favorably included storing the value of $\lfloor\sqrt{n}\rfloor$ rather than computing it each time, and expanding the polynomial Q(i) to $\lfloor\sqrt{n}\rfloor^2 + 2i\lfloor\sqrt{n}\rfloor + i^2 - n$, so we were no longer redundantly computing $(\lfloor\sqrt{n}\rfloor + i)^2 - n$. As a programming note, our primitive algorithm was implemented in Mathematica. As we began to pre-compute the values of the polynomials, we noticed Java outperformed Mathematica in terms of timing, so our quadratic sieve and all its tweaks were implemented successfully in Java.

Anchors Away

After much tinkering, we were able to resolve maddening issues in our program (such as miniscule changes that had massive effects), and we were ready to attempt a factorization of n_1. With some experimentation, we chose a factor base of size 30,000. The greatest prime in our 7 factor base was 752,207. We were testing intervals of size 500,000 and we named each of these intervals a round. We did some preliminary calculations to estimate how much time we might need. The first round takes about 10 seconds, the first 50 rounds take about three minutes and returned 22 relations. We ran the first two thousand rounds over the course of several hours. It was clear immediately that we were in need of substantial computing power and time. We harnessed all that we could by locating a computer lab at Queens College and parallelizing our quadratic sieve algorithm over 25 computers at a time (and we didn't even have to foot the electricity bill!). Each computer was assigned intervals to sieve through, and the relations collected were compiled into a master file. On the first day, we had about 4,500 relations, and on the second day we obtained another 9,000. As we climbed into higher intervals, the number of relations has decreased somewhat, but remained steady enough for quite some time. We obtained a total of 31,000 relations, but Silverman observes that, "in practice it is not necessary to collect F + 1 factorizations. Usually, having about .9F [F = size of factor base] rows in the matrix is sufficient to find a dependency" [14]. Silverman's implementation "acheived having N satellites give an N-fold speedup" [14]. It turns out our 25-computer takeover of the lab at Queens College was critical to our success.



Our "square sweatshop"

After the Sieve

Once we accumulate enough relations, faced the task of data processing. We constructed the matrix M from the exponent vectors of the factor base and the relations over that factor base. Thus, M has as many columns as there are primes in our factor base, *ie* 30,000, and as many rows as we found relations. For safety's sake we found 31,000 relations over the course of 72 hours, running from 5 to 25 computers at a time, or over 400 hours of computer time over all machines. The text file containing these relations was about 1.5GB in size. We were anxious to proceed to the processing step, and constructed a matrix from 29,704 relations. We transposed M and found the null space. The null space indicated which exponent vectors multiply to perfect squares. We tested the first vector in the null space, which gave us a congruence of squares, and luckily, $x \not\equiv \pm y$. The number we recovered was 9457663801784055781292440587131633[3], and 613054310726032886180943888436325837702226698886723435429939101863 / 9457663801784055781292440587131633 = 64820903298591432157114065708311[4].
We checked these two factors for primality just to be safe, and were confident we had found the prime factorization of n.

We began shouting and dancing in the library celebrating our success, and then exchanged gruff words with surrounding students whose studies we had disrupted. We were then able to calm down and easily calculate
$\varphi(n)$ = 495347883066634572034202661856543582695757465921524068587231299631360.
Following just as easily was the calculation

[3] 34 digits in length

[4] 32 digits in length

d = 127831711759131502460439396608140279405356765399102985441866141840351.
We raised message *m* to the power of *d* and the result was
decrypt = 1121225152171520201541591920182520182511292020125205144518145 1919.
With a little trial and error, we were able to determine the unencrypted message of our Evil Enemy: ALL YOU GOT TO DO IS TRY TRY A LITTLE TENDERNESS.
Evil indeed!

Afterword: Notes on Optimizations

There are far more optimizations of the quadratic sieve that we did not implement. We shall briefly review some of the most common ones in the context of our own efforts to program a quadratic sieve. A notable difference is our use of division to determine which polynomials factor completely over the factor base. In practice, it is far better to utilize natural logarithms to estimate factorizations. As observed by Andrew Granville West (not to be confused with Andrew Granville),
"If Z = z · z' then log(Z) - log(z) - log(z') = 0" [16]. Instead of storing large values of the polynomials Q(*i*), the computer is now coping with the much smaller log(Q(*i*)) values. West describes subtracting logarithms at the appropriate locations and flagging values below a threshold as possible relations. Other variations on this method report solely the values that are zero (missing certain *B*-smooth numbers for the same reason we were losing numbers when our division threshold was set to 1), and others choose to initialize at 0 and add values of log*p* and report those locations that are greater than a predetermined cutoff [8][13]. Despite the advantages, our team did not use logarithms to make our quadratic sieve more efficient. We weighed our options against our time constraints and ultimately, anxious to have a parallelized algorithm up and running, we decided to implement our division-based algorithm rather than pause to write new code. For this same reason, we also did not premultiply *c* by a small square-free integer[5], "with the purpose to bias the factor base towards the smaller primes" [13]. Once we began to implement our quadratic sieve on *c*, we were reluctant to run the code again on some k · *c*. As soon as we started finding relations, it was full steam ahead and there was no going back. Fortunately for us, we were successful without these optimizations.

References

[1] K. Boklan, "How I broke the confederate code (137 years too late)," *Cryptologia*, 30:4 (2006) 340-345.

[2] D. Boneh, "Twenty years of attacks on the RSA cryptosystem," *Notices of the American Mathematical Society* 46:2 (1999) 203-213.

[3] J. Buhler, S. Wagon, "Basic algorithms in number theory," *Surveys in algorithmic number theory*, edited by J. P. Buhler and P. Stevenhagen, Math. Sci. Res. Inst. Publ. 44, Cambridge University Press, New York (2008) 25-68.

[5]Not divisible by the square of any prime [6]

[4] Dante. *Inferno*. Trans. John Ciardi. New York: Signet Classic, 2001.

[5] A. Granville, “Smooth numbers: computational number theory and beyond,” *Surveys in algorithmic number theory*, edited by J. P. Buhler and P. Stevenhagen, Math. Sci. Res. Inst. Publ. 44, Cambridge University Press, New York (2008) 276-323.

[6] G. H. Hardy and E. M. Wright, *An Introduction to the Theory of Numbers*. Oxford Clarendon Press (2008) Sixth edition.

[7] C. Liu, Z. Ye, “Estimating the phi(n) of upper/lower bound in its RSA cryptosystem,” Date accessed 5 April 2014 https://eprint.iacr.org/2012/666.pdf

[8] C. Pomerance, “The quadratic sieve factoring algorithm,” *Advances in Cryptography* (T. Beth, N. Cot, and I. Ingemarrson, eds.), Lecture Notes in Comput. Sci., vol 209, Springer-Verlag, Berlin and New York (1985) 169-182.

[9] C. Pomerance, “A tale of two sieves,” *Notices of the AMS* vol 43:12 (1996) 1473-1485.

[10] C. Pomerance, “On the distribution of the values of Euler’s function,” *Acta Arith*. 47 (1986) 63-70.

[11] C. Pomerance, “The role of smooth numbers in number theoretic algorithms,” *Proc. International Cong. of Mathematicians*, Zurich (1994) 411-422.

[12] C. Pomerance, “Smooth numbers and the quadratic sieve,” *Surveys in algorithmic number theory*, edited by J. P. Buhler and P. Stevenhagen, Math. Sci. Res. Inst. Publ. 44, Cambridge University Press, New York (2008) 69-81.

[13] H. te Riele, W. Lioen, and D. Winter, “Factoring with the quadratic sieve on large vector computers,” *J. Comp Appl. Math.,* 27 (1989) 267-278.

[14] R. D. Silverman, “The multiple polynomial quadratic sieve,” *Math. Comp.* 48 (1987) 329-339.

[15] R. Séroul, "The Sieve of Eratosthenes," *Programming for Mathematicians* 8.6 Berlin: Springer-Verlag (2000) 169-175.

[16] A. G. West, “Bound Optimization for parallel quadratic sieving using large prime variations,” *Washington and Lee University* Honors Thesis (2007) 1-64.