Projet INF442 #8 (**)
Euclid vs RSA: Cryptanalysis with batch GCDs

Benjamin Smith
smith@lix.polytechnique.fr

X2014
April 1, 2016

Abstract

RSA is the most widespread public-key cryptosystem on the Internet. Suppose we are given a massive collection of 1024-bit RSA keys (for example: every such public key available on the internet). Each of these keys contains a product $N = pq$ of two 512-bit random-looking primes, and the security of the cryptosystem depends on the difficulty of factoring N (that is, deriving p and q given only N). Factoring any *one* N is a seriously hard problem. But in practice, if we start with a large enough set of keys, then we can often factor *some* of them. In an ideal world, no two keys share a p or a q. But in the real world, many keys are generated using poorly-configured (or compromised) random number generators—which means that occasionally the same prime will pop up in two different keys, and then we can easily find that common prime as the greatest common divisor (GCD) of the two keys, using the classic Euclidean algorithm. Computing all of the GCDs pairwise is slow (the difficulty increases quadratically with the number of keys), but we can do much better using a "batch GCD" algorithm. The aim of this project is to implement an efficient distributed batch GCD algorithm, and apply it to collections of millions of RSA keys. The sequential version of this algorithm mirrors a real-world attack from 2012 (Heninger et al.), which broke tens of thousands of deployed RSA keys. While the algorithm appears simple enough, creating a distributed version involves several subtle choices...

1 Introduction

RSA is the most widespread public-key cryptosystem on the Internet. Each user (ie, each client and each server) has a public key (N, e) and a private key d, such that

- $N = pq$ is the product of two primes p and q, of about the same size, such that[1] $\gcd(p-1, q-1) = 2$;
- e and d are integers in $[2..N-2]$, neither divisible by p or q, such that $ed \equiv 1 \pmod{\lambda(N)}$, where[2] $\lambda(N) = (p-1)(q-1)/2$.

Recall that if x and y are integers, then $g = \gcd(x, y)$ is the greatest common divisor of x and y: that is, the largest positive integer such that $x \equiv y \equiv 0 \pmod{g}$.

[1] Since p and q are odd, both $p-1$ and $q-1$ are always even; the condition $\gcd(p-1, q-1) = 2$ means that $p-1$ and $q-1$ must not share any other common factors apart from 2.

[2] $\lambda(N)$ is the *exponent* of the multiplicative group of invertible residues modulo N; that is, $\lambda(N)$ is the smallest positive integer such that $x^{\lambda(N)} \equiv 1 \pmod{N}$ for all x such that $\gcd(x, N) = 1$. In general, if $N = pq$ is the product of two primes, then $\lambda(N) = (p-1)(q-1)/\gcd(p-1, q-1)$; our previous condition yields the formula for $\lambda(N)$ above.

Naïve RSA encryption and decryption Let m be a message, encoded as an element of $(\mathbb{Z}/N\mathbb{Z})^\times$. To send m to the user with public key (N,e), we compute the encrypted message $x = m^e \pmod N$ and send x. To decrypt x, the user computes $x^d \pmod N$ using their private key d; then

$$x^d \equiv (m^e)^d \equiv m^{ed} \equiv m \pmod N ,$$

because $ed \equiv 1 \pmod{\lambda(N)}$ (and so $x^{\lambda(N)} \equiv 1 \pmod N$).

The secret is safe so long as d remains hidden. Now, we can easily compute d if we know $\lambda(N)$, which amounts to knowing p and q. The security of the cryptosystem therefore relies on an attacker being unable to compute the prime factors p and q of N.

The most popular class of RSA keys involve 1024-bit moduli: that is, N is a 1024-bit integer, and p and q are each 512-bit integers. (In practice, RSA is more useful for digital signatures than for encryption, but the signature algorithm is slightly more complicated, and we will not go into detail on it here.)

Factoring $N = pq$ How can we compute p and q? A naïve attacker might try guessing one of the secret primes and checking whether it divides the user's public key (testing divisibility is cheap), but this is extremely unlikely to yield a factor: there are more than 2^{500} 512-bit primes, which is far beyond contemporary computing resources. Indeed, this is far beyond the number of atoms in the universe!

The state of the art factoring algorithm for RSA numbers is the general number field sieve (NFS). It is a subexponential algorithm: asymptotically faster than any exponential algorithm, such as the "trial division" approach above, but asymptotically slower than any polynomial algorithm. The *public*[3] record for NFS factorization is a 768-bit RSA number, factored by a group of researchers from EPFL, NTT, Bonn, LORIA, Microsoft, and CWI in December 2009. This took the equivalent of about 2000 core-years on a 2.2GHz machine, and about two years in wall-clock time. Factoring a 1024-bit RSA number has been estimated at being about 1000 times harder. The record using publicly-available software is a 704-bit RSA modulus, factored using the CADO-NFS software in July 2012.

While factoring a single 1024-bit RSA modulus is therefore way beyond what we can reasonably achieve in this class, it is much easier to find a common factor of two distinct RSA moduli, if such a factor exists. Indeed, we can quickly compute the gcd of N_1 and N_2; and if $N_1 \neq N_2$ and $\gcd(N_1,N_2) \neq 1$, then $\gcd(N_1,N_2)$ must be a factor of both N_1 and N_2. As we will see, this means that sometimes attacking *many moduli simultaneously* can be successful, even with modest time and computing resources.

The Batch GCD attack To generate an RSA modulus N, we generate a pair of *random* primes p and q satisfying a few technical conditions (such as $\gcd(p-1,q-1) = 2$), and then multiply them together to get N. So now suppose we have a huge set

$$\{N_0 = p_0q_0, N_1 = p_1q_1, N_2 = p_2q_2, \ldots, N_k = p_kq_k\}$$

of real-world RSA moduli. All of the N_i will have been generated using the method above, using random primes. The key word here is *random*: in the real world, there is a good chance that some people have used sources of randomness that are weak, badly configured, or fatally compromised. In particular, *some primes may appear more than once*!

This is very bad news for internet security, but good news for us, because we can compute these shared primes efficiently, and break the corresponding keys. The reason we can do this is that while factoring one number is hard, detecting common factors between any pair of numbers is easy.

[3] We do not know what size numbers can be factored by government agencies.

Abstractly, we want to compute $\gcd(N_i, N_j)$ for every pair of moduli in $\mathscr{S}$. While each gcd is easy enough to compute, this is a quadratic number of gcds, which is suboptimal. To do better, we use a *batch GCD* algorithm.

2 Batch GCDs

Let $N_0, \ldots, N_{k-1}$ be a sequence of k positive integers, and let

$$M := \prod_{i=0}^{k-1} N_i$$

be their product. The batch-GCD of the sequence is the sequence

$$BGCD((N_0, \ldots, N_k)) := (G_0, \ldots, G_k) \quad \text{where each} \quad G_i := \gcd(N_i, M/N_j) \ .$$

Note that $BGCD(N_1, N_2) = \gcd(N_1, N_2)$. To compute the batch GCD, we proceed as follows:

1. Compute the product $M = \prod_{i=0}^{k} N_i$ using a product tree;
2. Compute the intermediate sequence of remainders $R_0, \ldots, R_k$ such that each $R_i \equiv M \bmod N_i^2$, using a remainder tree;
3. Compute the results using $G_i = \gcd(R_i, R_i/N_i)$. There are three possibilities for any given G_i:
   - $G_i = 1$: in this case N_i has no common factors with any N_j, $j \neq i$.
   - $1 < G_i < N_i$: in this case G_i is a proper divisor of N_i, which means it is either p_i or q_i. We have found a factor of N_i, and broken the key!
   - $G_i = N_i$: in this case both of the prime factors p_i and q_i of N_i divide other moduli in the set. This could mean that N_i is repeated (in which case we have not broken N_i); otherwise, p_i divides some N_{j_1} and q_i divides some N_{j_2}; with a bit of careful matching, we should be able to break those keys.

3 The project

You will be given a series of datasets; each is a file containing a large number of RSA moduli of a given size. Your task is to break as many keys as you can, as quickly as you can.

Task 0 Prove that the batch GCD algorithm is correct! Extend it to derive an algorithm which, given a list of k RSA moduli $(N_0, N_1, \ldots, N_k)$, returns a list of known factorizations: that is, a list $((i, p_i, q_i))_{i=0}^{r}$ such that $N_i = p_i q_i$. Estimate its total running time in terms of the number of moduli k and the cost $M(n)$ of multiplying two n-bit integers.

Task 1 Implement the sequential batch GCD algorithm above, and use it to attack the datasets. Describe the performance of your implementation. Since we are working with integers far larger than the native 64-bit integers of our machines, you will need to use a big-integer package such as GMP or NTL.

Task 2 The sequential attack requires a lot of memory, and plenty of cycles. Design and implement a distributed batch GCD attack. How can you distribute the workload in the product and remainder trees, and the final detection of factors? How does your choice of topology impact on the algorithm?