Algorithm Soup

A bit of math and a byte of computer science

The (Almost) Secret Algorithm Researchers Used to Break Thousands of RSA Keys

RSA encryption allows for anyone to send me messages that only I can decode. To set this up, I select two large random primes p and q (each of which is hundreds of bits long), and release their product $x = p \cdot q$ online for everyone to see; x is known as my *public key*. In addition, I pick some number e which shares no factors with $p - 1$ or $q - 1$ and release it online as well.

The beauty of RSA encryption is that using only the information I publicly released, anyone can encode a message they want to send me. But without knowing the values of p and q, nobody but me can decode the message. And even though everyone knows my public key $x = p \cdot q$, that doesn't give them any efficient way to find values for p or q. In fact, even factoring a 232-digit number took a group of researchers more than 1,500 years of computing time (distributed among hundreds of computers).

On the surface, RSA encryption seems uncrackable. And it might be too, except for one small problem. Almost everyone uses the same random-prime-number generators.

A few years ago, this gave researchers an idea. Suppose Bob and Alice both post public keys online. But since they both used the same program to generate random prime numbers, there's a higher-than-random chance that their public keys share a prime factor. Factoring Bob's or Alice's public keys individually would be nearly impossible. But finding any common factors between them is

much easier. In fact, the time needed to compute the largest common divisor between two numbers is close to proportional to the number of digits in the two numbers. Once I identify the common prime factor between Bob's and Alice's keys, I can factor it out to obtain the prime factorization of both keys. In turn, I can decode any messages sent to either Bob or Alice.

Armed with this idea, the researchers scanned the web and collected 6.2 million actual public keys. They then computed the largest common divisor between pairs of keys, cracking a key whenever it shared a prime factor with any other key. All in all, they were able to break 12,934 keys. In other words, if used carelessly, RSA encryption provides less than 99.8% security.

At first glance this seems like the whole story. Reading through their paper more closely, however, reveals something strange. According to the authors, they were able to run the entire computation in a matter of hours on a *single* core. But a back-of-the-envelope calculation suggests that it should take *years* to compute GCD's between 36 trillion pairs of keys, not *hours*.

So how did they do it? The authors hint in a footnote that at the heart of their computation is an asymptotically fast algorithm, allowing them to bring the running time of the computation down to nearly linear; but the actual description of the algorithm is kept a secret from the reader, perhaps to guard against malicious use. Within just months of the paper's publication, though, follow-up papers had already discussed various approaches in detail, both presenting fast algorithms (see this paper and this paper), and even showing how to use GPUs to make the brute-force approach viable (see this paper).

There's probably a lesson here about not bragging about things if you want them to stay secret. But for this post I'm not interested in lessons. I'm interested in algorithms. And this one turns out to be both relatively simple and quite fun.

Algorithm Prerequisites: Our algorithm will deal with integers having an asymptotically large number of digits. Consequently, we cannot treat addition and multiplication as constant-time operations.

For n-bit integers, addition takes $O(n)$ time. Using long multiplication, multiplication would seem to take $O(n^2)$ time. However, it turns out there is an algorithm which runs in time $O(n \log n \log \log n)$.

Computing the GCD naively using the Euclidean algorithm would take $O(n^2 \log n \log\log n)$ time. Once again, however, researchers have found a better algorithm, running in time $O(n \log^2 n \log\log n)$.

Fortunately, all of these algorithms are already implemented for us in GMP, the C++ big-integer library. For the rest of the post we will use Big-O-Tilde notation, a variant of Big-O notation that ignores logarithmic factors. For example, while GCD-computation takes time $O(n \log^2 n \log\log n)$, in Big-O-Tilde notation we write that it takes time $\widetilde{\mathrm{O}}(n)$.

Transforming the Problem: Denote the set of public RSA keys by $k_1, \ldots, k_n$, where each key is the product of two large prime numbers (i.e., hundred digits). Note that n is the total number of keys. Rather than computing the GCD of each pair of keys, we can instead compute for each key k_i the GCD of it and the product of all the other keys $\prod_{t \neq i} k_t$. If a key k_i shares exactly one prime factor with other keys, then this provides that prime factor. If both prime factors of k_i are shared with other keys, however, then the computation will fail to actually extract the individual prime factors. This case is probably rare enough that it's not worth worrying about.

The Algorithm: The algorithm has a slightly unusual recursive structure in that the recursion occurs in the middle of the algorithm rather than at the end.

At the beginning of the algorithm, all we have is the keys,

$k_1,$
$k_2,$
$k_3, \cdots$

The first step of the algorithm is to pair off the keys and compute their products,

$j_1 = k_1 \cdot k_2,$
$j_2 = k_3 \cdot k_4,$
$j_3 = k_5 \cdot k_6, \cdots$

Next we recurse on the sequence of numbers $j_1, \ldots, j_{n/2}$, in order to compute

$r_1 = GCD(j_1, \prod_{t \neq 1} j_t),$

$r_2 = GCD(j_2, \prod_{t \neq 2} j_t)$,
$r_3 = GCD(j_3, \prod_{t \neq 3} j_t), \cdots$

Our goal is to compute $s_i = GCD(k_i, \prod_{t \neq i} k_t)$ for each key k_i. The key insight is that when i is odd, s_i can be expressed as
$s_i = GCD(k_i, r_{(i+1)/2} \cdot k_{i+1})$,
and that when i is even, s_i can be expressed as
$s_i = GCD(k_i, r_{i/2} \cdot k_{i-1})$.
To see why this is the case, one can verify that the term on the right side of the GCD is guaranteed to be a multiple of $GCD(k_i, \prod_{t \neq i} k_t)$, while also being a divisor of $\prod_{t \neq i} k_t$. This, in turn, implies that the GCD-computation will evaluate to exactly $GCD(k_i, \prod_{t \neq i} k_t)$, as desired.

Computing each of the s_i's in terms of the r_i's and k_i's completes the algorithm.

Bounding the Running Time: Let m denote the total number of bits needed to write down $k_1, k_2, \ldots, k_n$. Each time the algorithm recurses, the total number of bits in the input being recursed on is guaranteed to be no more than at the previous level of recursion; this is because the new inputs are products of pairs of elements from the old input.

Therefore each of the $O(\log n)$ levels of recursion act on an input of total size $O(m)$ bits. Moreover, the arithmetic operations within each level of recursion take total time at most $\tilde{O}(m)$. Thus the total running time of the algorithm is also $\tilde{O}(m)$ (since the $O(\log n)$ recursion levels can be absorbed into the Big-O-Tilde notation).

If we unwrap the running time into standard Big-O notation, we get $O(m \log^3 m \log \log m)$.

Is it practical? At first glance, the triple-logarithmic factor might seem like a deal breaker for this algorithm. It turns out the actual performance is pretty reasonable. This paper found that the algorithm takes time roughly 7.65 seconds per thousand keys, meaning it would take a little more than 13 hours to run on 6.2 million keys.

One of the log factors can be removed using a slightly more clever variant of the algorithm, which avoids GCD computations at all but the first level of recursion (See this paper). The improved algorithm takes about 4.5 seconds per thou-

sand keys, resulting in a total running time of about 7.5 hours to handle 6.2 million keys.

So there we go. A computation that should have taken years is reduced to a matter of hours. And all it took was a bit of clever recursion.

PUBLISHED BY

William Kuszmaul

I am an MIT Ph.D. student in the computer science department. My research focuses on algorithms and combinatorics. View all posts by William Kuszmaul →

January 15, 2019 William Kuszmaul

8 thoughts on “The (Almost) Secret Algorithm Researchers Used to Break Thousands of RSA Keys”

Mathieu Sabourin

January 15, 2019 at 11:14 pm

Very interesting article. Thank you

★ Like

Juan Pablo Daniel Borgna

January 16, 2019 at 3:09 am

Loved the article and now I want to try by myself. Thinking on how to collect a

few million of certs, group them by used random generator and then try to crack them like this.

★ Like

Arun Kumar

January 16, 2019 at 6:10 am

This is really well written, covering almost all prerequisite information for the topic. And the math is also well explained. Kudos!

★ Like

Jim

January 16, 2019 at 1:51 pm

But my question is how to PREVENT this from happening? How does one avoid having a a shared prime factor? Or can you?

★ Like

Patrik Lundin

January 17, 2019 at 4:13 am

Fully determining what must happen would likely take tracking down which keys fail, and what exactly made them share a prime factor. But I think a really solid guess is “Use a proper, well-seeded RNG”. Those little “shake your mouse” or “type some stuff’ or other “give me some physical events to ensure sufficient entropy”, do them. If you’re unsure where your random is coming from, look into it (hugely system dependent). Because, for sure, one way this would happen would be to run the same RNG (not much you can do there – like most crypto they’re generally well written and tested enough that any individual bright ideas do more damage at scale) and not have sufficient entropy to make them diverge evenly of the space. And it’s not particularly far fetched to think that one in five hundred-ish either used a generator that didn’t freeze up and be all “Whoa, human! This computer was just rebooted! /dev/urandom has a mere 64 bits of true entropy!” or used some other en-

tropy creator that full-on says it's not *sure* there's enough, and just said "Whatever – what's going to happen, you use a mere 32 bits of randomness instead of 128? In what possible way could someone leverage that into an attack?!". Well.. this whole episode kind of fleshed that out – what was a theoretical flaw (very clearly a flaw, but none the less) is now a practical exploit, albeit one that doesn't shred your encryption like tissue, but it does inject a 0.3%ish failure rate which is bad if you're expecting five nines.

★ Like

Jeremy Lakeman

January 17, 2019 at 5:26 am

From memory, the original paper suggested these common factors came from embedded devices with almost no sources of available entropy, rolling an ssh key on first boot. Perhaps made by the same manufacturer.

★ Like

Pingback: RSA Encryption Cracked Easily (Sometimes) | Hackaday

---

NSA

January 17, 2019 at 3:27 pm

yoink

★ Like
