RSA OPTIMIZATIONS AND TIMING ATTACKS ON RSA

JOSHIKA CHAKRAVERTY

Abstract. I will explain how a Timing Attack on RSA works. I will review what RSA is, and then describe five ways that computing systems save time. These timesavers are exponentiation by squaring, the Chinese Remainder Theorem, Sliding Windows, Montgomery Representation and Karatsuba Multiplication. Then I will outline a basic timing attack that takes advantage of these timesavers.

1. A Review of RSA encryption

RSA is a public key cryptosystem that depends on a trapdoor function, a function that is easy to compute in one direction and is extremely difficult in the other direction. RSA works because multiplying two large prime numbers takes time but it definitely doable, but factoring a multiple of two large primes is almost impossible to do in polynomial time. The two people who want send messages to each other are Alice and Bob and our eavesdropper is aptly named Eve.

Alice chooses two primes p and q and multiplies them into part of the public key N, where $N = pq$.

She computes the totient function of N, which is $\phi(N)$. Then Alice selects her public encryption exponent e so that it is $\gcd(e, \phi(n) = 1$ and $e \leq N$. Alice also

Alice also chooses a private decryption exponent d so that $de \equiv 1(\text{mod n})$.

Bob is trying to send the message m, so he sends a ciphertext c where $c \equiv m^e(\text{mod N})$. Alice decrypts c by calculating $m \equiv c^d(\text{mod n})$.

2. RSA Optimizations

RSA takes a long time to compute. The longer p and q are, the longer N becomes and the more time it takes to compute the other variable. A common size for p and q is 512 bits, and each product almost doubles in length. Luckily, Alice Bob have four different methods to optimize their multiplication.

2.1. **Exponentiation by Squaring.**

The standard method for fast exponentiation is the squaring and multiplying method. When trying to compute $c^d(\text{mod N})$ we first write N in binary so that

$$d = b_k ... b_0.$$

Then we can define a sequence $r_{k+1} ... r_0$ by setting $r_{k+1} = 1$ and then recursively defining

$$r_i^2 = r_{i+1}^2 x^{b_i}$$

for $i = k, ..., 0$ where$r_0 = x^d$.

What this function calculates is all the powers of two that needs to be computed, and multiplies lower powers together to find higher powers. To avoid saving and computations with long numbers, we reduce by $modN$ at every step. Overall, Alice or Bob have to perform the squaring approximately operation a total of $\log_2 d$ times and multiply half as many times. In big O notation, exponentiation by squaring should take $O((d \log c)^n)$.

Date: December 7, 2019.

2.2. **Chinese Remainder Theorem.**

Theorem 2.1. *When we have a number $N = pq$ where p and q are distinct primes, for any pair of positive integers x_1 and x_2 where $x_1 \leq p$ and $x_2 \leq q$, there is a unique number x which is less than N and $x_1 = x \pmod p$ and $x_2 = x \pmod q$*

The Chinese Remainder Theorem helps compute $m = c^d$(mod n) in a faster way. Recall that $p \leq q$ and e is an integer such that $xe \equiv 1$(mod N). first compute and store the following values:

$$dp = e^{-1}(modp_0 1)$$

$$\text{dq} = e^{-1} \text{ mod (q-1)}$$

and

$$q^{-1} = q^{-1} \text{ (mod } p)$$

.

In the next step, figure out what m is by splitting m into two parts: m_1 and m_2.

$$\text{let } m_1 = c^{dp}\text{(mod p)}$$

and

$$\text{let } m_2 = c^{dq} \text{ (mod q)}$$

. Calculate $h = q^{-1} \cdot (m_1 - m_2)$ (mod p) Finally, calculate $m = m_2 + m_1$.

2.3. **Sliding Windows.** Sliding Windows is another way to raise numbers to a high power in shorter amount of time. In this section we will be computing $y = x^E$.

For this algorithm to work, we need to write the exponent E in binary form just like in exponentiation by squaring. The binary expansion of d would look like

$$E = (d_{n-1}d_{n-2}...d_1d_0)$$

Then, partition d into F_i *words* or windows of length $L(F_i)$. The F_i's are just parts of the binary representation of E. The total number of windows is represented is k, so

$$i = 0, 1, ..., k-1.$$

The nice thins about windows it that they all do not have to be the same size, and it is okay is some of the windows have to length at all. However, the windows that are of nonzero length must have a significant digit, or leftmost digit equal to one because a window with a significant digit of one is thought to have a length of zero.

$$d = L(F_i)_{max}$$

Which means that d is the length of the longest window. Calculate and store x^ω where

$$\omega = 1, 2, 3...2^d - 1.$$

After storing all of the x^ω's, calculate

$$y_{k-1} = x^{F_{k-1}}$$

For every i from $k-2$ to 0 , calculate

$$y_{i-1} = y_i^{2^{L(F_i)}}$$

until $i = 0$.

If $F_i \neq 0$, then $y = y * x^{F_i}$. This algorithm will then return $y = x^d$.

2.4. **Karatsuba's Algorithm.** Karatsuba's Algorithm multiplies two large numbers together by splitting the number up into three smaller products. Say that we are trying to compute $P = N_0 \cdot N_1$. Break each of the factors into parts in terms of integers a_0, a_1, b_0 and b_1.

$$N_0 = a_0 + a_1\omega$$
and
$$N_1 = b_0 + b_1\omega$$
so that $a_0 \leq \omega$ and $b_0 \leq \omega$ for a random number ω.

Because of general multiplication rules:

$$N_0 \cdot N_1 = a_0b_0 + (a_0b_1 + a_1b_0)\omega + a_1b_1\omega^2.$$

This looks like we need four multiplications but we actually only need two because

$$a_0b_1 + a_1b_0 = (a_0 + a_1)(b_0 + b_1) - a_0b_0 - a_1b_1.$$

a_0b_0 and a_1b_1 already have to be calculated for $N_0 \cdot N_1$ so we only have to do two multiplications. Addition is much faster to do, so the extra addition does not significantly increase the time.

2.5. **Montogomery Representation.** Montgomery Representation is a way to save time during modular multiplication. This is used not only used by RSA but is also implemented in Diffie-Helman and ElGamal.

Definition 2.2. Montgomery Domain In modulus N let x be an integer in the ring $\mathbb{Z}/n\mathbb{Z}$. Let $R = 2^k$ where k is now the number of bits in n. Move x into the ring by calculating $x' = xR(\text{mod N})$. To regain the original value of x, divide by R. N has to be an odd number so that it is relatively prime to R and R can divide.

Regular multiplication with factors a and b would lead to a product of $ab(\text{mod N})$. The Montogomery Representations of a and b are a' and b' respectively.

$$a' \cdot b' = aR \cdot bR = ab \cdot R^2(modN).$$

A single division of that product by R would lead to abR (mod N). Lets let cab.

To easily divide R, there is an algorithm called the Montgomery Reduction. If the binary representation of ab has k zeroes on the least significant side, $R = 2^k$ can be found by shifting right by k positions. If there are ones, then we have to add some number t to ab to make the least significant sides add up to zero. Here are three conditions that t needs to have:

(1) t must be a multiple of n
(2) the addition of t cannot change the value of $c(modN)$ (which it won't if t is a multiple of n).
(3) $c + t = 0(\text{mod R})$; t is the opposite of c (mod R).

By the first and third conditions,

$$ab + nt = 0(\text{mod R}) \text{ so } t = -ab/n(\text{mod R}).$$

Now we can find another value $n_i = -1/n(modR)$ so $t_i = -abn_i(modR)$.

The Montogomery Reduction Algorithm works in three steps:

(1) $t_i = cn_i$
(2) $ab = ab + t_i * n$
(3) $ab = ab/R$

If c ends up being larger than n, subtract n from the c. This extra step ensures that the product is in the range $[0, n)$.

Montgomery Multiplication saves times because three integer multiplications with numbers comparable length takes about as much time as one modular multiplication with numbers of this length. Werner Schindler realized that extra reductions are more likely when the product is closer to R. He proved that the probability is:

$$\Pr[\text{extra reduction}] = ab \text{ (mod R)}/2R.$$

This means that as the product ab approaches either of the factors of R or N from the left, the probability of a reduction increases and the time will be longer. On the other hand, if the product is a little over one of the factors, the probability of a reduction greatly decreases.

3. Attack on Montgomery Reduction and Karatsuba's Algorithm

This attack only works when Alice and Bob are using the Chinese Remainder Theorem to save time. One server that uses the Chinese Remainder Theorem is OpenSSL, which is what David Brumley and Dan Boneh's paper is mostly concerned with.
In this method, Eve is trying to decrypt some ciphertext g by using the tricks above to approximate q, where q is the smaller prime factor of N.
RSA as implemented by OpenSSL does two important things that are exposed through this attack

(1) RSA utilizes Montgomery Reductions, so a time difference due to the extra reduction is noticable.
(2) RSA regularly uses Karatsuba's Algorithm to multiply numbers that are approximately the same size and uses regular multiplication for numbers that are different lengths. Karatsuba is faster that regular multiplication so we can measure the time to find big descrepancies in timing.

3.1. Comparing Times.

In this timing attack, we are decrypting a ciphertext g. From the section on Montogomery Reduction, recall that if g is below a multiple of p or q, one of the factors N then the probability of an extra reduction is higher but if g is just above a multiple of p or q then the number of extra reductions becomes lower. Since the exact factor doesn't matter, I will only refer to factor q. With Karatsuba, when g is just above a multiple of q, when $g(modq)$ is very small so it will probably be multiplied to a much larger integer. Then RSA will utilize a slower normal multiplication. On the flip side, if g is just below a multiple of q, then $g(modq)$ will be large and since it is more likely to be multiplied to another large number, a faster Karatsuba's Algorithm will be used. To sum, a ciphertext g that is just a little lower than a multiple of q will take more less using Karatsuba's Algorithm but more time with Montgomery Reductions.

3.2. Outline of the Attack.

Let $N = pq$ with $q \leq p$. We will try to guess bits of q one at a time, from the most significant bit, which refers to larger powers, to the least significant bits, which refers to the bits closer to one. After we find most of the significant bits, there is a lattice based reduction algorithm to find the rest of the factor called **Coppersmith's Method**. Normally, the guess g of q starts with a number between 2^{512} and 2^{511} . Then we time all the possible combos for the first 2-3 most significant bits and plot time versus value of the bits. The very first peak is the reveals the value of q, since the time gets slower right up to q and then drops when it becomes lower than q. Say that we have the top $i-1$ bits, and set g to now be an integer that has the same top $i-1$ bits as q but the rest of the bits are 0. Then we can find the ith bit in the following way:

(1) let g_{hi} be the same as g except make the ith bit of g_{hi} one. If the ith bit of q is actually one, then $g \leq g_{hi} \leq q$. But if the ith bit of q is zero, then $g \leq q \leq g_{hi}$.
(2) Find $u_g = gR^{-1}modN$ and $u_{g_{hi}} = g_{hi}R^{-1}$. This step is just to reverse the Montogomery Reduction since RSA will multiply both of those numbers by R.
(3) Measure the time it takes to decrypt u_g and $u_{g_{hi}}$. Set t_1 be the time it takes to decrypt u_g and let t_2 be the amount of time it takes to decrypt $u_{g_{hi}}$.
(4) Find the difference between t_1 and t_2. If the ith bit of q is 0, then this difference will be large and if the ith bit of q is 1 then the timing difference will be small. Large and small are subjective terms, but sending several random values of g and observing timing values will give a good sense of what large and small mean in this context.

4. Why should we care about timing attacks?

Timing attacks are extremely tedious and depend on several lucky factors, like actually being able to time how long decryption takes and having access put carefully selected keys through this system. If timing attacks were actually successful against RSA, then RSA would not be the most widely used public key cryptosystem. Billions of computers use RSA to secretly pass information. It is important to keep delving into timing attacks against RSA to potentially find weaknesses in the system and to better understand the overlooked time saving techniques.

References

[BB05] David Brumley and Dan Boneh. Remote timing attacks are practical. *Computer Networks*, 48(5):701–716, 2005.

[Ber] G. Bertroni. Montgomery multiplication. *Foundations of Cryptography*.

[Koç95] Cetin K Koç. Analysis of sliding window techniques for exponentiation. *Computers & Mathematics with Applications*, 30(10):17–24, 1995.

[Koc96] Paul C Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *Annual International Cryptology Conference*, pages 104–113. Springer, 1996.

[vC09] Mark van Cuijk. Timing attacks on rsa, 2009.

[Won] Wing H Wong. Timing attacks on rsa: revealing your secrets through the fourth dimension.

[Zel17] Nickolai Zeldovich. 16. Side-Channel Attacks. `https://www.youtube.com/watch?v=3v5Von-oNUg`, March 2017.