Partial Key Exposure Attack On Low-Exponent RSA

Eric W. Everstine

1 Introduction

Let $N = pq$ be an RSA modulus with e, d encryption exponents such that $ed \equiv 1 \bmod \phi(N)$. Then, for small public exponent e, it is possible to recover the entire private exponent d, and therefore factor N, given the $n/4$ least significant bits of d, where n is the number of bits of N. This attack is called a *partial key exposure attack*.

1.1 Why is this useful?

While quite interesting mathematically, this attack on RSA may at first glance appear to have very limited use in the real world. Is it really feasible that one can somehow obtain only the $n/4$ least significant bits of d and therefore utilize the attack?

The answer is actually yes. There are a variety of attacks on RSA; some of which are called timing attacks. Timing attacks track the amount of time a computer spends computing various steps of a given cryptographic protocol and information can be gleaned from this [1]. These attacks take various amounts of time, but assuming one can be utilized to expose the $n/4$ LSB of d in and RSA protocol, this partial key exposure attack can then be used to efficiently recover the rest of d.

1.2 Review of the RSA algorithm[6]

The RSA algorithm (named after its inventors, Rivest, Shamir and Adleman) is a mathematically based public-key cryptosystem that is dependent on the assumption that factoring is "hard". And hard it is, as even the most sophisticated factoring algorithms, given a large enough integer N, would take such an inordinate amount of time as to render it useless.

And thus RSA uses such an integer N, where $N = pq$, p and q large primes. It then makes good use of Euler's Theorem to choose the encryption exponent e and decryption exponent d:

Theorem 1 (Euler[5]). If m is a positive integer and a is an integer with GCD(a, m) = 1, then $a^{\phi(m)} \equiv 1 \bmod (m)$.

Here, GCD(a,m) is the Greatest Common Divisor of a and m and $\phi(m)$ is Euler's phi function, defined to be the number of positive integers less than m that are relatively prime to m. Two numbers are relatively prime if their GCD is 1. And, most importantly, if one can factor m, one can quickly calculate $\phi(m)$.

For a simple example, take $m = 15$ and $a = 2$. Then $\phi(m) = 8$. We see that

$$2^8 = 256 \equiv 1 \bmod (15)$$

as 15 divides 255 evenly.

To show how RSA makes use of this, assume your friend Bob wants to send you a message M without the devious Eve intercepting and reading it. Also assume M has been converted by one of a variety of ways into a string of numbers. For simplicity, we will bunch the numbers of M together to form one large number (although one can separate it into any number of blocks and encode those blocks independently). For RSA to work properly, we require $M < N$.

First, you choose your N, the product of large primes p and q, and then the exponents e and d such that $ed \equiv 1 \bmod \phi(N)$. You make e and N public, but keep d private. Bob computes $M_2 \equiv M^e \bmod (N)$ and sends it over public channel to you. You compute $M_2^d = (M^e)^d \equiv M^{ed} \bmod (N)$. But, $ed \equiv 1 \bmod \phi(N)$ so

$$M^{ed} \equiv M^{1 \bmod \phi(N)} = M^{0 \bmod \phi(N)} * M \equiv M^{\phi(N)} * M \equiv M \bmod (N) = M,$$

since $M < N$. And so you have just uncovered M using the private exponent d. Furthermore, even if Eve intercepts the transmission (which one assumes she will), in order to reveal M she would either have to be able to compute the e^{th} root of $M_2 \bmod (N)$ or be able to factor N in order to quickly calculate $\phi(N)$, thus exposing d. No other methods are known for determining M, but both of the methods discussed and even methods as of yet unknown are thought to be hard.

1.3 Notation and Definitions

The following list summarizes the definitions and assumptions used in this paper.

$N = pq$ is the n-bit RSA modulus, using large primes p and q, where we assume p and q satisfy

$$N^{1/2} / 2 < q < p < 2 N^{1/2}.$$

For convenience in future equations, we define $s = p + q$.
Lastly, define k to be the unique integer such that

$$ed - k\phi(N) = ed - k(N - s + 1) = 1.$$

That is, since $ed \equiv 1 \bmod \phi(N)$, 1 can be written as a linear combination of ed and $\phi(N)$. k is simply the number one must multiply $\phi(N)$ to get this linear combination. Since $\phi(N) > d$ we know that $k < e$.

2 Supporting Theorems

Before presenting the main theorem, it is important to first present theorems on which the main one depends. The first one shows how one can efficiently find small solutions (x_0, y_0) to a bivariate polynomial given appropriate, known bounds on x_0 and y_0 in advance.

Theorem 2 (Coppersmith[3]). Let $f(x, y)$ be a polynomial in two variables over the integers, Z, of maximum degree δ in each variable separately, and assume the coefficients of f are relatively prime as a set. Let X, Y be bounds on the desired solutions x_0, y_0. Define $f^*(x, y) := f(Xx, Yy)$ and let D be the absolute value of the largest coefficient of f^*. If $XY < D^{2/(3\delta)}$, then in time polynomial in $(\log D, 2^{\delta})$, we can find all integer pairs (x_0,y_0) with $p(x_0, y_0) = 0$, $|x_0| < X$, $|y_0| < Y$.

A proof of this will not be provided here, but it uses lattice basis reduction, and is not fun. Actually Theorem 2 all together isn't very fun, but it does provide the framework to produce a corollary that is immensely helpful to our cause (although it may or may not be very fun, either):

Corollary 1 [2]. Let $N = pq$ be an n-bit RSA modulus. Let $r \geq 2^{n/4}$ be given and suppose $p_0 := p \bmod r$ is known. Then it is possible to factor N in time polynomial in n.

Proof. Since we know p_0 and that it is $p \bmod r$, then we can find $q_0 := q \equiv N/p_0 \bmod r$. Therefore, we can create the polynomial

$$f(x,y) = (rx + p_0)(ry + q_0) - N. \qquad (1)$$

The solution (x_0, y_0) to this (with $0 < x_0 < X = 2^{n/2+1}/r$ and $0 < y_0 < Y = 2^{n/2+1}/r$) will reveal the factorization of N. These bounds hold since by definition $N < 2^n$ (since it is an n-bit number), and at $x = X$, $y = Y$, and assuming $r = 2^{n/4}$ (its lower bound) $(rx)(ry)$ would yield

$$(2^{n/4}2^{n/2+1}/2^{n/4})(2^{n/4}2^{n/2+1}/2^{n/4}) = 2^{n+2} > N.$$

Therefore the solutions must be strictly less than X and Y. So now we have a bivariate polynomial to which we seek a solution; the exact situation Theorem 2 handles. The GCD of the coefficients of $f(x,y)$ is r, so in order to use Theorem 2 (since the coefficients must be relatively prime) we must divide through by r to get $g(x,y) = f(x,y)/r$. The largest coefficient of $g(x,y)$ is somewhat small, so we define a new function $g^*(x,y) = g(Xx,Yy)$ which now has a largest coefficient that is at least $2^{n+2}/r$. Now, in order to use Theorem 2, it is required that

$$XY = 2^{n+2}/r^2 < (2^{n+2}/r)^{2/3},$$

and by solving algebraically, we see that this is satisfied whenever $r > 2^{(n+2)/4}$. Finally, by doing an exhaustive search on the first two bits of x_0 and y_0, this can be reduced to $r \geq 2^{n/4}$. □

Now that the framework is in place, we can put it to some good use.

3 Main Theorem

In order to efficiently implement the partial key exposure attack, we need one more requirement—that the encryption exponent, e, is sufficiently small. There is no set limit on how large e can be, but it must be small enough such that with available computing power, it is possible to do an exhaustive search on all values less than it.

Theorem 3[2]. Let $N = pq$ be an n-bit RSA modulus. Let $1 \leq e, d \leq \phi(N)$ satisfy $ed \equiv 1 \bmod \phi(N)$. There is an algorithm that given the $n/4$ least significant bits of d computes all of d in polynomial time in n and e.

Proof. Since we are given the $n/4$ least significant bits of d, call it d_0, we know $d \equiv d_0 \bmod 2^{n/4}$. But remember that we have defined a value k such that

$$ed - k\phi(N) = ed - k(N - s + 1) = 1,$$

where $s = p + q$. Therefore $ed = 1 + k\phi(N) = 1 + k(N - s + 1)$, and we now have

$$ed_0 \equiv 1 + k(N - s + 1) \pmod{2^{n/4}} \qquad (2)$$

So now what one can do to carry out the attack is first try all candidate values of k, which, since $k < e$, is all values on the range $[0 \ldots e]$. For each of these values, solve the above equation to obtain $s \bmod 2^{n/4}$. We can now use this value in the equation

$$p^2 - sp + N \equiv 0 \pmod{2^{n/4}}. \quad (3)$$

Notice what the left side of this equation reduces to:

$$p^2 - sp + N = p^2 - (p+q)p + N = p^2 - p^2 - qp + N = p^2 - p^2 - N + N = 0.$$

But since we are dealing with values d_0 and s that are mod $2^{n/4}$, our answer of 0 is also mod $2^{n/4}$. We solve equation (3) to obtain a candidate value for p mod $2^{n/4}$. With this candidate value, we can find a candidate $q_0 \equiv N/p_0$ and solve equation (1) quickly via the algorithm discussed in Theorem 2. □

3.1 Running Time Analysis

One sees that the running time of this algorithm is most dependent on the first and last steps. The first step requires stepping through up to e values of k. For each of these values, it is simply solving a couple modular equations, before the final step. The final step is actually factoring N, and this step is polynomial[1] in log D where D is the largest coefficient of the polynomial which is, in this case, equation (1). The largest coefficient is $XY = 2^{n+2}/r^2$ and so log $(XY) \approx n$ and thus the running time of this factoring step is n^{ε} for some ε. Therefore the running time is $O(en^{\varepsilon})$ which, for a fixed size RSA modulus and thus a fixed n, is simply linear in e.

3.2 A Numerical Example

Suppose you want to crack a message encoded with an RSA modulus N = 1633, and you know that e = 23. Also pretend you are unable to simply divide 1633 by all primes less than 40 (its approximate square root). 1633 in binary is 11001100001, which is 11 bits. Therefore, if by using timing attacks or some other method you are able to obtain the 3 least significant bits of d (if $n/4$ is a fraction, always round up), you could implement the above algorithm and expose both d and the factorization of N. Luckily, you are able to discover those 3 bits, and they are 011, or 3 in decimal. Let's see how it's done.

The first step is to look at equation (2):

$$ed_0 \equiv 1 + k(N - s + 1) \pmod{2^{n/4}}.$$

So in this problem we have:

$$(23)(3) \equiv 1 + k(1633 - s + 1) \pmod{2^3}$$

[1]When dealing with a largest coefficient that is n bits, the actual running time is roughly n^4[4]. This is a very rough estimate however, so the exact number is not emphasized.

So to work through the algebra and reduce mod 8:

$$69 \equiv 1 + k(1634 - s) \pmod 8$$

$$5 \equiv 1 + k(1634 - s) \pmod 8$$

$$4 \equiv k(1634 - s) \pmod 8 \qquad (4)$$

Now we begin to test candidate values of k. Since $e = 23$, there are only 23 possible values to test. For this example, we'll try two values: one to show what happens with a wrong value, and one that shows what happens when it's correct. First, the wrong value: let $k = 2$. We have:

$$4 \equiv 2(1634 - s) \pmod 8.$$

There is no inverse of 2 mod 8, but this equation can still be solved:

$$4 \equiv 3268 - 2s \pmod 8$$

$$2s \equiv 3264 \pmod 8$$

$$2s \equiv 0 \pmod 8$$

$$s \equiv 0 \pmod 8$$

Now we plug 3 into equation (3):

$$p^2 - sp + N \equiv 0 \pmod{2^{n/4}} \qquad (5)$$

$$p^2 - 0p + 1633 \equiv 0 \pmod 8$$

$$p^2 + 1633 \equiv 0 \pmod 8$$

$$p^2 \equiv -1633 \pmod 8$$

$$p^2 \equiv 7 \pmod 8$$

However, we only have to check 0 through 7 to find that 7 does not have a square root mod 8, and so the value fails. Were we working through the algorithm, we would move to the next number.

However, we would never actually get to $k = 2$ in this example because the correct value is $k = 1$. Picking up after equation (4) we get:

$$4 \equiv 1(1634 - s) \pmod 8$$

$$s \equiv 6 \pmod 8$$

and equation (5) now becomes

$$p^2 - 6p + 1633 \equiv 0 \pmod 8$$

$$p^2 - 6p \equiv 7 \pmod 8$$

$$p(p - 6) \equiv 7 \pmod 8$$

and by trying the 8 possible values for p, we find that $p \equiv 7 \bmod 8$. So now with this "candidate" value for $p_0 \equiv p \bmod 8$, we can determine a possible value for $q_0 \equiv q \bmod 8$. To do this, we need to recognize that $p_0 q_0 \equiv N \bmod 8$. Therefore:

$$7q_0 \equiv 1633 \pmod 8$$

$$7q_0 \equiv 1 \pmod 8$$

remember that the inverse of 7 mod 8 is 7, so multiplying both sides by 7 gives $q_0 \equiv 7 \bmod 8$. With these values for p_0 and q_0, we can now create the polynomial found in equation (1):

$$f(x,y) = (rx + p_0)(ry + q_0) - N$$

which becomes

$$f(x,y) = (8x + 7)(8y + 7) - 1633$$

the roots of which will yield the factorization of N. The algorithm from Theorem 2 can do this in time polynomial in n, but for small enough numbers like we are working with here, it is simple enough to try test values, and sure enough this equation has roots $x = 2$ and $y = 8$. This means that p and q are (8*2 + 7) = 23 and (8*8 + 7) = 71, respectively. Finally, now that p and q are exposed, it is easy enough to calculate $\phi(N) = (p - 1)(q - 1) = 22 * 70 = 1540$. To find d, we need to solve this equation:

$$ed - k\phi(N) = 1$$

Since $k = 1$, this becomes

$$23d - 1540 = 1$$

and the routine calculation shows $d = 67$. Now, any message sent using the given N and e as a key can easily be decoded.

3.3 Ok, That Was a LOT Harder Than Brute Force. What Gives?

The previous example certainly shows the algorithm works, but wouldn't it have just been quicker to try a brute force division of 1633 by all the primes less than its square root? In this example, yes it would have. However, in the real world, RSA moduli are huge, to the tune of 1024 bits. To show just how unwieldy a number of this size is, here's another theorem.

Theorem 4 (Prime Number Theorem[5]). The ratio of $\pi(x)$ to $x/(\ln x)$ approaches 1 as x grows without bound, where $\pi(x)$ is the number of prime numbers less than or equal to x.

In other words, the number of primes less than or equal to x is approximately equal to $x/(\ln x)$. This means that 2^{1024} has about $2^{1024}/(\ln 2^{1024}) \approx 2.5 \times 10^{305}$ primes less than it. So a computer that can fully check a trillion primes per *second* would still take about 8×10^{285} *years* to check them all. Even if we realize that we only have to check all primes up to the square root of N, this would still take 1.2×10^{144} years. Now compare that to the above algorithm whose running time we can approximate by en^4. Our super-computer(s) that can perform a trillion calculations per second would now take at most *e seconds* (give or take a relatively small constant multiplier).

3.4 Bringing It All Together

This algorithm far from renders RSA useless, however. Remember the running time is ce for some constant c. For an encryption exponent that is very small, like in our toy example above, the algorithm runs very fast even if the modulus N is large. Even for moderate e, the algorithm runs fairly efficiently: take $e \approx 2^{16}$ and $N \approx 2^{1024}$ and our supercomputer would still manage in roughly 18 hours, worst case (remember that for an N that large, brute force factoring would take the same insane number of years regardless of the size of e). However, for very large e, say of the magnitude of 2^{512}, the algorithm has essentially gained nothing: with a large N as in the example in the previous section, our super-computer would take somewhere along the line of 4.3×10^{146} years; that is, it has actually lost ground on brute-force division by primes. Therefore the algorithm is rendered useless (when compared to brute-force) by extremely high e or extremely low N (of course RSA itself is useless for small N).

4 Conclusion

Just like numerous other methods for "cracking" RSA, this algorithm is amazingly effective under certain circumstances, but rendered impotent

with relative ease. All one has to do is ensure proper bounds are placed on *e*, i.e. that is *e* is made large, and this attack will fail. So until factoring is made easy, or some other "back door" is found, all this method accomplishes is to tell us how to better implement RSA. For now at least, RSA is completely safe.

References

1. Boneh. "Twenty Years of Attacks on the RSA Cryptosystem". *American Mathematics Society Notices* 46. 1999. pp. 203-212.

2. Boneh, Durfee, Frankel. "An Attack on RSA Given a Small Fraction of the Private Key Bits". *AsiaCrypt '98*. pp. 25 – 34.

3. D. Coppersmith. "Finding a small root of a univariate modular equation", *Proc. Of Eurocrypt '96*. pp. 155 – 165.

4. Gathen and Gerhard. *Modern Computing Algebra*. Cambridge University Press, 1999. pp. 449 – 454.

5. Rosen. *Elementary Number Theory*. 4th ed. Addison-Wesley, 2000. pp. 71, 217.

6. Trappe, Washington. *Introduction to Cryptography*. University of Maryland Press, 2001. pp. 151 – 156.