Assorted Attacks on the RSA Cryptographic Algorithm

Ed ond J Murphy
Honors Thesis
Boston College
Co puter Science Dep rt ent
Professor How rd Str ubing
M y

Abstract: This thesis concentr tes on the vulner bilities of the RSA Cryptogr phic Algorith when it is not securely i ple ented hile it h s been proven th t brute force tt ck on the lgorith is not pr ctic l there re in spects of the lgorith th t require proper use to prevent b ck door tt cks The tt cks perfor ed in this thesis tte pt to e ploit both the tic l nd inherent ti ing vulner bilities of the lgorith Further ore si ple pr ctices which prevent theses tt cks re discussed

RSA Cryptographic Algorithm

Developed by Ron Rivest Adi Sh ir nd Len Adle n in the RSA public key cryptogr phic lgorith h s since been widely used in v riety of co puter security pplic tions

The first step of the lgorith is to select two different pri e nu bers p nd q Ne t we c lcul te the odulus n p q nd second ry odulus φ n p q e then select n integer e our public key such th t e is positive nu ber less th n nd rel tively pri e to φ n Fin lly we t ke the inverse of e od φ n to produce d our priv te key All of the fore entioned steps re used in pr ctice with very l rge nu bers to ensure dded security The size of these nu bers h s incre sed over ti e in ae e en a e e e one e ey a en e a a e aca

a a n a ee nea en e

a e ae c an ae ey e ae ea y enc y ae

a a n ee a n

$$e - M^e \; n$$

ee ec y e aa c e e e c ey n an M

a an e c c a e nay a e M e ann e ec e e

an e e a e ec e e e y e ea e a ee a n n e

ae ey

$$M - e \; n$$

e n e ae y c ea a e e e a n a e n e e ea

e can e een y e an n ee a n

$$M - e \; n - M^e \; n - M^e \; n$$

a y n e n c a y e ee

en en e an an n e e n an

c a n an n an a a y n e e

e n e a n

$$m^{k\phi(n)+} = m^{k(p-}$$

If Alice and Bob want to have a private conversation they each generate their own public and private keys and trade public key sets ($PubK_{Bob} = \{e_{Bob}, n_{Bob}\}$). If Bob wishes to send a message to Alice he encrypts his plaintext with the public key of Alice:

$$C_{Bob} = M_{Bob}^{e(Alice)} (\text{mod } n_{Alice})$$

and Alice uses her private key to decrypt the message.

A malicious computer user can very easily obtain public key information, as it is common knowledge on the network, and encrypted messages can be obtained by

Kocher Timing Attack

The implementation of a timing attack on the RSA cryptographic system exploits variations in the computation time of the decryption of the ciphertext. We start by analyzing a simple modular exponentiation for decryption M = C^d(mod n) where C has been obtained by eavesdropping on an ongoing conversation and the public key (e, n) is public knowledge. The following algorithm is used for the decryption where w is the number of bits, and the most significant bit is defined as '0':

Let $s_0 = 1$,
For k = 0 upto w-1:
  If (bit k of d) is 1 **then**
    Let $M_k = (s_k * C) \bmod n$.
  Else
    Let $M_k = s_k$.
  Let $s_{k+1} = M_k^2 \bmod n$.
EndFor.
Return (M_{w-1}).

Figure 1

Since computer operations are not always performed in constant speed we need to assemble a group, or block, of ciphertexts to develop reliable results. Using this block (the size of which will be discuss later in this paper) we now begin the timing portion of the attack, first computing the time needed to decrypt the message with the actual private exponent for each ciphertext (obtained by sending the ciphertext and modulus to the server) $T = e + \Sigma_{i=0}^{w-1} t_i$, where t_i corresponds to the amount of time needed to perform the decryption on bit i of the ciphertext and e represents the overhead within the decryption. At this point it is important to note that although the decryption algorithm above begins with k=0, the '0' is actually referring the most significant bit of the private key. We gather a block of ciphertexts and calculate the time to decrypt our private guess with the most significant bit equal to a 0 (T_0) and a 1 (T_1) for a single iteration of the decryption loop. By subtracting the guesses T_0 and T_1 from T we are left with the time that it takes to compute the guessed bits. Taking in to account the extra time needed by the algorithm to “decrypt” a bit that is set, as explained above, by simply computing the variances and subsequently comparing them we are now able to predict the first bit of the private key.

Notice that when bit k of d is set we do modular multiplication whereas when is not set there is a simple assignment. The time needed to perform the modular multiplication as well as the squaring is significantly more than the simple assignment and squaring and it is on this difference that we will focus our attack.

In theory, when comparing the variances of the two guesses the correct guess would have a smaller variance from the actual time expected. With the first bit guessed we can now proceed to the second and repeat the same procedure. As more bits are correctly guessed the timing period will increase, which in turn creates more stable results and higher percentage of correct guesses. On the other hand an incorrect guess would result in larger variance numbers indicating that you need to re-guess the previous bit.

After proving that the algorithm is exploitable the next step is to gather a block of ciphertexts, either by eavesdropping on an ongoing conversation or generating them using the previously obtained public exponent and modulus for the conversation. The experimental results, obtained by analysis of RSAREF Modular Multiplication and

Modular Exponentiation times, of Kocher's paper show that a block of 250 ciphertexts should produce the correct result 84% of the time. [2]

Kocher Implementation Attempts

Using the Java BigInteger package and Java timing package the first attempt at the attack was mounted. After being unable to obtain meaningful results we looked at the Java instance of the modular exponentiation routine being used to decrypt the data. We were able to determine that it was using the Montgomery Multiplication method to perform the operation, while Kocher's paper suggested a simpler algorithm using repeated squaring.

After implementing the repeated squaring algorithm and a method to extract the bits of the BigInteger keys, a second attempt at Kocher's timing attack was mounted. While the initial results look promising, repeated attempts showed that the percentage of most significant bits predicted correctly was hovering around fifty percent. Determined to produce meaningful results we made a slight alteration to Kocher's attack; rather than simply guessing that the first bit to be recovered was a '1' and expecting to see a higher variance when this bit was not set we decided to guess both a '1' and a '0' and compare the resulting variances.

Once again the percentage of correct results obtained appeared to be nothing more than a coin flip. Making another slight alteration I decided to have the algorithm attempt to guess the least significant bit first, but received much the same results. Having exhausted all possible alterations of the Kocher attack the only logical conclusion was that we were having problems receiving accurate timing results. In order to combat inaccurate timing we repeated the timing of

After many hours spent searching the internet for a package with the appropriate methods for the timing attack, and a few that failed to compile, we finally found MIRACL or the Multiprecision Integer and Rational Arithmetic C/C++ Library.[10] The package was largely self-explanatory and came with adequate documentation. The parameters for the functions were generally in the format of source, source, destination. The largest adjustment that I had to accommodate for in my code was the fact that the division algorithm returned the remainder of the division in the first parameter. This specification required a few extra steps to be taken to insure the data of the variable in the first argument remained unchanged.

After familiarizing myself with the new package I translated the Java code into C and looked for a appropriate method of timing. Following testing both inherent C timing methods and code developed by Bryant and O'Hallaron [4] we decided to go with the latter, as it offered clock cycle counting.

My code first generates an RSA key set with 256-bit encryption and a small public key (in the interest of minimizing the time to encrypt of the ciphertexts). It then enters a loop, which generates a number of ciphertexts (with a randomly generated number used as the plaintext) specified by the user via the command line and attempts to recover the most significant bit of the private key using this block of ciphertexts. The

n n n n nc
n c c n n cc
c n n c n c
nc n c n y ny n
n n c n n c nc
n c c c n
n c n n y
c c y c n c
n n c c
c n n cc n y y n n
n n n c
n n y n c c c
n n n n y
n c c c n y c n c
c n c n n y n n c n y c
n n y n n c
n y
n n n y n
nn n c nc c n n
n n n n n n n n
n c c n n y n
n c n c n n y n c
n y nc n n n c nn n

at constant rate. To my further disappointment while running the script I noticed that an increased number of ciphertexts in the block was not yielding any improvement in guessing the bit of the private key. At this point we decided, because of the inaccuracies of the timing results, we needed to shift our focus away from Kocher's attack and towards other attacks on the RSA Cryptographic system.

“Practical” Timing Attack

A second attack developed by Dhem, Koeune, Leroux, Mestre, Quisquater, and Willems uses the same general idea as Kocher's work, but attempts to simplify both the timing and the calculations performed. They state that although Kocher's idea was theoretically feasible and he presented a lot of data suggesting its possibility, there is no evidence that Kocher actually performed the attack himself. [5] The group of Belgian Computer Scientists were in fact unable to implement Kocher's idea in practice and decided to shift the focus of the attack. Rather than attacking the entire loop as Kocher does, they decided to attack the multiplication. Using a cryptographic library developed for the CASCADE smart card, they attacked the decryption algorithm shown below, The modular multiplication and squaring performed in this algorithm are done using the Montgomery method, and it is a small inconsistency in the multiplication method that Dhem, et. al. exploit. Namely, the method performs and extra subtraction when the intermediary result of the multiplication is greater than the value of the modulus. [5] Thus the ciphertexts can be separated into two groups, those that require the extra subtraction during Montgomery multiplication (C_1) and those that do not (C_2).

Looking back to the algorithm we can see that the multiplication step is performed only if bit i of the private key is a '1'. Using this knowledge, and the inconsistencies of the Montgomery multiplication we can see that when the private key is a '1' there should be a difference between the execution time of ciphertexts in group C_1 and the execution times of the ciphertexts in group C_2. Whereas if bit i of the private key is a '0' we would expect to see no timing difference between the two groups.

Just as with the Kocher attack, while the theory of the attack seems flawless the implementation presents problems that are hard to solve. Although Dhem, et. al. were able to recover 128-bit keys using 50,000 samples they do admit to limitations. [5] Beginning with the simpler of the two problems presented, "How do we know whether sample A is *different* than sample B" or how do we determine whether a reduction was performed on a given ciphertext or not. Although in theory the algorithm should run in constant time, in reality this is certainly not the case. This being the case we now have a difficult time identifying not only whether or not a reduction was performed, but also while running the actual attack we must decide how different the timing of group C_1 must be from group C_2 in order to assign the bit i of the private key to be a '1'. The second problem is inherent to the Montgomery multiplication and impossible to correct without modifying components to the RSA algorithm. In experimental results the Belgian group found that when RSA is allowed to operate as it should the extra reduction is only performed only 17% of the time. They were able to increase this probability to numbers as high as 50% by fixing the modulus and one of the factors, however these modifications would not be performed in practice and thus compromises the effectiveness of the attack.[5]

With this in mind the group reworked their attack to concentrate on the squaring operation that is performed figure 2. The attack on the square works essentially the same way as its multiplication counterpart, again relying on the extra reduction performed when using the Montgomery method. Rather than simply timing the entire loop and attempting to identify whether or not a multiplication was performed the attack is stopped right before the if statement, and from here the timing begins. As a result, instead of do the samples have to be) is also solved, as we are looking at a comparison between a guess of '1' or a guess of '0' and the actual time. Since we now have a predetermined point of reference we no longer have to calibrate our difference margin.

While very excited about the new and seemingly more successful timing attack as proposed by Dhem, Koeune, Quisquater and Willems, we realized that the issue of accurate timing results had not gone away. The paper by the Belgian scientists suggests that the attack is based on a variation in timing of 422 clock cycles out of 7,400,000, so it was clear to us that the accuracy of measurements was still crucial to the success of the attack. [5] So we decided that prior to any attempts at implementing the attack we should first secure accurate timing results.

Timing Trials and Tribulations

The first timing trials were performed using the Bryant and O'Halloran [4] code to time a multiplication, and a squaring using both the Lenstra LIP package [9] and the Scott MIRACL package.[10] The numbers that were used were randomly generated, with a ceiling of 2^{128}-1, using the random number generators supplied by the respective con ncy n o o n y c on o o o o

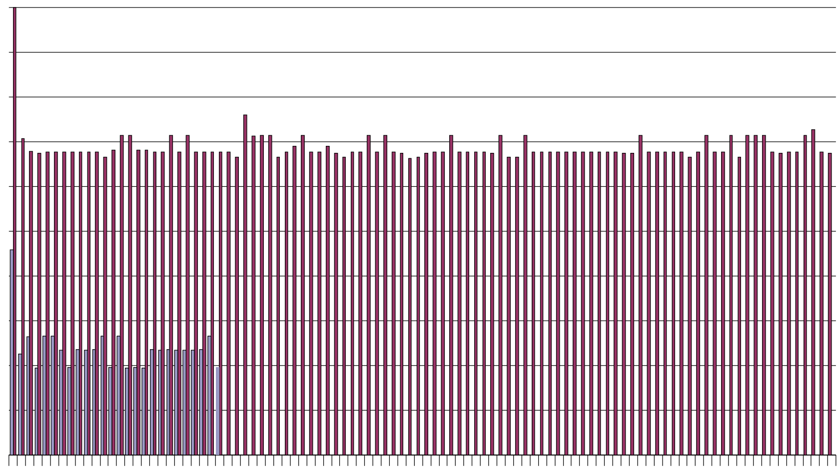

Figure 4

Additional tests were performed to compute the average time of the operation over 10000 trials. Statistics from the MIRACL test showed that the repetition seemed to decrease the randomness of the time values with each run of the test, however, clock cycle counts still varied largely from run to run. LIP statistics from these tests did not show any significant improvement over the timing of a single run of the operation, however, they did correct the high initial time problem.

Running out of options we decided to develop our own timing method using assembly code, and included a warming of the cache memory prior to the timings. Our assembly timing method produce much more consistent result, but did not improve the detection of a reduction step performed in the Montgomery operations.

Continued Fraction and the Continued Fraction Algorithm

Having little success in obtaining timing results that were accurate enough to perform a timing attack we decided to slightly shift the focus of the thesis to include other attacks we can be performed with out the reliance on timing. We were able to find a comprehensive list of such attacks in a paper written by Daniel Boneh that analyzed the many attacks that have been attempted since the RSA algorithm had been adopted in common practice. [6] We looked at each of the attacks and decided to focus on one that exposed a small private key using the mathematical theory of continued fractions.

Continued fractions are primarily used to discover a close approximation for the numerator and denominator of a real number when less approximate value of that fraction is known[7], and as we will shortly see this discovery has significant implications for the security of the RSA algorithm.[8] The common expression of a continued fraction is as follows:

q

For example to compute the continued fraction of $\frac{4}{11}$ we first invert the fraction:

$$0+\cfrac{1}{\cfrac{11}{4}}$$

then reduce the fraction in the denominator:

$$0+\cfrac{1}{2+\cfrac{3}{4}}$$

we repeat by inverting the fraction in the denominator:

$$0+\cfrac{1}{2+\cfrac{1}{\cfrac{4}{3}}}$$

and finally reduce the denominator to obtain the simple continued fraction:

$$0+\cfrac{1}{2+\cfrac{1}{1+\cfrac{1}{3}}}$$

The continued fraction expansion for $\frac{4}{11}$ is $< 0 ,2 ,1 ,3 >$.

It can be shown that the fraction can be reconstructed from q_0 using the following method:

$$
\begin{array}{lll}
n_0 = q_0, & d_0 = 1, & \\
n_1 = q_0 q_1 + 1, & d_1 = q_1, & \\
n_i = q_i n_{i-2}, & d_i = q_i d_{i-1} + d_{i-2} & \text{for } i = 2,3,...,m
\end{array}
$$

Figure 6 [8]

While m is the final reconstructed fraction, the intermediate values for $\frac{n}{d}$ are referred to as convergents. Finally, as per the continued fraction algorithm presented by Wiener[8]:

Given f', and underestimate of f
 hile f is not found
  Calculate q_i'
  Use figure 6 to construct
   $\langle q_0', q_1', ..., q_{i-1}, q_i + 1\rangle$ if is even,
   $\langle q_0', q_1', ..., q_{i-1}, q_i\rangle$ if is odd,
  Check whether the constructed fraction is equal to f.
End hile

re

Notice that 1 is added to the the term q_i before the convergent is computed if is a even number. This is done because the value for the guess of f should always be larger than f', since f' is an underestimate of actual f, and it can be shown that the convergent for even values of without the added value is indeed less than f'.

E posing S ll Priv te Key

The attack on a small RSA private key, as developed by Michael Wiener, makes use of continued fractions in order expose the private key, d. The attack works with a low private key because the fraction $\frac{e}{N}$, where e is the RSA public key and N is the RSA modulus, is a close underestimate of $\frac{\ }{d}$, where k is the result of $\frac{ed}{\phi(N)+1}$ and d is the RSA private key. It is important to note that because of the constraint of being a close underesti ate it can be show that the attack is only guaranteed to if the private

key satisfies the equation d $-N$

To e pose the private key we use the continued fraction algorith set forth in the previous section with a few odifications and e tra calculations to deter ine whether or not the convergent yields the correct continued fraction hat follows is a detailed description of odified continued fraction algorith

i

r $\overline{N}$

While the guess of d $-N$

Calculate q_i

Calculate r_i the re ainder when q_i is factored out of

Calculate the guess of $\frac{}{d}$ as described in the second step of 7

Calculate the guess of e d

Calculate the guess of ϕ n given by $\lfloor \frac{d}{} \rfloor$

e can now perfor our first test for an incorrect guess of the private key If the guess of ϕ n is equal to we can clearly assu e that the guess of d is incorrect and forgo the following two steps

Calculate the guess of $\frac{(+)}{}$ given by $N - \phi$

A o e eo eo e o ee e e eo o e e o e

which we increase the extent of the search for d by stopping a few loop iterations after the suggested boundary of $\frac{1}{3}N^{0.25}$. [6] He points out that the attack is guaranteed to work with in this boundary, but this does not mean that it will not work outside of the boundary.

The second improvement has more of a mathematical basis and significantly increases the discoverable private keys. Wiener states that the denominator of the underestimate (N) used in the attack is an over estimate of $\phi(n)$ and, while we don't know $\phi(n)$, he suggests a closer overestimate:

$$\lfloor(\sqrt{\quad}$$

While g is still computed by my program, because when using MIRACL division it is a byproduct of the guess of $\phi(n)$, it is not displayed when the program is run.

My first program performs an attack on one set of RSA keys generated randomly by the program, each intermediate step of the algorithm is displayed in the console. The private key is automatically set as close as possible to the boundary of $\frac{1}{3}N^{0.25}$, and in the case that the boundary is not relatively prime to $\phi(n)$, the next attempted value is generated by adding two to the boundary. Furthermore the user can specify, in the command line, how far above above the boundary they wish for the private to be. The number given by the user in the command line, x, is actually computed to be $x = 2^x$ and added to the boundary before generation of the private key. The variable x is calculated by the equation above because this feature was built in for testing the actual boundary for exposing a small exponent by continued fractions, and I quickly realized that this number needs to be quite large before the attack fails to function on a consistent basis. Furthermore, the execution time of the attack does not seem to be effected by the size of the public key and for 1024-bit encryption they key is exposed in an average of 15 milliseconds.

I then developed a second version of the small private attack program, which was intended for boundary testing purposes and simply runs the attack one hundred times, generating a new RSA key set each time, and keeping track of the successful attempts at private key exposure. After experimenting with single runs of this program to establish a general neighborhood of where the attack began to fail I developed a script to test the performance of the attack on numbers in that neighborhood. I then ran this script to test encryption sizes of 256, 512 and 1024-bits. The following graphs represent the results of the tests on 512-bit and 1024-bit encryption. In order to determine the percentages the script was run twice, thus percentages represent number of correct exposures over 1000 exposure attempts.

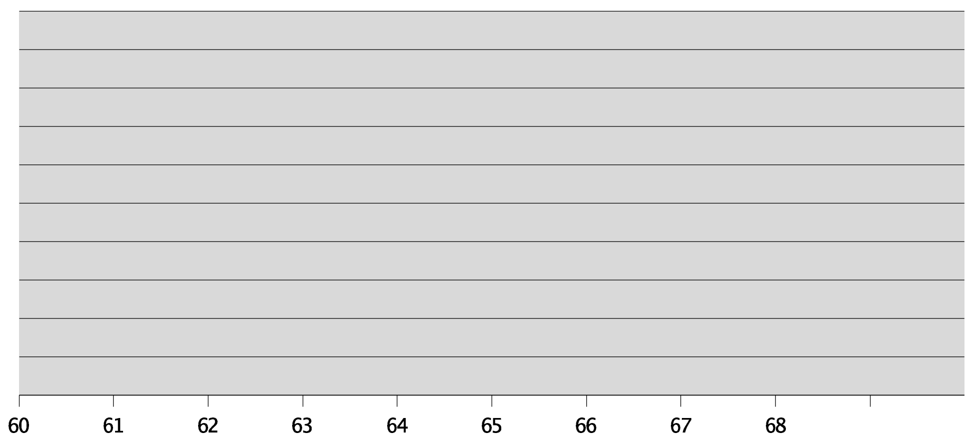

Figure 8

As is shown by the graph above, with 512-bit encryption, by simply allowing the to run until failure I was able to increase the boundary of insecure private keys by 2^{63} and still obtain a one hundred percent success rate. While at first this may seem like an incredible amount of added vulnerability, when taking as a percentage of the size of the encryption rate the increase is actually extremely small.

Figure 9

Figure 9 shows that when the rate of encryption is doubled so is the exponent for additional exposure. Taking in to account the both the rate of encryption and size of the additional exposure are exponents in the equation 2^z, the actual expansion of the boundary decreases when taken as a percentage of the encryption rate. In comparing the three graphs (including the 256-bit encryption test not shown) I found that the exponent used to test the boundary approximately doubled each time and as you can see above the rate of decline in the percentage of correct exposures declines at the same rate for all three encryption rates. These findings suggest that while Boneh's boundary may not be exactly right it is fairly close, and furthermore there is indeed a function that maps the size of the modulus to size of an insecure private key.

Preventing RSA Attacks

de y n o e c c on of e decryp on T s e od y see o e de ec se does no res n ny dd on c c on rns o s on y e e c rder o perfor T e n rod on of r ndo ngs s essen y e s e s e ncons s enc es n co p er ng nd eore c y c n e er ged o y ncre s ng e n er of s p es sed

oc er s p per s gges s c e er pre en on e od co on y referred o s nd ng T s e od c s for e c c on of n dd on se of r ndo y gener ed n ers For e A gor oc er s gges s s c osen o e re e y pr e o e od n nd s co p ed y e fo o ng eq on

$$v_i = (v_f^-$$

Further Exploration

With regard to the mathematical attacks I would like to work on expanding the boundary of the continued fractions, small private exponent attack, as well as implementing an attack on a low public exponent. To expand the boundary of the continued fractions attack I would perform testing on the improved denominator as suggested by Wiener. [8] Boneh suggested a number of attacks on a small public exponents, all of which rely on the LLL lattice algorithm. After gaining a solid understanding of the math behind this algorithm I would like to implement one of these attacks. My ideal goal would be to see how far I could advance both the small private and small public key vulnerabilities, in order to be able to suggest an optimal range for key generation.

References

[1] W. Stallings. Cryptography and Network Security: Principles and Practices. New Jersey: Prentice Hall 2003

[2] P. C. Kocher. Timing Attacks on Implementations of DiffieHellman, RSA, DSS and Other Systems <http://www.cyptography.com/resources/whitepapers/TimingAttacks.pdf>

[3] “Java SDK 1.4.2 API” <http://java.sun.com/j2se/1.4.2/docs/api/>

[4] Bryant and O'Halloran. <http://csapp.cs.cmu.edu/public/ics/code/perf/clock.c>

[5] Dhem, Koeune, Leroux, Mestre, Quisquater, and Willems. A Practical Implementation of the Timing Attack. <http://www.cs.jhu.edu/~fabian/courses/CS600.624/Timing-full.pdf>

[6] D. Boneh. Twenty Years of Attacks on the RSA Cryptosystem. <http://crypto.stanford.edu/~dabo/papers/RSA-survey.pdf>

[7] Eric W. Weisstein. "Continued Fraction." From MathWorld--A Wolfram Web Resource. <http://mathworld.wolfram.com/ContinuedFraction.html>

[8] Michael J. Wiener. Cryptanalysis of Short RSA Secret Exponents. IEEE Transactions on Information Theory, vol. 36. no. 3, 1990, pp.553-558. <http://www3.sympatico.ca/wienerfamily/Michael/MicaelPapers/ShortSecretExponents.pdf>

[9] Arjen Lenstra. LIP: Large Integer Package. Bellcore <http://www.enseignement.polytechnique.fr/profs/informatique/Philippe.Chassignet/97-98/BIGNUMS/lipdoc.ps>

[10] M. Scott. MIRACL: Multiprecision Integer and Rational Arithmetic C/C++ Library. Shamus Software Ltd. <http://indigo.ie/~mscott/>