Lattice Reduction on Low-Exponent RSA

Jason Dyer

August 2002

1 Preliminaries to Coppersmith's Algorithm

Coppersmith's algorithm relies on a simple flaw in the RSA algorithm when messages are small compared to the public number N. Consider a message x encrypted with exponent $e = 3$ using modulus N for the public key where $a < \sqrt[3]{N}$. Then the encryption z of x can be decrypted simply by taking the cube root, because the x^3 operation never rotated x over the modulus N.

This is a highly specific case, but it can be generalized to other cases, the most interesting being Coppersmith's short pad attack. In the short pad attack the message has the same conditions as above but also a simple padding P which is known to the code-breaker. When $e = 3$ the encryption can be considered forming the polynomial $(x+P)^3 = z$. Then Coppersmith's algorithm can be applied – this will solve the polynomial, reducing the case to the simple one above. (For a discussion of good padding that disallows this attack, see the section on Proper Use of Random Padding.)

Also of note is the Franklin-Reiter related message attack. If the user sends related messages such that the related part can be considered equivalent to "padding", the same problem arises. (An example of this might be starting a set of messages with "The password is".) The sending of similar messages is a user problem, not an programmer problem, and thus cannot be controlled. While it is extremely unlikely the proper conditions will occur, any user action must be accounted for. The discussion here will be restricted to the short pad attack, but the fact the related message attack exists will become important later.

2 The Algorithm Itself

The basic idea here is to find a polynomials whose roots contain the solution and whose coefficient are small so that its roots are calculatable. We obtain this polynomial as a linear combination of a collection of polynomials whose roots all contain the solution. Let us suppose we have a polynomial $(x+P)^3$ or

$$x^3 + 3Px^2 + 3P^2x + P^3.$$

Note that if $(x+P)^3 - z = 0 \bmod N$ then $(x+P)^3 - z = 0 \bmod N^k$. We need to form the following polynomials:

$$g_{u,v}(x) = N^{m-v}x^u f(x)^v$$

for some predefined m. m can be considered the "expansion factor" and will be precisely defined later. u will be defined as $0, ..., m$ and v will be defined as $0, ..., d-1$.

For example, when $m = 3$, $u = 1$ and $v = 2$, the polynomial $g_{0,0}$ will be

$$N^{3-2}x^1 f(x)^2$$

or rather

$$N(x^7 + 6x^6P + 15x^5P^2 + 20x^4P^3 + 15x^3P^4 + 6x^2P^5 + xP^6).$$

We form a lattice considering all possibilities for u and v. The lattice is designed so the first column represents the constant coefficient, the second column represents the coefficient with x^1, the third the coefficient with x^2, and so on. In the example given the rows would be $g_{0,0}, g_{1,0}, g_{2,0}, g_{0,1}, g_{1,1}, g_{2,1}$, etc. Let $a = 3P$, $b = 3P^2$ and $c = P^3$ and form the matrix starting with:

$$\begin{array}{cccccccccccc} N^3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & N^3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & N^3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ cN^2 & bN^2 & aN^2 & N^2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & cN^2 & bN^2 & aN^2 & N^2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & cN^2 & bN^2 & aN^2 & N^2 & 0 & 0 & 0 & 0 & 0 & 0 \end{array} \tag{1}$$

and so on until $u = 2$, $v = 3$.

If we can find the short vectors of the matrix we will be able to find vectors less than N^m and therefore the answer we are seeking because these vectors will represent tractable polynomials with roots that are possible solutions to the problem. The LLL algorithm is able to do this. We will consider it a "black box" and not go into specifics here; essentially it performs Gaussian reduction in an iterated fashion to get approximations of short vectors. In this case the approximation is good enough to solve our problem.

More specifically, the LLL algorithm will work if $m = \log N/e$. This is the expansion factor discussed earlier. Once the short vectors are generated solving the polynomials will yield the answer to the problem.

Note that all the results above can be generalized for any e. However, most prior tests have been done using only $e = 3$ and considering a "high-enough" e to be out of range without giving specifics.

I tried Various values of e and m of the algorithm in an implementation using Mathematica 4.1. The results are in the Appendix.

3 Some further considerations

A paradox of sorts appears to arise when N is large: the algorithm will actually work faster when e is large than when e is small. Consider a 150-digit number N (approximately RSA-512). $\log N$ is approximately 150. When $e = 17$, $m = 9$ to solve the equation, but when $e = 3$, $m = 50$. However, it should be noted that the expansion factor $m = 50$ accounts for the larger number of messages that can be processed (that is, $x < \sqrt[3]{N}$) and using $e = 3$, $m = 9$ will make the algorithm account for merely those that satisfy $x < \sqrt[17]{N}$ and resolve the paradox.

Therefore, in practical use of the algorithm the user should be given an option to change the bound, so that they will not be puzzled by the difference in calculations.

4 Proper use of random padding

What constitutes a "good random padding" also deserves mention. We will use base 2 (that is, what computers represent numbers as) in the examples. One method of padding would be shifting the message to represent the high bits of a cipher before adding random padding to the low bits. The message 10101010 when shifted by 8 will be 1010101000000000; random padding would turn this into (for example) 1010101011010011. Coppersmith points out, given $e = 3$, that this padding will allow the attack to succeed, given the padding is less than one-ninth the length of N.

Coppersmith in [3] discusses several ways around this problem, the simplest being to make the padding larger than the necessary length. He concludes this is an inefficient method (given there are methods of using less padding that maintain security), and also notes that increasing the number of encrypted messages may make the padding threshold be even less, up to one-sixth of the length of N.

Spreading the encryption throughout the message defeats the attack entirely. For example, given a message in a 6 bit representation (so that each character consists of six binary digits, allowing 64 possible characters) it could be converted into 8 bits with two random bits added. A two-character message with the characters 110011 and 001100 would be changed into (for example) 11001111 and 00110001. However, uniformly spreading this system across the message gives the same inefficiency. Consider a message that is 500 characters long. The 6 bit representation contains 3000 bits. The 8 bit version contains an extra 1000. This expands the length of the message by a full third, far greater than we needed. Therefore the padding should be spread out; perhaps one character every five, this would make the ration of message to padding 1 to 15. Given $e = 3$, if this ratio is not less than 1 to 9 we might as well use the previously mentioned "inefficient" method.

Also note that the Franklin-Reiter related message attack applies in cases where messages are large or small, the only condition being that the related part is a sizable enough proportion. Therefore padding (if used as security when $e = 3$) should be applied to all cases, not just ones where the message is small. As mentioned previously, while the conditions for an attack are unlikely every user action must be accounted for.

5 Conclusions on Low Exponent RSA

If proper padding is used as specified in the previous section, the Coppersmith algorithm fails in every case. However, current wisdom on the subject (as given, for example, in [4]) gives the recommends setting the exponent to something high, like $e = 65535$, rather than using padding.

In cases on software this is a sensible plan, and it allows the implementer to not worry about padding security. However, it should be noted that judging from the practical results in Appendix A, $e = 65535$ is perhaps a bit paranoid and a lower number will suffice.

The popular encryption program PGP uses $e = 17$ and padding in cases where the message is small, but not padding in all messages – this leaves the theoretical possibility open of performing the Franklin-Reiter attack if $e = 17$ is too low. Judging from Appendix A it would take years to calculate, but it is possible with heavy optimization and supercomputer resources this is managable. The exact limits on the computation remains an open question.

With certain hardware implementations, an exponent of $e = 3$ may be highly desirable compared to a larger number. If this is the case the padding in the prior section should be used and the message will remain secure. This is contrary to the recommendation of [4] which indicates e should always be higher.

References

[1] D. Boneh. 20 years of attacks on RSA. *Notices of A.M.S.*, vol. 46, no. 2, pp. 203-213, 1999.

[2] M. Cary. *Lattice Basis Reduction Algorithms and Applications.* `http://www.cs.washington.edu/homes/cary/lattice.pdf`

[3] D. Coppersmith. Small Solutions to polynomial equations, and low exponent RSA vulnerabilities. *Journal of Cryptography*, vol. 10, pp. 233-260, 1997.

[4] P. Garrett. *Making, Breaking Codes.* Prentice Hall, 2001.

[5] "infiNity". The PGP Attack FAQ. `http://www.stack.nl/~galactus/remailers/attack-faq.html`

[6] L. Lovasz. *An Algorithmic Theory of Numbers, Graphs and Convexity.* SIAM Publications, 1986.

[7] A. Menezes, P. van Oorschot, and S. Vanstone. *Handbook of Applied Cryptography.* CRC, 1996.

[8] PGP Frequently Asked Questions With Answers. `http://www.faqs.org/faqs/pgp-faq/part1/`

[9] C. P. Schnorr. A Hierarchy of Polynomial Time Lattice Basis Reduction Algorithms. *Theoretical Computer Science 53.* Elsevier Science, pp. 201-224, 1987.

[10] A. Schönage. Factorization of univariate integer polynomials by diophantine approximation and by an improved basis reduction algorithm. *Proceeds of 11th Coll. on Automata, Languages and Programming*, Antwerpen 1984, *Lecture Notes in Computer Science 172.* Springer, pp. 436-447, 1984.

Appendix I: Timing Results and Source Code

All results obtained using Mathematica 4.1 on a 850 mhz Pentium III

```
e = 3; m = 3: 0.83 Seconds
e = 5; m = 3: 2.59 Seconds
e = 7; m = 3: 10.38 Seconds
e = 9; m = 3: 31.03 Seconds
e = 17; m = 3: 672.57 Seconds
e = 3; m = 3: 0.83 Seconds
e = 3; m = 5: 2.75 Seconds
e = 3; m = 7: 11.37 Seconds
e = 3; m = 9: 57.45 Seconds
e = 3; m = 17: 973.88 Seconds
e = 3; m = 3: 0.83 Seconds
e = 4; m = 4: 3.3 Seconds
e = 5; m = 5: 21.2 Seconds
e = 6; m = 6: 106.33 Seconds
e = 7; m = 7: 510.64 Seconds
e = 8; m = 8: 2112.27 Seconds
```

In calculating the scale of larger calculations, only very rough approximation is needed. Using a simple expansion factor of 5 (roughly what the previous results yield):

```
e = 9; m = 9: 10361
e = 10; m = 10: 52807
e = 11; m = 11: 264034
e = 12; m = 12: 1320169
e = 13; m = 13: 6600844
e = 14; m = 14: 33004219
e = 15; m = 15: 165021094
e = 16; m = 16: 825105469
e = 17; m = 17: 4125527344 (130 years)
```

and presuming the same expansion rate from between
(e = 3; m = 9)->(e = 3; m = 17) as from between
(e = 17; m = 9)->(e = 17; m = 17):

```
e = 17; m = 9: 243368327 (8 years)
```

Note that this is only precise enough to give an idea of scale; that is, we know the calculation takes years rather than centuries.

```
[Sample variables for testing]
p:=3011
q:=1889
n:=p*q
P:=11111111
x:=22
e:=3
z:=PowerMod[o + P, e, p * q]
m=3
[The code proper starts here]
f := (P + x)^e - z
d := e
dim := e*(m + 1)
R := Array[r, {dim, dim}]
crow := 0
For[v = 0, v <= m, For[u = 0, u <= d - 1, crow++;
    For[ccol = 1, ccol <= dim, If[ccol > crow, r[crow, ccol] := 0];
      If[(ccol - u - 1) < 0, r[crow, ccol] := 0];
      If[ccol <= crow && (ccol - u - 1) >= 0,
        r[crow, ccol] = (n^(m - v))*Coefficient[f^v, x, ccol - u - 1]];
      ccol++];
    u++];
  v++]
Print["Initialization complete."]
Timing[LatticeReduce[R]]
Clear[x]
For[i = 1, i <= dim,
  s = 0;
   For[j = 1, j <= dim, s2 = s;
    s = s2 + r[i, j]*x^(j - 1);
    j++];
  Print[Solve[{s == 0, Modulus == n}, x, Mode->Modular]];
i++]
```