Key Recovery Method for CRT Implementation of RSA

Matthew Campagna, Amit Sethi,
{matthew.campagna, amit.sethi}@pb.com
Secure Systems
Pitney Bowes, Inc.

Abstract

This paper analyzes a key recovery method for RSA signature generation or decryption implementations using the Chinese Remainder Theorem (CRT) speed up. The CRT-based RSA implementation is common in both low computing power devices and high speed cryptographic acceleration cards. This recovery method is designed to work in conjunction with a side-channel attack where the CRT exponents are discovered from a message decryption or signature generation operation, the public exponent is assumed small and the public modulus is unknown. Since many RSA implementations use the small, low hamming weight public exponent 65537 this turns out to be a realistic method. An algorithm for recovering the private key, modulus and prime factorization candidates is presented with a proof of correctness. Runtime estimates and sample source code is given.

1 Introduction

The basic problem that we address in this paper, is that of recovering all RSA parameters, given only the private CRT-RSA exponents, and a message-signature pair *(m,s)* (or a plaintext-ciphertext pair). We make an assumption about the public exponent, if it is not known. Specifically, we will assume that the public exponent is relatively small, in an exhaustible space.

There are several scenarios in which this method of key recovery would be useful. One scenario is where an adversary performs a side-channel attack on a device to recover the CRT-RSA exponents, but does not know any other public or private key information. Note that this is very likely, since a single power trace is sufficient to reveal exponents in many devices. One power trace does not usually reveal information about any other parameters, however. Other scenarios where our method is useful, include loss of public keys, and loss of legitimate private key records.

1.1 RSA

RSA is a popular public-key encryption scheme, and a signature scheme. It was designed in 1978 by R.L. Rivest, A. Shamir, and L. Adleman. We will only discuss the signature scheme here. The encryption scheme is similar. The algorithms in this section are slightly modified versions of the algorithms in [3].

To generate an RSA key, we do the following:

Algorithm 1.1 ***RSA Key Generation***
INPUT: Bitlength of modulus, k.
OUTPUT: Public key (e, N), and private key (d, N).

1 Generate prime p of bitlength $\lfloor k/2 \rfloor$

2 Generate prime q of bitlength $k - \lfloor k/2 \rfloor$

3 Compute $N \leftarrow p \cdot q$

4 Select e to be an integer, where $GCD(e, (p-1) \cdot (q-1)) = 1$

5 Compute d such that $e \cdot d \equiv 1 \pmod{(p-1) \cdot (q-1)}$

6 Return $((e, N), (d, N))$

In practice, e is often a small and low density integer. A small and low density public key provides computational efficiencies during the encrypt and verify RSA algorithms.

To sign a message, we use the following algorithm, where h is a hashing algorithm, such as $SHA1$:

Algorithm 1.2 ***RSA Sign***
INPUT: Private Key (d, N), and message to be signed, m.
OUTPUT: s, signature of m using private key (d, N).

1 $s \leftarrow h(m)^d \pmod{N}$

2 Return (s)

To verify a signature s, we use the following algorithm:

Algorithm 1.3 ***RSA Verify***
INPUT: Public Key (e, N), and message-signature pair to be verified (m, s).
OUTPUT: $VALID$ or $INVALID$.

1. $hm' \leftarrow s^e \pmod N$
2. *If* $h(m) = hm'$, *Return* $VALID$
   Else, Return $INVALID$

The RSA algorithm is believed to be secure if the problem of factoring large integers is "hard". That is, given a large number n with two large prime factors p and q, it must be computationally infeasible to find p and q. Otherwise, an adversary could factor n, and then easily find $d \equiv e^{-1} \pmod{(p-1)\cdot(q-1)}$ to obtain the private key. Currently, an efficient algorithm for factoring integers does not exist. However, we do not know whether there is a simpler method of finding d, given the public key. That is, we do not know whether factoring is necessary to find the private key.

Even though the RSA algorithm is assumed mathematically secure, specific implementations may not be secure. "Side-channel" attacks against implementations are possible. These attacks do not attack the algorithm directly using mathematical techniques, but instead use information (such as power consumption or electromagnetic emissions) leaked from an implementation in order to derive information about the private key.

The exponentiation is the most time-consuming operation in RSA. The Chinese Remainder Theorem is used to obtain an approximate 4-fold increase in signing speed.

The public exponent is frequently chosen to be quite small to speed up verification, since exponentiation is the most time-consuming operation in RSA. To obtain an approximately 4-fold increase in signing speed, the Chinese Remainder Theorem (CRT) is often used.

1.2 Chinese Remainder Theorem

The Chinese Remainder Theorem provides a method for solving certain systems of congruence relations. One of the many applications of the theorem, is speeding up RSA signature generation.

Theorem 1.1 *Let $m_1, m_2, \ldots, m_n$ be distinct positive integers such that $GCD(m_i, m_j) = 1$ if $i \neq j$. Then, for any integers $a_1, a_2, \cdots, a_n$, the simultaneous congruences*

$$x \equiv a_1 \pmod{m_1}$$

$$x \equiv a_2 \pmod{m_2}$$
$$\vdots$$
$$x \equiv a_n \pmod{m_n}$$

always have a solution. Moreover, if $x = x_0$ is one solution, then the complete solution is $x \equiv x_0 \pmod{m_1 m_2 \ldots m_n}$ [1]. The solution [4] is

$$x_0 \equiv a_1 b_1 \frac{M}{m_1} + \ldots + a_n b_n \frac{M}{m_1} \pmod{M}$$

where

$$M = m_1 m_2 \ldots m_n$$

and the b_i are determined from

$$b_i \frac{M}{m_i} \equiv 1 \pmod{m_i}$$

Garner's Algorithm is an efficient implementation of the Chinese Remainder Theorem, and can be used to find solutions to systems of congruences. The following algorithm has been obtained from [2]:

Algorithm 1.4 ***Garner's Algorithm***
INPUT: A positive integer $M = \prod_{i=1}^{n} m_i > 1$, with $GCD(m_i, m_j) = 1$ for all $i \neq j$, and $a(x) = (a_1, a_2, \ldots, a_n)$.
OUTPUT: The integer x in radix b representation.

1 *For i from 2 to n do:*

- 1.1 $C_i \leftarrow 1$
- 1.2 *For j from 1 to $(i-1)$ do:*
  - 1.2.1 $u \leftarrow m_j^{-1} \pmod{m_i}$ *(Using the Euclidean Algorithm)*
  - 1.2.2 $C_i \leftarrow u \cdot C_i \pmod{m_i}$

2 $u \leftarrow a_1, x \leftarrow u$

3 *For i from 2 to n do:*

- 3.1 $u \leftarrow (v_i - x) C_i \pmod{m_i}$
- 3.2 $x \leftarrow x + u \cdot \prod_{j=1}^{i-1} m_j$

4 *Return(x)*

1.3 CRT-RSA

Now, we will consider a way of speeding up signature generation, using the Chinese Remainder Theorem. Using this method, we store the private key in a different format. The public key is still (e, N), but the private key is now (d_p, d_q, p, q), where $d_p \equiv d \pmod{(p-1)}$, and $d_q \equiv d \pmod{(q-1)}$. As in the previous RSA Sign/Verify algorithms, h represents a hashing function.

Algorithm 1.5 ***CRT-RSA Sign***
INPUT: Private Key (d_p, d_q, p, q), and message m to be signed.
OUTPUT: Signature s of m using private key (d_p, d_q, p, q).

1 $s_p \leftarrow h(m)^{d_p} \pmod{p}$

2 $s_q \leftarrow h(m)^{d_q} \pmod{q}$

3 Solve the following two congruences mod pq using Garner's Algorithm:
$s \equiv s_p \pmod{p}$
$s \equiv s_q \pmod{q}$

4 Return (s)

Similarly, we can make decryption faster by performing two smaller exponentiations, instead of directly computing $m \equiv c^d \pmod{N}$. Once we have computed M_p and M_q, we can efficiently combine the solutions using Garner's Algorithm to get the required result.

2 Key Recovery Method

We start this section under the assumption that the CRT exponents d_p and d_q have been recovered via side channel analysis. If the public key information is known (e, N) it is straight forward to recover the factorization of N and the full private key. We can compute $m_p \equiv c^{d_p} \pmod{N}$ for some $c \equiv m^e \pmod{N}$. Then $p = gcd(m - m_p, N)$. Similarly we can recover q. This is clear since

$$
\begin{aligned}
m \equiv c^d \pmod{N} \Longrightarrow m &\equiv c^d \pmod{p} \\
&\equiv c^{d_p + k\cdot(p-1)} \pmod{p} \\
&\equiv c^{d_p} c^{(p-1)k} \pmod{p} \\
&\equiv c^{d_p} (1)^k \pmod{p} \\
&\equiv c^{d_p} \pmod{p}
\end{aligned} \tag{1}
$$

Also, $m_p \equiv c^{d_p} \pmod{n} \Longrightarrow m_p \equiv c^{d_p} \pmod{p}$ since $n = pq$, $m \equiv m_p \pmod{p}$. So, $m - m_p = t \cdot p$ for some t, hence $m - m_p$ and n, have a common factor p.

Next we present an algorithm which produces a set of candidates for the public key exponent and prime factorization given the CRT exponents d_p and d_q, and an upper bound for the public exponent e. We show given a message-signature pair or plaintext-ciphertext pair it is a trivial matter to test the candidates for the correct solution.

2.1 Prime Candidate Recovery Algorithm

Algorithm 2.1 ***Constructing Prime Candidate Set Algorithm***

INPUT: CRT exponents $d_p, d_q > 0$, and an upper bound u for e
OUTPUT: Prime factors and public exponent set $\{(e_1, p_1, q_1), \ldots (e_k, p_k, q_k)\}$

1 Set $i \leftarrow 0$, and $j \leftarrow 3$

2 Compute $d1_p \leftarrow d_p \cdot j - 1$

3 Set $k \leftarrow 3$

4 Compute h and r such that $d1_p = kh + r$

5 If $r = 0$

- *5.1 Set $p \leftarrow h + 1$*
- *5.2 If p is a prime*
  - *5.2.1 Compute $d1_q \leftarrow d_q \cdot j - 1$*
  - *5.2.2 Set $l \leftarrow 3$*
  - *5.2.3 Compute h and r such that $d1_q = lh + r$*
  - *5.2.4 If $r = 0$*

5.2.4.1 Set $q \leftarrow h + 1$

5.2.4.2 If q *is a prime Set* $e_i \leftarrow j$, $p_i \leftarrow p$, $q_i \leftarrow q$, $i \leftarrow i + 1$

5.2.5 Set $l \leftarrow l + 1$

5.2.6 If $l < j$ *Go to 5.2.3*

6 Set $k \leftarrow k + 1$

7 If $k < j$ *Go to 4*

8 Set $j \leftarrow j + 2$

9 If $j \leq u$ *Go To 2*

10 Return $\{(e_1, p_1, q_1), \ldots, (e_k, p_k, q_k)\}$

We now analyze the correctness of this prime factor recovery method, the approximate runtime complexity, and the probabilistic cardinality of the candidate set.

First we show that under the assumption that the odd public exponent e is $3 \leq e \leq u$, and the RSA factors p and q are true primes, the returning set contains the triplet (e, p, q) where e is the public exponent and p and q are the prime factors of the RSA modulus. It is clear from the algorithm description that j will eventually be set to e. Now by construction $d_p \equiv d \pmod{p-1}$, from this it follows that $e \cdot d_p - 1 \equiv 0 \pmod{p-1}$, since

$$\begin{aligned} d_p &\equiv d \pmod{p-1} \\ \Longrightarrow e \cdot d_p &\equiv e \cdot d \pmod{p-1} \\ \Longrightarrow e \cdot d_p &\equiv 1 \pmod{p-1} \\ \Longrightarrow e \cdot d_p - 1 &\equiv 0 \pmod{p-1}. \end{aligned}$$

Therefore $e \cdot d_p - 1 = t \cdot (p-1)$ for some $0 < t < e$. Hence when $j = e$ and $k = t$ Step (4) will be satisfied; and we will enter the loop defined within Step 5.b. Now, for a similar argument, $e \cdot d_q - 1 = s \cdot (q-1)$ for some $2 < s < e$. Hence when $j = e$, $k = t$ and $l = s$ the causal solution (e, p, q) will be placed in the solution set. This satisfies the requirement that the algorithm will return a non-empty solution set under the assumptions above and that it will contain the correct RSA parameters (e, p, q).

So now let's consider the probabilistic size of the candidate set. Let $S = \{(e_i, p_i, q_i)\}$ produced by the algorithm. For any given j, k and l, we need to consider the probability of contributing to this set. Given $d1_p = d_p \cdot j - 1$, what is the probability that k divides $d1_p$ evenly. This is fairly straight forward probability of $1/k$. Now given the instance that $d1_p = k \cdot h$ for some $h \in \mathbb{N}$ what is the probability that $h+1$ is a prime number. This is governed by the Prime Number Theorem that has as a consequence that the probability that a random number n is prime is roughly $1/log(n)$. In our case we estimate this value to $1/log(d_p)$ since $h+1 \approx d_p$ in size. Similarly for l we have a $1/l$ probability that $d1_q = l \cdot h$, for some $h \in \mathbb{N}$ and a probability of $1/log(h+1)$ that $h+1$ is prime, which we can estimate by $1/log(d_q)$. Thus for any given j, k and l the probability that it contributes a candidate is:

$$\begin{aligned} Pr((j,k,l) \text{ contributes to } S) &\approx \left(\frac{1}{k}\right)\left(\frac{1}{log(d_p)}\right)\left(\frac{1}{l}\right)\left(\frac{1}{log(d_q)}\right) \\ &= \left(\frac{1}{l \cdot k \cdot log(d_p) \cdot log(d_q)}\right) \end{aligned}$$

These facts can be used to estimate the cardinality of our solution set S. We now have an estimate for the probability of being included in the solution set, if we sum these probabilities over all experiments we get an approximate cardinality of our candidate set.

$$\begin{aligned} |S| &\approx \sum_{j=1}^{\lfloor u/2 \rfloor} \left(\sum_{k=3}^{2j+1} \frac{1}{k} \cdot \frac{1}{log(d_p)}\right)\left(\sum_{l=3}^{2j+1} \frac{1}{l} \cdot \frac{1}{log(d_q)}\right) \\ &= \left(\frac{1}{log(d_p) \cdot log(d_q)}\right) \sum_{j=1}^{\lfloor u/2 \rfloor} \left(\sum_{k=3}^{2j+1} \frac{1}{k}\right)\left(\sum_{l=3}^{2j+1} \frac{1}{l}\right) \\ &\approx \left(\frac{1}{log(d_p) \cdot log(d_q)}\right) \sum_{j=1}^{\lfloor u/2 \rfloor} (log(2j+1) - 3/2)^2 \end{aligned}$$

Now if we assume $u = 65537$, and p and q are 512-bits, we can further approximate the cardinality of S:

$$
\begin{aligned}
|S| &\approx \left(\frac{1}{log(d_p)\cdot log(d_q)}\right)\sum_{j=1}^{32768}(log(2j+1)-3/2)^2 \\
&\approx \left(\frac{1}{log(2^{512})}\right)^2\sum_{j=1}^{32768}(log(2j+1)-3/2)^2 \\
&\approx \left(\frac{2450944}{125948}\right) \\
&\approx 19
\end{aligned}
$$

This value is a very reasonable set to subject to obvious secondary testing. It should be noted that each triplet (e_i, p_i, q_i) in S can be used to define valid RSA public and private keys $(e_i, N_i = p_i \cdot q_i)$ and d_i such that the CRT exponent equations $d_p \equiv d_i \pmod{p_i - 1}$ and $d_q \equiv d_i \pmod{q_i - 1}$ hold. If we are given a valid message-signature (m, s), we can test the candidates by computing the corresponding public key (e_i, N_i) and testing $h(m) \equiv s^{e_i} \pmod{N_i}$, where h is the hashing function used to generate the signature. Clearly, the causal candidate will pass this test; and all the non-causal values will fail with almost certain probability.

In the case where the value e is known the set greatly reduces. We get the following estimate for a fixed e value.

$$
\begin{aligned}
|S_e| &\approx \left(\sum_{k=3}^{e}\frac{1}{k}\frac{1}{log(d_p)}\right)\left(\sum_{l=3}^{e}\frac{1}{l}\frac{1}{log(d_q)}\right) \\
&= \left(\frac{1}{log(d_p)\cdot log(d_q)}\right)\left(\sum_{k=3}^{e}\frac{1}{k}\right)\left(\sum_{l=3}^{e}\frac{1}{l}\right) \\
&\approx \left(\frac{(log(e)-3/2)^2}{log(d_p)\cdot log(d_q)}\right)
\end{aligned}
$$

For the following e values and N sizes we get the following table of non-casual cardinality of S_e.

e	$log_2(N)$	$\|S_e\|$
3	512	0.000003
3	1024	0.00000088
3	2048	0.00000022
65537	512	0.0029
65537	1024	0.00073
65337	2047	0.00018

Now if the actual RSA public exponent is a small we e would expect to see the causal primes p and q and additional prime candidates with very low probability. As e grows the probability of course increases.

3 Conclusion

In practice the source code of the following section produced around 90 candidates, roughly four times of the expected cardinality. This is largely due to the non randomness of the actual integers $d1_p$ and the resulting values being tested for primality. The candidate set contains many large p_i and q_i multiple times.

This solution raises two obvious extensions. Namely as u, the upper bound, increases or if e is a large known value how can we expand this method to solve for d and N, or p and q? A second obvious question is in regards to selected CRT method. If the underlying RSA signature algorithm is not using the CRT method and the full exponent d is recovered instead of the CRT exponents d_p and d_q how can we expand this method to solve for e and N when e is small but unknown or large but known.

4 Source Code

This section contains some sample source code that implements the candidate set construction for two CRT exponents and an assumption of $e < 65538$.

```
int     crt_rsa_recover()
{
    rsa_key_pair        keys;
    csl_int             dp, dq, p, q, N, a, m, c, d, r, p1, q1, dp1, dq1;
    int                 i, count, count2, count3, l, t, j, k;
    //  initialize
    rsa_key_init(keys);
    csl_init(dp);    csl_init(dq);    csl_init(p);    csl_init(q);
    csl_init(N);     csl_init(a);     csl_init(m);    csl_init(c);
    csl_init(d);     csl_init(r);     csl_init(p1);   csl_init(q1);
    csl_init(dp1);   csl_init(dq1);
    //  generate key pair
    rsa_gen_keypair(keys, 1024);
    //  print out key pair
    printf("RSA PARAMS:\np: ");    csl_print(stdout, keys->p);
    printf("\nq: ");        csl_print(stdout, keys->q);
    printf("\nN: ");        csl_print(stdout, keys->N);
    printf("\ne: ");        csl_print(stdout, keys->e);
    printf("\nd: ");        csl_print(stdout, keys->d);
    printf("\n");
    //  ok, now grab p-1, and q-1
    csl_sub_ui(p1, keys->p, 1);    csl_sub_ui(q1, keys->q, 1);
    //  compute d_p = d (mod(p-1)) and d_q = d (mod(q-1))
    csl_mod(dp, keys->d, p1);      csl_mod(dq, keys->d, q1);
    //  print out recovered exponents
    printf("d_p = ");       csl_print(stdout, dp);
    printf("\nd_q = ");     csl_print(stdout, dq);
    //  compute dp1, dq1
    csl_mul(dp1, dp, keys->e);    csl_mul(dq1, dq, keys->e);
    csl_sub_ui(dp1, dp1, 1);     csl_sub_ui(dq1, dq1, 1);
    count = 0;    count2 = 0;    count3 = 0;
    //  start the generalized loop for keys->e unknown
    for(j=3;j<65538;j+=2){
        csl_mul_ui(dp1, dp, j); csl_sub_ui(dp1, dp1, 1);
        for(k=3;k<=j;k++){
            csl_set_ui(d, k);   csl_div_n_qr(a, r, dp1, d);
            if(r->size == 0){
                ++count;        csl_add_ui(p, a, 1);
                if((p->d[0]&0x01)&&(miller_rabin(p, 10)){
                    ++count2;   csl_mul_ui(dq1, dq, j);
                    csl_sub_ui(dq1, dq1, 1);
                    for(l=3;l<=j;l++){
                        csl_set_ui(d, l);
```

```
                        csl_div_n_qr(a, r, dq1, d);
                        if(r->size == 0){
                            csl_add_ui(q, a, 1);
                            if((q->d[0]&0x01)&&(miller_rabin(q, 10)){
                                ++count3;
                                printf("candidate: (%d, ", j);
                                csl_print(stdout, p);
                                printf(", ");
                                csl_print(stdout, q);
                                printf(")\n");
    }   }   }   }   }   }   }
    printf("count: %d \t count2: %d\t count3: %d\n", count, count2, count3);
    return 0;
}
```

References

[1] William J. Gilbert and Scott A. Vanstone. *Classical Algebra.* Graphic Services, University of Waterloo, 3rd edition, 1993.

[2] Alfred Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography.* CRC Press, Oct 1996.

[3] R.L. Rivest, A. Shamir, and L Adleman. *A Method for Obtaining Digital Signatures and Public-Key Cryptosystems.* http://citeseer.ist.psu.edu/rivest78method.html, 1978.

[4] Eric W. Weisstein. *CRC Concise Encyclopedia of Mathematics.* CRC Press, 2nd edition, 1999.