ICR – Practical Work #3

Fault Attacks against RSA-CRT

Gugger Joël [1]

June 7, 2016

[1] joel.gugger@master.hes-so.ch

Abstract

A way to accelerate the RSA signature procedure consists in exploiting the fact that one knows the two primes p and q, as it is a private-key operation, and to use the Chinese Remainder Theorem (CRT).

The goal of this practical work consists in implementing a fast RSA signature procedure that exploits the CRT and to study the security of such an implementation at the light of fault attacks. This practical work can be implemented either in C, C++, Java or Python, with the big-numbers arithmetic library of your choice.

Contents

1 RSA-CRT **3**
1.1 Questions . . . 3
1.1.1 How have you tested that your routines are properly working? . . . 3
1.1.2 What is the gain in terms of speed that you obtain when using RSA-CRT with respect to a standard RSA signature generation procedure? . . . 3
1.1.3 What are the values that one could pre-compute and store besides n and d, in order to speed up as much as possible the signature generation procedure? . . . 3
1.2 Implementation . . . 4
1.2.1 RSA key generation routine . . . 4
1.2.2 Standard RSA signature and verification routines . . . 4
1.2.3 Fast RSA signature procedure . . . 7

2 Boneh-DeMillo-Lipton Attack **9**
2.1 Mathematical description . . . 9
2.2 Questions . . . 9
2.2.1 In practice, how is it possible to induce faults in cryptographic implementations? . . . 9
2.2.2 Is this attack working on a non-deterministic padding scheme? . . . 9
2.3 Simulating Boneh-DeMillo-Lipton attack . . . 9

3 Implementing Shamir's Trick **10**
3.1 Mathematical description . . . 10
3.2 Implementation . . . 10

Chapter 1

RSA-CRT

This part is dedicated to the implementation of a RSA-CRT fast signing procedure. One can assume that 1024-bit RSA keys[1] are used and that the digest formatting operation is performed elsewhere.

1.1 Questions

1.1.1 How have you tested that your routines are properly working?

Since three methods (standard signature, CRT signature and verification) have been implemented , verifying the proper operation was done using outputs methods as other inputs.

A test is a fact verifying the signature with the string used to generate the signature, then with another character string . The test must pass then fail if the implementation is correct.

1.1.2 What is the gain in terms of speed that you obtain when using RSA-CRT with respect to a standard RSA signature generation procedure?

To measure the speed increase, the same message was signed by each of the methods 10'000 times. The results obtained are as follows in seconds

```
Standard=28.407     CRT=7.996
```

The speed up is **3.5** times faster.

1.1.3 What are the values that one could pre-compute and store besides n and d, in order to speed up as much as possible the signature generation procedure?

To implement the RSA-CRT we need to pre-compute three values : d_p, d_q and q_{inv}. We also need to store p and q with, or instead of, n.

[1] I used the implementation with 2048-bit RSA Keys

1.2 Implementation

1.2.1 RSA key generation routine

The implementation is based on the corrected series of exercises one of the course. Modification were made to the structure of the private key to store the necessary variables for the RSA -CRT implementation.

```
32 typedef struct {
33     mpz_t p;
34     mpz_t q;
35     mpz_t dP;
36     mpz_t dQ;
37     mpz_t qInv;
38     mpz_t n;
39     mpz_t d;
40 } RSA_private_key_t;
```

For using the Chinese remainder algorithm, we need to compute three additional variables :

$$d_p = e^{-1} \pmod{p-1}$$

$$d_q = e^{-1} \pmod{q-1}$$

$$q_{inv} = q^{-1} \pmod{p}$$

To calculate these variables, the generation routine was modified. GMP library provides a built in method to make this calculs.

```
125         mpz_set (privkey->p, p);
126         mpz_set (privkey->q, q);
127 
128         /* d_P = e^-1  (mod p - 1)  */
129         mpz_invert (privkey->dP, pubkey->e, pm1);
130         TRACEVAR (privkey->dP, "dP");
131         /* d_Q = e^-1  (mod 1 - 1)  */
132         mpz_invert (privkey->dQ, pubkey->e, qm1);
133         TRACEVAR (privkey->dQ, "dQ");
134         /* q_Inv = q^-1  (mod p)  */
135         mpz_invert (privkey->qInv, privkey->q, p);
136         TRACEVAR (privkey->qInv, "qInv");
```

1.2.2 Standard RSA signature and verification routines

The RSA signature scheme works like this:

1. Creates a message digest of the information to be sent.
2. Represents this digest as an integer m between 1 and $n-1$.
3. Uses her private key (n, d) to compute the signature $s = m^d \pmod{n}$.
4. Sends this signature s to the recipient, B.

To implement this feature , I add a generate_textbookRSA_standard_signature method. The method takes the big number mpz_t, the data to be signed and the private key. An implementation of SHA256 is used to create the digest[2].

```
220 int generate_textbookRSA_standard_signature (
221     mpz_t s,
222     uchar data[],
223     const RSA_private_key_t *privkey)
224 {
225     /* Process of signing the message m */
226     /* it uses the secret key sk = (p, q, d) */
227     /* so that s = m^d (mod n) where n = p * q. */
228     assert(s != NULL);
229     assert(data != NULL);
230     assert(privkey != NULL);
231 
232     SHA256_CTX *sha256 = malloc (sizeof (SHA256_CTX));
233     uchar hash[32];
234     mpz_t m;
235 
236     mpz_inits (m, NULL);
237 
238     sha256_init(sha256);
239     sha256_update(sha256, data, strlen(data));
240     sha256_final(sha256, hash);
241 
242     mpz_import (m, sizeof (hash), 1, 1, 1, 0, hash);
243 
244     /* Computing s = m^d (mod n) */
245     mpz_powm (s, m, privkey->d, privkey->n);
246 
247     // TRACEVAR (s, "s");
248 
249     mpz_clears (m, NULL);
250     return EXIT_SUCCESS;
251 }
```

Implementing the RSA signature verification scheme is similar to the signature generation routine, pretty much the same code. The protocol works like this:

1. Uses sender A's public key (n, e) to compute integer $v = s^e \pmod{n}$.
2. Extracts the message digest from this integer.
3. Independently computes the message digest of the information that has been signed.
4. If both message digests are identical, the signature is valid.

[2] See http://bradconte.com/sha256_c.

The same SHA256 implementation is used. First we compute the data digest, and secondly we apply the formula:

$$v = s^e \pmod{n}$$

Finally the two results are compared. If they are of the same value, then the signature is correct and is really from the expected private key. Otherwise, the data were changed or private key is not good.

```
296 int verify_textbookRSA_standard_signature (
297     mpz_t s,
298     uchar data[],
299     RSA_public_key_t *pubkey)
300 {
301     /* Compute integer v = s^e (mod n). */
302     /* Extracts the message digest from this integer. */
303     /* Independently computes the message digest */
304     /* of the information that has been signed. */
305     /* If both message digests are identical, */
306     /* the signature is valid. */
307     assert(s != NULL);
308     assert(data != NULL);
309     assert(pubkey != NULL);
310 
311     SHA256_CTX *sha256 = malloc (sizeof (SHA256_CTX));
312     uchar hash[32];
313     mpz_t m, v;
314 
315     mpz_inits (m, v, NULL);
316 
317     sha256_init(sha256);
318     sha256_update(sha256, data, strlen(data));
319     sha256_final(sha256, hash);
320 
321     mpz_import (m, sizeof (hash), 1, 1, 1, 0, hash);
322 
323     /* Computing v = s^e (mod n) */
324     mpz_powm (v, s, pubkey->e, pubkey->n);
325 
326     TRACEVAR (v, "v");
327     TRACEVAR (m, "m");
328 
329     /* s^e (mod n) = Hash(m) (mod n) */
330     if ( mpz_cmp (v, m) ) {
331         mpz_clears (m, v, NULL);
332         return EXIT_SUCCESS;
333     } else {
334         mpz_clears (m, v, NULL);
335         return EXIT_FAILURE;
336     }
337 }
```

1.2.3 Fast RSA signature procedure

We can use the CRT to compute $m = c^d \pmod{n}$ more efficiently. To do this, we need to precompute the following values given p, q with $p > q$

$$d_p = e^{-1} \pmod{p-1}$$

$$d_q = e^{-1} \pmod{q-1}$$

$$q_{inv} = q^{-1} \pmod{p}$$

With these values, we can compute the message m given c with

$$m1 = c^{d_p} \pmod{p}$$

$$m2 = c^{d_q} \pmod{q}$$

$$h = q_{inv} * (m1 - m2) \pmod{p}$$

$$m = m2 + h * q$$

The RSA -CRT implementation is similar in all respects to the standard implementation. Only the calculation portion varies. Again, the GMP library provided us with the necessary methods for calculations on large numbers.

```
275     /* Computing s = m^d (mod n) with RSA-CRT */
276
277     /* m1 = c^dp (mod p) */
278     mpz_powm (m1, m, privkey->dP, privkey->p);
279     /* m2 = c^dq (mod q) */
280     mpz_powm (m2, m, privkey->dQ, privkey->q);
281     /* h = qinv * (m1 - m2) (mod p) */
282     mpz_sub (m1m2, m1, m2);
283     mpz_mul (qh, privkey->qInv, m1m2);
284     mpz_mod (h, qh, privkey->p);
285     /* m = m2 + h * q   */
286     mpz_mul (hq, h, privkey->q);
287     mpz_add (s, m2, hq);
```

For a given string of characters, both implementations provide the same signature. This signature is then verified by the function verify_textbookRSA _standard_signature. If the signature is not valide, an error message is output.

```
372     /* Key generation procedure */
373     generate_textbookRSA_keys (pubkey, privkey);
374
375     generate_textbookRSA_standard_signature (s, msg, privkey);
376
377     printf ("\nRSA-CRT signature:");
378     generate_textbookRSA_CRT_signature (s, msg, privkey);
379
380     if ( !verify_textbookRSA_standard_signature (s, msg, pubkey)
        ↪  ) {
381         fprintf (stderr, "\nError: signature not valid\n");
```

```
382     }
383
384     char msg2[] = "New message";
385
386     if ( !verify_textbookRSA_standard_signature (s, msg2,
        ↪  pubkey) ) {
387         fprintf (stderr, "\nError: signature not valid\n");
388     }
```

This code displays the following results. Both generation routine output the same result, the verification pass the first time, and the second verification, which has failed, failed. The behavior is as expected.

```
1  s : 768888fb6b65df66c8b0857cbc494728a8736c77d02bdc09\
2  461f48baba0219702737eb8ee4540caf6d2bf1d1b8fca7b1aae8\
3  1c3810f00712ef042417d4ebff335c05c6112c10ec9314b1a577d\
4  99084ff5974492161e9ba90f290dc592315980963323e41a89c7\
5  c18d9ad88a4e6bb1eb6e66b784e5b8a01d6eae657547d16849\
6  b751a9c5091326a3c2a3ed2f49caba7bc86e4e36c75ed44b4ed\
7  94320dc2b66c42531a6e0f25b8b0ef0d67d529adf6fdf05fe1d6ff\
8  0916f743da5b891903d893d66a4a34a5b5507b8e6bc7db34917\
9  b2c099bc30c0160377ac4b174ec5a696dc540f802b39b921805\
10 551da00bf861900e088199377c7d2c2cd4e827c879d9bb651
11
12 RSA-CRT signature:
13 s : 768888fb6b65df66c8b0857cbc494728a8736c77d02bdc094\
14 61f48baba0219702737eb8ee4540caf6d2bf1d1b8fca7b1aae81\
15 c3810f00712ef042417d4ebff335c05c6112c10ec9314b1a577d\
16 99084ff5974492161e9ba90f290dc592315980963323e41a89c\
17 7c18d9ad88a4e6bb1eb6e66b784e5b8a01d6eae657547d168\
18 49b751a9c5091326a3c2a3ed2f49caba7bc86e4e36c75ed44b\
19 4ed94320dc2b66c42531a6e0f25b8b0ef0d67d529adf6fdf05fe\
20 1d6ff0916f743da5b891903d893d66a4a34a5b5507b8e6bc7d\
21 b34917b2c099bc30c0160377ac4b174ec5a696dc540f802b39\
22 b921805551da00bf861900e088199377c7d2c2cd4e827c879\
23 d9bb651
24
25 v :
   ↪  b19318d0e9ba063a5fe94bc3cb9d9b5d79e06bfed220c86fb137cb40aef36140
26
27 m :
   ↪  b19318d0e9ba063a5fe94bc3cb9d9b5d79e06bfed220c86fb137cb40aef36140
28
29 v :
   ↪  b19318d0e9ba063a5fe94bc3cb9d9b5d79e06bfed220c86fb137cb40aef36140
30
31 m :
   ↪  78f5975a5d705e9528dd0e8d41206534b7e8c269b139bb151d5c0ca0928247c3
32
33 Error: signature not valid
```

Chapter 2

Boneh-DeMillo-Lipton Attack

In 1997, Boneh, DeMillo and Lipton have demonstrated that if a fault is induced during one of the two partial signature computation steps, that erroneous signature can be exploited in order to factor the public modulus.

2.1 Mathematical description

Task 2. *Describe in mathematical terms how the Boneh-DeMillo-Lipton fault attack against RSA-CRT is working.*

2.2 Questions

2.2.1 In practice, how is it possible to induce faults in cryptographic implementations?

2.2.2 Is this attack working on a non-deterministic padding scheme?

2.3 Simulating Boneh-DeMillo-Lipton attack

Task 3. *Write a program simulating Boneh-DeMillo-Lipton attack that allows to factor $n = p * q$ in a very efficient way.*

Chapter 3

Implementing Shamir's Trick

Several countermeasures have been proposed to defend against Boneh-DeMillo-Lipton attack. In this part, we will study and implement the one that is known as *Shamir's trick*. This technique essentially works as follows: the partial signatures are computed modulo rp and rq, where r is a small (i.e., 32-bit) random integer, instead of working modulo p and q, respectively.

3.1 Mathematical description

Task 4. *Describe in mathematical terms how Shamir's trick works.*

3.2 Implementation

Task 5. *Implement an RSA-CRT routine protected against Boneh-DeMillo-Lipton attack thanks to Shamir's trick.*

Abstract

The sources of the project are available on GitHub at the following address:
https://github.com/GuggerJoel/Crypto-ICR-lab003