

ICR – Practical Work #3

Fault Attacks against RSA-CRT

Gugger Joël ¹

June 9, 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?	10
2.3	Simulating Boneh-DeMillo-Lipton attack	10
3	Implementing Shamir's Trick	11
3.1	Mathematical description	11
3.2	Implementation	11

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 keys¹ 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

¹ | **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.

```

20 typedef struct {
21     mpz_t p;
22     mpz_t q;
23     mpz_t dP;
24     mpz_t dQ;
25     mpz_t qInv;
26     mpz_t n;
27     mpz_t d;
28 } 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.

```

113     mpz_set (privkey->p, p);
114     mpz_set (privkey->q, q);
115
116     /* d_P = e^{-1} (mod p - 1) */
117     mpz_invert (privkey->dP, pubkey->e, pm1);
118     TRACEVAR (privkey->dP, "dP");
119     /* d_Q = e^{-1} (mod 1 - 1) */
120     mpz_invert (privkey->dQ, pubkey->e, qm1);
121     TRACEVAR (privkey->dQ, "dQ");
122     /* q_Inv = q^{-1} (mod p) */
123     mpz_invert (privkey->qInv, privkey->q, p);
124     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 digest².

```

208 int generate_textbookRSA_standard_signature (
209     mpz_t s,
210     uchar data[],
211     const RSA_private_key_t *privkey)
212 {
213     /* Process of signing the message m */
214     /* it uses the secret key sk = (p,q,d) */
215     /* so that s = m^d (mod n) where n = p*q. */
216     assert(s != NULL);
217     assert(data != NULL);
218     assert(privkey != NULL);
219
220     SHA256_CTX *sha256 = malloc (sizeof (SHA256_CTX));
221     uchar hash[32];
222     mpz_t m;
223
224     mpz_inits (m, NULL);
225
226     sha256_init(sha256);
227     sha256_update(sha256, data, strlen(data));
228     sha256_final(sha256, hash);
229
230     mpz_import (m, sizeof (hash), 1, 1, 1, 0, hash);
231
232     /* Computing s = m^d (mod n) */
233     mpz_powm (s, m, privkey->d, privkey->n);
234
235     // TRACEVAR (s, "s");
236
237     mpz_clears (m, NULL);
238     return EXIT_SUCCESS;
239 }

```

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 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.

```

284 int verify_textbookRSA_standard_signature (
285     mpz_t s,
286     uchar data[],
287     RSA_public_key_t *pubkey)
288 {
289     /* Compute integer  $v = s^e \pmod{n}$ . */
290     /* Extracts the message digest from this integer. */
291     /* Independently computes the message digest */
292     /* of the information that has been signed. */
293     /* If both message digests are identical, */
294     /* the signature is valid. */
295     assert(s != NULL);
296     assert(data != NULL);
297     assert(pubkey != NULL);
298
299     SHA256_CTX *sha256 = malloc (sizeof (SHA256_CTX));
300     uchar hash[32];
301     mpz_t m, v;
302
303     mpz_inits (m, v, NULL);
304
305     sha256_init(sha256);
306     sha256_update(sha256, data, strlen(data));
307     sha256_final(sha256, hash);
308
309     mpz_import (m, sizeof (hash), 1, 1, 1, 0, hash);
310
311     /* Computing  $v = S^e \pmod{n}$  */
312     mpz_powm (v, s, pubkey->e, pubkey->n);
313
314     TRACEVAR (v, "v");
315     TRACEVAR (m, "m");
316
317     /*  $S^e \pmod{n} = Hash(m) \pmod{n}$  */
318     if ( mpz_cmp (v, m) ) {
319         mpz_clears (m, v, NULL);
320         return EXIT_SUCCESS;
321     } else {
322         mpz_clears (m, v, NULL);
323         return EXIT_FAILURE;
324     }
325 }

```

1.2.3 Fast RSA signature procedure

We can use the CRT to compute $S = m^d \pmod{n}$ 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 signature S given m with

$$M_1 = m^{d_p} \pmod{p}$$

$$M_2 = m^{d_q} \pmod{q}$$

$$h = q_{inv} * (M_1 - M_2) \pmod{p}$$

$$S = M_2 + h * q$$

The RSA-CRT implementation is similar in all respects to the standard implementation. Only the 12 lines varies. Again, the GMP library provided us with the necessary methods for calculations on large numbers.

```

263      /* Computing  $S = m^d \pmod{n}$  with RSA-CRT */
264
265      /*  $M_1 = m^{d_p} \pmod{p}$  */
266      mpz_powm (m1, m, privkey->dP, privkey->p);
267      /*  $M_2 = m^{d_q} \pmod{q}$  */
268      mpz_powm (m2, m, privkey->dQ, privkey->q);
269      /*  $h = q_{inv} * (M_1 - M_2) \pmod{p}$  */
270      mpz_sub (m1m2, m1, m2);
271      mpz_mul (qh, privkey->qInv, m1m2);
272      mpz_mod (h, qh, privkey->p);
273      /*  $S = M_2 + h * q$  */
274      mpz_mul (hq, h, privkey->q);
275      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.

```

40      /* Key generation procedure */
41      generate_textbookRSA_keys (pubkey, privkey);
42
43      generate_textbookRSA_standard_signature (s, msg, privkey);
44
45      printf ("\nRSA-CRT signature:");
46      generate_textbookRSA_CRT_signature (s, msg, privkey);
47
48      if ( !verify_textbookRSA_standard_signature (s, msg, pubkey)
         ↪ ) {

```



```

49         fprintf (stderr, "\nError: signature not valid\n");
50     }
51
52     char msg2[] = "New message";
53
54     if ( !verify_textbookRSA_standard_signature (s, msg2,
55         ↪ pubkey) ) {
56         fprintf (stderr, "\nError: signature not valid\n");
57     }

```

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 :
26 ↪ b19318d0e9ba063a5fe94bc3cb9d9b5d79e06bfed220c86fb137cb40aef36140
27
28 m :
29 ↪ b19318d0e9ba063a5fe94bc3cb9d9b5d79e06bfed220c86fb137cb40aef36140
30
31 v :
32 ↪ b19318d0e9ba063a5fe94bc3cb9d9b5d79e06bfed220c86fb137cb40aef36140
33
34 m :
35 ↪ 78f5975a5d705e9528dd0e8d41206534b7e8c269b139bb151d5c0ca0928247c3
36
37 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.*

This attack is available when a fault appears when exactly one of two M_1 or M_2 will be computed incorrectly. If M_1 is correct, but \widehat{M}_2 is not. The resulting signature is $\widehat{S} = \widehat{M}_2 + h * q$, when Bob receives the signature \widehat{S} , he knows it is a false signature since $\widehat{S}^e \neq h(m) \pmod{n}$.

QED: As $\widehat{S}^e \equiv h(m) \pmod{p}$ but $\widehat{S}^e \not\equiv h(m) \pmod{q}$ we can factorize n by $p = \text{GCD}(S^e - h(m), n)$.

2.2 Questions

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

Heavily inspired by the article "The Sorcerer's Apprentice Guide to Fault Attacks"¹.

Most of methods required to have a physical access on the device. Not a complete access, but it must be possible to interfere with the environment. Like this, we can change the temperature of the chip, variate the supply voltage to skip instructions, or variate the external clock to cause data miss reading.

¹ <https://eprint.iacr.org/2004/100.pdf>

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

No, this attack applies only to any deterministic padding function μ , such as RSA PKCS#1 v1.5 or Full-Domain Hash.

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.*

The article² that I used for implementing the attack against RSA-CRT.

² <https://crypto.stanford.edu/~dabo/papers/RSA-survey.pdf>

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`