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

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 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?

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?

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?

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.

```
typedef struct {
    mpz_t p;
    mpz_t q;
    mpz_t dP;
    mpz_t dQ;
    mpz_t qInv;
    mpz_t n;
    mpz_t d;
} 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.

```
        mpz_set (privkey->p, p);
        mpz_set (privkey->q, q);

        /* d_P = e^{-1} (mod p - 1) */
        mpz_invert (privkey->dP, pubkey->e, pm1);
        TRACEVAR (privkey->dP, "dP");
        /* d_Q = e^{-1} (mod 1 - 1) */
        mpz_invert (privkey->dQ, pubkey->e, qm1);
        TRACEVAR (privkey->dQ, "dQ");
        /* q_Inv = q^{-1} (mod p) */
        mpz_invert (privkey->qInv, privkey->q, p);
        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[1].

```
int generate_textbookRSA_standard_signature (
    mpz_t s,
    uchar data[],
    const RSA_private_key_t *privkey)
{
    /* Process of signing the message m */
    /* it uses the secret key sk = (p, q, d) */
    /* so that s = m^d (mod n) where n = p * q. */
    assert(s != NULL);
    assert(data != NULL);
    assert(privkey != NULL);
```

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

```

    SHA256_CTX *sha256 = malloc (sizeof (SHA256_CTX));
    uchar hash[32];
    mpz_t m;

    mpz_inits (m, NULL);

    sha256_init(sha256);
    sha256_update(sha256, data, strlen(data));
    sha256_final(sha256, hash);

    mpz_import (m, sizeof (hash), 1, 1, 1, 0, hash);

    /* Computing s = m^d (mod n) */
    mpz_powm (s, m, privkey->d, privkey->n);

    TRACEVAR (s, "s");

    mpz_clears (m, NULL);
    return EXIT_SUCCESS;
}
```

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.

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.

```
int verify_textbookRSA_standard_signature (
    mpz_t s,
    uchar data[],
    RSA_public_key_t *pubkey)
{
    /* Compute integer v = s^e (mod n). */
    /* Extracts the message digest from this integer. */
    /* Independently computes the message digest */
```

```
    /* of the information that has been signed. */
    /* If both message digests are identical, */
    /* the signature is valid. */
    assert(s != NULL);
    assert(data != NULL);
    assert(pubkey != NULL);

    SHA256_CTX *sha256 = malloc (sizeof (SHA256_CTX));
    uchar hash[32];
    mpz_t m, v;

    mpz_inits (m, v, NULL);

    sha256_init(sha256);
    sha256_update(sha256, data, strlen(data));
    sha256_final(sha256, hash);

    mpz_import (m, sizeof (hash), 1, 1, 1, 0, hash);

    /* Computing v = s^e (mod n) */
    mpz_powm (v, s, pubkey->e, pubkey->n);

    TRACEVAR (v, "v");
    TRACEVAR (m, "m");

    /* s^e (mod n) = Hash(m) (mod n) */
    if ( mpz_cmp (v, m) ) {
        mpz_clears (m, v, NULL);
        return EXIT_SUCCESS;
    } else {
        mpz_clears (m, v, NULL);
        return EXIT_FAILURE;
    }
}
```

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

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

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

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

$$m = m2 + h * q$$

```
int generate_textbookRSA_CRT_signature (
    mpz_t s,
    uchar data[],
    const RSA_private_key_t *privkey)
{
    assert(s != NULL);
    assert(data != NULL);
    assert(privkey != NULL);

    SHA256_CTX *sha256 = malloc (sizeof (SHA256_CTX));
    uchar hash[32];
    mpz_t m, m1, m2, h, hq, qh, m1m2;

    mpz_inits (m, m1, m2, h, hq, qh, m1m2, NULL);

    sha256_init(sha256);
    sha256_update(sha256, data, strlen(data));
    sha256_final(sha256, hash);

    mpz_import (m, sizeof (hash), 1, 1, 1, 0, hash);

    /* Computing s = m^d (mod n) with RSA-CRT */

    /* m1 = c^{d_p} (mod p) */
    mpz_powm (m1, m, privkey->dP, privkey->p);
    /* m2 = c^{d_q} (mod q) */
    mpz_powm (m2, m, privkey->dQ, privkey->q);
    /* h = q_{inv} * (m1 - m2) (mod p) */
    mpz_sub (m1m2, m1, m2);
    mpz_mul (qh, privkey->qInv, m1m2);
    mpz_mod (h, qh, privkey->p);
    /* m = m2 + h * q */
```

```
    mpz_mul (hq, h, privkey->q);
    mpz_add (s, m2, hq);

    TRACEVAR (s, "s");

    mpz_clears (m, m1, m2, h, hq, qh, m1m2, NULL);
    return EXIT_SUCCESS;
}
```

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