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

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.

```
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 :

$$\begin{aligned} d_p &\equiv e^{-1} \pmod{p-1} \\ d_q &\equiv e^{-1} \pmod{q-1} \\ q_{inv} &\equiv q^{-1} \pmod{p} \end{aligned}$$

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[2].

```
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);

    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 1;
}
```

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 $\langle n, e \rangle$ to compute integer $v \equiv S^e \pmod{n}$.
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

$$v \equiv S^e \pmod{n}$$

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) == 0 ) {
        mpz_clears (m, v, NULL);
        return 1;
    } else {
        mpz_clears (m, v, NULL);
        return 0;
    }
}
```

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 $p > q$

$$\begin{aligned} d_p &\equiv e^{-1} \pmod{p-1} \\ d_q &\equiv e^{-1} \pmod{q-1} \\ q_{inv} &\equiv q^{-1} \pmod{p} \end{aligned}$$

With these values, we can compute the signature S given m with

$$\begin{aligned} M_1 &\equiv m^{d_p} \pmod{p} \\ M_2 &\equiv m^{d_q} \pmod{q} \\ h &\equiv q_{inv} * (M_1 - M_2) \pmod{p} \\ S &= M_2 + h * q \end{aligned}$$

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.

```c
    /* Computing S = m^d (mod n) with RSA-CRT */

    /* M_1 = m^{d_p} (mod p) */
    mpz_powm (m1, m, privkey->dP, privkey->p);
    /* M_2 = m^{d_q} (mod q) */
    mpz_powm (m2, m, privkey->dQ, privkey->q);
    /* h = q_{inv} * (M_1 - M_2) (mod p) */
    mpz_sub (m1m2, m1, m2);
    mpz_mul (qh, privkey->qInv, m1m2);
    mpz_mod (h, qh, privkey->p);
    /* S = M_2 + h * q */
    mpz_mul (hq, h, privkey->q);
    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.

```c
    /* Key generation procedure */
    generate_textbookRSA_keys (pubkey, privkey);

    generate_textbookRSA_standard_signature (s, msg, privkey);

    printf ("\nRSA-CRT signature:");
    generate_textbookRSA_CRT_signature (s, msg, privkey);

    if ( !verify_textbookRSA_standard_signature (s, msg, pubkey) ) {
        fprintf (stderr, "\nError: RSA-CRT signature not valid\n");
    }

```

```
    char msg2[] = "New message";

    if ( !verify_textbookRSA_standard_signature (s, msg2,
     ↪  pubkey) ) {
        fprintf (stderr, "\nError: signature not valid with
         ↪  msg2\n");
    }
```

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.

```
s : 768888fb6b65df66c8b0857cbc494728a8736c77d02bdc09\
461f48baba0219702737eb8ee4540caf6d2bf1d1b8fca7b1aae8\
1c3810f00712ef042417d4ebff335c05c6112c10ec9314b1a577d\
99084ff5974492161e9ba90f290dc592315980963323e41a89c7\
c18d9ad88a4e6bb1eb6e66b784e5b8a01d6eae657547d16849\
b751a9c5091326a3c2a3ed2f49caba7bc86e4e36c75ed44b4ed\
94320dc2b66c42531a6e0f25b8b0ef0d67d529adf6fdf05fe1d6ff\
0916f743da5b891903d893d66a4a34a5b5507b8e6bc7db34917\
b2c099bc30c0160377ac4b174ec5a696dc540f802b39b921805\
551da00bf861900e088199377c7d2c2cd4e827c879d9bb651

RSA-CRT signature:
s : 768888fb6b65df66c8b0857cbc494728a8736c77d02bdc094\
61f48baba0219702737eb8ee4540caf6d2bf1d1b8fca7b1aae81\
c3810f00712ef042417d4ebff335c05c6112c10ec9314b1a577d\
99084ff5974492161e9ba90f290dc592315980963323e41a89c\
7c18d9ad88a4e6bb1eb6e66b784e5b8a01d6eae657547d168\
49b751a9c5091326a3c2a3ed2f49caba7bc86e4e36c75ed44b\
4ed94320dc2b66c42531a6e0f25b8b0ef0d67d529adf6fdf05fe\
1d6ff0916f743da5b891903d893d66a4a34a5b5507b8e6bc7d\
b34917b2c099bc30c0160377ac4b174ec5a696dc540f802b39\
b921805551da00bf861900e088199377c7d2c2cd4e827c879\
d9bb651

RSA-CRT signature:
v :
 ↪  b19318d0e9ba063a5fe94bc3cb9d9b5d79e06bfed220c86fb137cb40aef36140
m :
 ↪  b19318d0e9ba063a5fe94bc3cb9d9b5d79e06bfed220c86fb137cb40aef36140
v :
 ↪  b19318d0e9ba063a5fe94bc3cb9d9b5d79e06bfed220c86fb137cb40aef36140
m :
 ↪  78f5975a5d705e9528dd0e8d41206534b7e8c269b139bb151d5c0ca0928247c3

Error: signature not valid with msg2
```

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 appaires 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 interfer with the environement. 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.*

I was inspired by the article "Twenty Years of Attacks on the RSA Cryptosystem"[2] to implement the attack against RSA-CRT.

To easily induce a mistake when signing, I implemented a method for generating an improper rangeland signature.

```c
int generate_fault_RSACRT_signature (mpz_t, uchar[], const RSA_private_key_t *);
```

This method repeats the same operation as the RSA-CRT implementation. A line has been added to induce the fault.

```c
    /* Computing S = m^d (mod n) with RSA-CRT */

    /* M_1 = m^{d_p} (mod p) */
    mpz_powm (m1, m, privkey->dP, privkey->p);
    /* M_2 = m^{d_q} (mod q) */
    mpz_powm (m2, m, privkey->dQ, privkey->q);

    /* Induce a fault in M_2 */
    mpz_sub_ui (m2, m2, 1);
```

By subtracting 1 to the variable M_2, we induce the fault for having

$$\begin{cases} \widehat{S}^e & \not\equiv & h(m) \pmod n \\ \widehat{S}^e & \equiv & h(m) \pmod p \\ \widehat{S}^e & \not\equiv & h(m) \pmod q \end{cases}$$

[2] https://crypto.stanford.edu/~dabo/papers/RSA-survey.pdf

The implementation of the exploit is simple. We sign with the generate_fault_RSACRT_signature method, we check if the signature is correct, when is not we make the calculations to retreive p. And when we have p, we can compute $q = n/p$.

```c
    generate_fault_RSACRT_signature (s, msg, privkey);

    if ( !verify_textbookRSA_standard_signature (s, msg, pubkey) ) {
        fprintf (stderr, "\nError: signature not valid\n");
        fprintf (stderr, "Attack begin...\n");

        /* Make p = GCD(S^e - h(m),n) */
        mpz_powm (se, s, pubkey->e, pubkey->n);
        mpz_sub (se, se, h);
        mpz_gcd (p, se, pubkey->n);
        mpz_cdiv_q (q, pubkey->n, p);

        TRACEVAR (p, "p");
        TRACEVAR (privkey->p, "privkey->p");

        TRACEVAR (q, "q");
        TRACEVAR (privkey->q, "privkey->q");

        if ( mpz_cmp (p, privkey->p) == 0 &&
             mpz_cmp (q, privkey->q) == 0 ) {
            printf("\n**********************");
            printf("\n* Successful attack! *");
            printf("\n**********************\n");
        } else {
            printf("\n**********************");
            printf("\n*   Failed attack!   *");
            printf("\n**********************\n");
        }
    }
```

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.*

Let $r \in \mathbb{R} \in \{0,1\}^{32}$ a random prime number with

$$\begin{aligned} S_{rp} &= m^{d \bmod \varphi(p \cdot r)} \pmod{p \cdot r} \\ S_{rq} &= m^{d \bmod \varphi(q \cdot r)} \pmod{q \cdot r} \end{aligned}$$

with

$$\begin{aligned} \varphi(p \cdot r) &= (p-1)(r-1) \\ \varphi(q \cdot r) &= (q-1)(r-1) \end{aligned}$$

If $S_{rp} \equiv S_{rq} \pmod{r}$, then both partial signature are correct, we can return $S = CRT(S_{rp}, S_{rq})$. Else, $S_{rp} \not\equiv S_{rq} \pmod{r}$, an error has occurred and we must return an error or retry.

To compute $S = CRT(S_{rp}, S_{rq})$ we need to recover S_p and S_q

$$\begin{aligned} S_p &\leftarrow S_{rp} \pmod{p} \\ S_q &\leftarrow S_{rq} \pmod{q} \end{aligned}$$

And recombine S_p and S_q as explained previously to get the signature S with the CRT

$$\begin{aligned} h &\equiv q_{inv} * (S_p - S_q) \pmod{p} \\ S &= S_q + h * q \end{aligned}$$

3.2 Implementation

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

A new method was created to implement the Shamir's Trick.

```c

int generate_RSACRT_signature_shamir (mpz_t, uchar[], const RSA_private_key_t *);
```

We need a random prime number $r \in \mathbb{R} \in \{0,1\}^{32}$. To do this we read 4 bytes on the /dev/urandom device, we convert the result to a big number with GMP and we search the next prime.

```c
    do {
        if ( read (fd, rnd, 4) != 4 ) {
            perror ("Error: impossible to read enough random bytes");
            /* Don't forget to close the file descriptor */
            if ( close (fd) ) {
                perror ("Error: impossible to close the randomness source");
            }
            return EXIT_FAILURE;
        }

        mpz_import (r, 4, 1, 1, 1, 0, rnd);
        mpz_nextprime (r, r);
```

All calculations are done with GMP. While $S_{rp} \not\equiv S_{rq} \pmod{r}$, the loop continue, we read a new random prime number and make calculations.

```c
        mpz_sub_ui (rm1, r, 1);
        /* $S_{rp} = m^{d \bmod \varphi(p \cdot r)} \pmod{p \cdot r}$ */
        mpz_mul (pr, privkey->p, r);
        mpz_sub_ui (pm1, privkey->p, 1);
        mpz_mul (pm1rm1, pm1, rm1);
        mpz_mod (dp, privkey->d, pm1rm1);
        mpz_powm (sp, m, dp, pr);
        /* $S_{rq} = m^{d \bmod \varphi(q \cdot r)} \pmod{q \cdot r}$ */
        mpz_mul (qr, privkey->q, r);
        mpz_sub_ui (qm1, privkey->q, 1);
        mpz_mul (qm1rm1, qm1, rm1);
        mpz_mod (dq, privkey->d, qm1rm1);
        mpz_powm (sq, m, dq, qr);
        /* $S_{rp} \equiv S_{rq} \pmod{r}$ */
        mpz_mod (tsp, sp, r);
        mpz_mod (tsq, sq, r);
    } while (mpz_cmp (tsp, tsq) != 0);
```

If $S_{rp} \equiv S_{rq} \pmod{r}$, we retrieve S_p and S_q and return $S = CRT(S_{rp}, S_{rq})$.

```
    mpz_mod (sp, sp, privkey->p);
    mpz_mod (sq, sq, privkey->q);

    /* h = q_inv * (S_p - S_q) (mod p) */
    mpz_sub (m1m2, sp, sq);
    mpz_mul (qh, privkey->qInv, m1m2);
    mpz_mod (h, qh, privkey->p);
    /* S = S_q + h * q */
    mpz_mul (hq, h, privkey->q);
    mpz_add (s, sq, hq);
```

Abstract

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