HAW Hamburg

IE7 - Introduction to Cryptography

Laboratory Report

Jerome Smith
Thishan Warnakulasooriya

supervised by
Prof. Dr. Heike Neumann

February 8th, 2023

Contents

1 Introduction and Task Description

1.1 RSA Algorithm

The RSA algorithm is named after its inventors Rivest, Shamir and Adleman and is one of the most famous examples of public key cryptography. Public key cryptography is characterized by the presence of a private as well as a public key which can be shared publicly without compromising the security of the system. This security is based on the fact that while multiplication of (large) numbers is fairly easy, the inverse direction - factorization of the product back to its two multiplicands (especially prime numbers) - is extremely difficult and until now no algorithm is known that is able to do that for large numbers with the use of classical computers within a reasonable amount of time [3] .

1.2 Laboratory Task

During this laboratory experiment, and RSA public key cryptography algorithm will be implemented. The implementation is supposed to be configurable for key lengths of 1024, 2048 and 4096 bits. Additionally, users will be able to decide whether decryption is to be performed with or without the use of the Chinese Remainder Theorem (CRT).

An evaluation of the security of the implementation, including algorithm verification and robustness as well as performance tests is to be conducted.

2 Implementation

2.1 General

Since there were no specific requirements given on a programming language and the security of the system is planned to be tested via power analysis side-channel attacks on a microcontroller, it was decided to implement the RSA algorithm in the C programming language. As the most recent standard of the C language (C99) is only able to handle integers of bit sizes up to 64 bits (`long long int` or `uint64_t` resp. `int64_t`), a special library for the handling with larger integers had to be used. The FLINT library [6] was chosen for this purpose as it offers the required capabilities and provides dedicated functions for arithmetic operations on such large numbers. In particular, the required addition, subtraction, multiplication and modulo operations are already implemented.

2.2 Private Key Generation

```
1       void vGeneratePrivateKey(fmpz_t d, fmpz_t a, fmpz_t b, fmpz_t e)
2       {
3         clock_t start_time = clock();
4
5         fmpz_t v, p_m, q_m, phi, one, result;
6         fmpz_init(v);
7         fmpz_init(p_m);
8         fmpz_init(q_m);
9         fmpz_init(phi);
10        fmpz_init(result);
11        fmpz_init(one);
12        fmpz_one(one);
13
14        // Computes phi(n) = (p-1)*(q-1)
15        fmpz_sub(p_m, a, one);
16        fmpz_sub(q_m, b, one);
17        fmpz_mul(phi, p_m, q_m);
18
19        // computes values a and b such that d*phi + v*e = 1, where 1 =
    gcd(e, phi)
20        fmpz_xgcd(one, d, v, e, phi);
21        fmpz_print(d);
22
23        // Ensure that d is positive
24        int counter = 0;
25        while (fmpz_sgn(d) < 0)
26        {
27          counter++;
28          fmpz_add(d, d, phi);
29        }
30        clock_t end_time = clock();
31        double elapsed_time = ((double)(end_time - start_time)) /
    CLOCKS_PER_SEC;
32      }
33
```

Listing 1: Implementation of RSA key generation

The algorithm for RSA key generation shown in Listing 1 follows the algorithm provided by the lecture slides [3]. After having calculated $phi(n) = (p-1) \cdot (q-1)$, the private key is obtained by also using the value of 0x10001 (public key) and inputting them into the extended euclidean algorithm with the condition of $\gcd(\phi(n), e) = 1$. The algorithm will then compute the private key d. It is important to note, that d must be a positive value. If the result of the algorithm should be negative, the value of $\phi(n)$ has to be added to d until it is positive.

2.3 Encryption and Decryption

2.3.1 Square and Multiply

```
1   void __attribute__((optimize(0))) vSquareAndMultiply(fmpz_t result,
    const fmpz_t base, const fmpz_t exponent, const fmpz_t modulus)
2   {
3     fmpz_t dummy_result;
4     fmpz_init(dummy_result);
5     fmpz_set_ui(result, 1u);
6
7     // Find the number of bits in the exponent
8     int length = fmpz_sizeinbase(exponent, 2);
9
10    for (int i = length - 1; i >= 0; i--)
11    {
12      // result = (result * result) % modulus
13      fmpz_mul(result, result, result);
14      fmpz_mod(result, result, modulus);
15
16      if (1 == fmpz_tstbit(exponent, i))
17      {
18        // Multiplication result * base if current exponent bit == 1
19        // result = (result * base) % modulus
20        fmpz_mul(result, result, base);
21        fmpz_mod(result, result, modulus); // Modulus step
22      }
23      else if (0 == fmpz_tstbit(exponent, i))
24      {
25        // Dummy multiplication if current exponent bit == 0
26        // dummy_result = (result * base) % modulus
27        fmpz_mul(dummy_result, result, base);
28        fmpz_mod(dummy_result, dummy_result, modulus); // Modulus step
29      }
30    }
31    fmpz_clear(dummy_result);
32  }
33
```

Listing 2: Implementation of Square and Multiply algorithm in C

The implementation of the Square and Multiply algorithm is again based on the version that has been introduced during the lecture. There are some slight modifications to enhance the robustness against certain attacks which will be addressed in the *Security* chapter (chapter 3).

2.4 Chinese Remainder Theorem

```
1   void vChineseRemainderTheorem(fmpz_t result, fmpz_t m, const fmpz_t
    p, const fmpz_t q, const fmpz_t d, const fmpz_t modulus)
2   {
3     fmpz_t dp, dq, u, v, one, tmpP, tmpQ, pPart, qPart, qRes, pRes;
4
5     fmpz_sub(tmpP, p, one);    // tmpP = p - 1
6     fmpz_mod(dp, d, tmpP);     // dp = d mod (p - 1)
7
8     fmpz_sub(tmpQ, q, one);    // tmpQ = q - 1
9     fmpz_mod(dq, d, tmpQ);     // dq = d mod (q - 1)
10
11    fmpz_xgcd(one, u, v, p, q); // 1 = u*p + v*q
12
13    vSquareAndMultiply(pPart, m, dp, p);  // pPart = m^dp mod p
14    vSquareAndMultiply(qPart, m, dq, q);  // qPart = m^dq mod q
15
16
17    fmpz_mul(pRes, u, p); // u * p
18    fmpz_mul(qRes, v, q); // v * q
19
20    // result = (u*p*qPart + v*q*pPart ) mod n
21    fmpz_fmma(result, pRes, qPart, qRes, pPart);
22    fmpz_mod(result, result, modulus);
23
24  }
25
```

Listing 3: Implementation of the Chinese Remainder Theorem algorithm

Listing 3 shows the implementation of the decryption using the Chinese Remainder Theorem algorithm. It takes advantage of the fact that the person who is decrypting the message does not only have knowledge of the private key, but also of the prime numbers (p and q) that were used to compute the modulo. As p and q have only half the number of bits of their product n, the whole decrypting process can be made much more efficient. [3][4].

3 Security

3.1 Side-Channel-Attacks

Side-Channel attacks are attacks on a cryptographic system that focus on observing the physical properties (like power consumption or duration) of a system over the course of time. Sometimes from the change of these properties it is possible to detect certain pattern that in turn allow the deduction of sensitive cryptographic information.
Certain parts of the RSA algorithm are especially vulnerable against specific side-channel attacks. A classic example for that is the square and multiply algorithm. Since in its basic form the number and order of modular multiplications is dependent of the bit pattern of the exponent, it is in danger of leaking sensitive information like the private key through power analysis or timing attacks.
Listing 2 lines 23 to 29 show how this problem was addressed in the used implementation. Even though there should not be an additional modular multiplication if the current exponent bit is 0, the algorithm still performs modular multiplication and stores the result in a *dummy* variable. This variable is not used and discarded at the end. The important fact is that for each bit of the exponent (no matter if its a 0 or a 1), the code execution (and thus the power consumption and execution time) will be exactly the same.

3.2 Fault-Injections

In Fault-Injection attacks, the attacker intentionally introduces errors (faults) in the code execution. Examples for this kind of attack are the flipping of a single bit or a attack that would cause the processor to skip a specific instruction. These kinds of attacks would for example be especially powerful against naive implementations of RSA signature verifications.

```
1       bool verification_failed = false;
2       verification_failed = RSA_verifySignature(signature, publicKey);
3
4       if(true == verification_failed)
5       {
6         while(1);
7       }
8
9       main();
10
```

Listing 4: Flawed signature verfication process of a generic secure boot

Listing 4 shows such a RSA signature verification process, that would be extremely vulnerable against Fault-Injection attacks. Since the whole process is designed in a way that a successful signature verification is the default case and it is only checked if that is

not the case, a skipping of the instruction of line 2 (calling of the signature verification algorithm) would allow the system to boot up normally as if integrity and authenticity were successfully verified. Another way to attack this implementation would be a bit flipping Fault-Injection attack. Since the boolean data type in C is basically just a variable of type `char` (`uint8_t`) which interprets the value 0 as *false* and every other value as *true*, flipping any of the 8 zero bits of a *false* would flip the boolean value from *false* to *true*. A counter measurement against this is the creation of a new, more secure boolean datatype.

```
1       typedef enum
2       {
3         SECURE_FALSE = 0xACEFAECFu, // 0b10101100111011111010111011001111
4         SECURE_TRUE = 0xCA38C3E8u   // 0b11001010001110001100001111101000
5       } secureBool_t;
6
```

Listing 5: Secure boolean type definition [5]

If `SECURE_FALSE` and `SECURE_TRUE` of `secureBool_t` are now used instead of simple `false` and `true` of type `bool` a single bit flip will not invert the boolean value. If the algorithm also assumes a default verification fail until the signature is successfully verified instead of the other way around, the security of the cryptographic system is significantly enhanced.

4 Tests

4.1 Algorithm Verification

In order to test and verify the validity and robustness of the RSA implementation, test vectors supplied by the National institute of Standards and Technology [1] were used. For each key size (1024, 2048, 4096) 10 tests were executed as this is listed as an requirement by the NIST.
The first 10 test vectors [2] for the modulus have been used as test cases.

Test Case	Bit size	Result	expected Result
1	1024	*passed*	*passed*
2	1024	*passed*	*passed*
3	1024	*failed*	*failed*
4	1024	*passed*	*passed*
5	1024	*passed*	*passed*
6	1024	*passed*	*passed*
7	1024	*passed*	*passed*
8	1024	*passed*	*passed*
9	1024	*failed*	*failed*
10	2048	*passed*	*passed*
11	2048	*passed*	*passed*
12	2048	*failed*	*failed*
13	2048	*failed*	*failed*
14	2048	*passed*	*passed*
15	2048	*passed*	*passed*
16	2048	*passed*	*passed*
17	2048	*passed*	*passed*
18	2048	*passed*	*passed*
19	2048	*passed*	*passed*
20	2048	*passed*	*passed*

Figure 1: Algorithm verification tests [2]

4.2 Performance

The reason for implementing a *Chinese Remainder Theorem* version of the RSA decryption was to increase the efficiency and thus the speed of the algorithm. Using a 2048 bit length this performance increase was measured.

Test Number	Duration of Key Generation [μs]	Duration of Decryption [μs]
1	26	5208
2	60	5562
3	25	5246
4	26	5200
5	31	6533
6	25	5335
7	25	6027
8	27	5197
9	31	6103
10	27	5194

Figure 2: Performance measurements for 2048 bit size and CRT off

Test Number	Duration of Key Generation [μs]	Duration of Decryption [μs]
1	31	2321
2	57	2909
3	61	2999
4	30	3807
5	26	2310
6	26	2305
7	26	2308
8	27	2325
9	26	2313
10	28	2305

Figure 3: Performance measurements for 2048 bit size and CRT on

The measured values result in the following mean values: It can clearly be seen that

Quantity	Value
$\bar{t}_{keyGen}$	$32.05\,\mu$s
$\bar{t}_{decrypt}$	$5560.5\,\mu$s
$\bar{t}_{decrypt,CRT}$	$2590.2\,\mu$s

Figure 4: Mean values of measurements

the introduction of the CRT in the decryption process significantly increases the speed of the algorithm. In the given case the RSA decryption was executed on average more than twice as fast.

4.3 Power Analysis Side-Channel Attack

The Side-Channel power analysis attack was performed on a LPC55S16 microcontroller by NXP. A shunt resistor (10 Ω) was put between the power supply and the microcontroller. Accordings to Ohm's law the current flowing through the resistor is proportional to the voltage drop ($U = R \cdot I$), thus an increase in voltage drop across the resistor also indicates an increase in power consumption. For the used tool chain (MCUXpresso by NXP) the code had to be adapted. Instead of the flint library, the mbedTLS library was used for this specific test.

```
1   void __attribute((optimize(0)))  powerAnalysis()
2   {
3     mbedtls_mpi_mul_mpi(&modulus, &p, &q);
4
5     uint32_t exponent = 0b10101;
6
7     // length = 32 - "leading zeros before most significant 1"
8     uint8_t  length = (32 - __CLZ(exponent));
9
10    while(1)
11    {
12      mbedtls_mpi_lset(&result, 1);
13      GPIO->SET[0] = (1 << 1); // Set GPIO Port 0 Pin 1 to HIGH
14      for(int i = length - 1; i >= 0; i--)
15      {
16        mbedtls_mpi_mul_mpi(&result, &result, &result);
17        mbedtls_mpi_mod_mpi(&result, &result, &modulus);
18
19        if(exponent & (1 << i))
20        {
21          mbedtls_mpi_mul_mpi(&result, &result, &base);
22          mbedtls_mpi_mod_mpi(&result, &result, &modulus);
23        }
24      }
25      GPIO->CLR[0] = (1 << 1);  // Set GPIO Port 0 Pin 0 to LOW
26    }
27  }
28
```

Listing 6: Modified program to run on LPC55S16 Cortex-M33

For the test exponents with a lower number of bits have been used. The principle of the attack itself is still valid, but with exponents that have a very high number of bits, the pattern would have been very hard to visualize in this report.

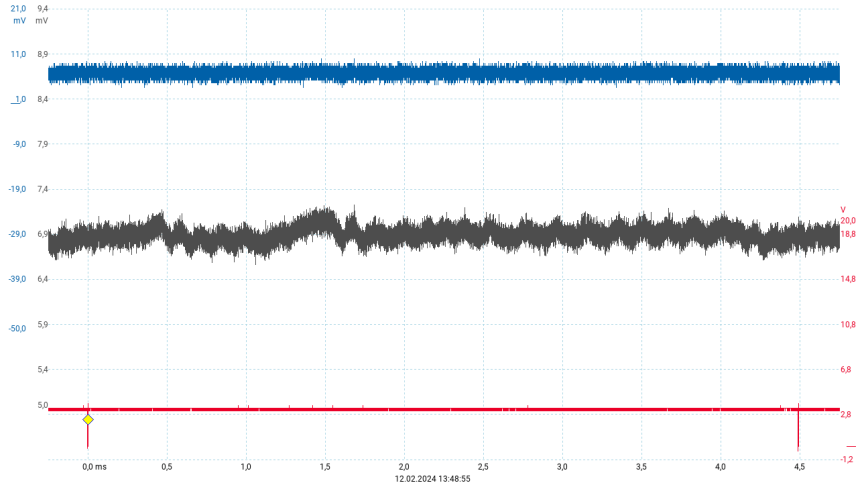

Figure 5: Voltage curve across the shunt resistor with an exponent of 10101 and no delay

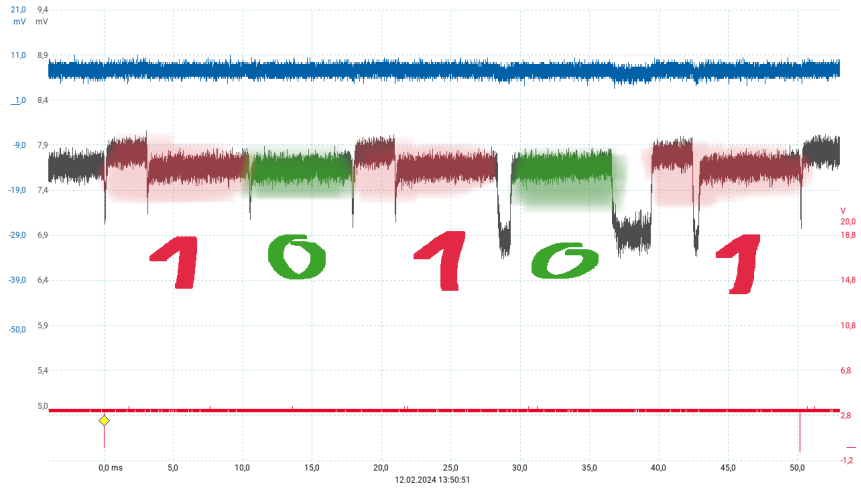

Figure 6: Voltage curve with an exponent of 10101 and added delay

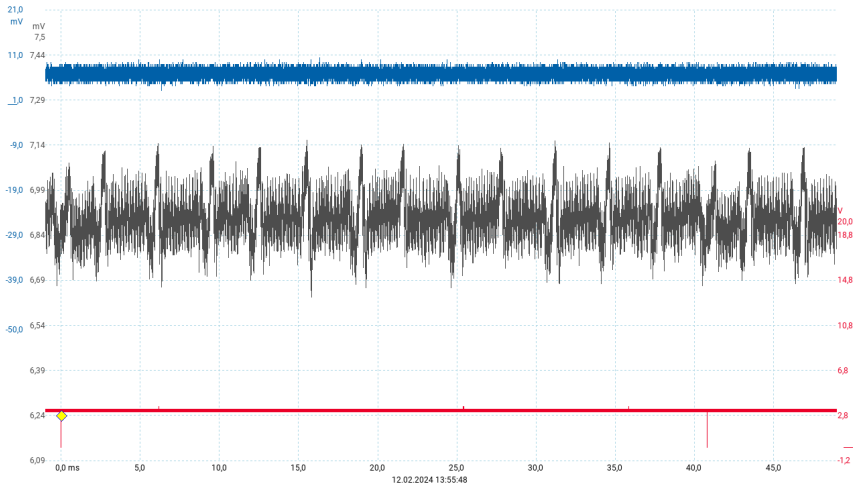

Figure 7: Voltage curve with an exponent of 0xDEAF and no delay

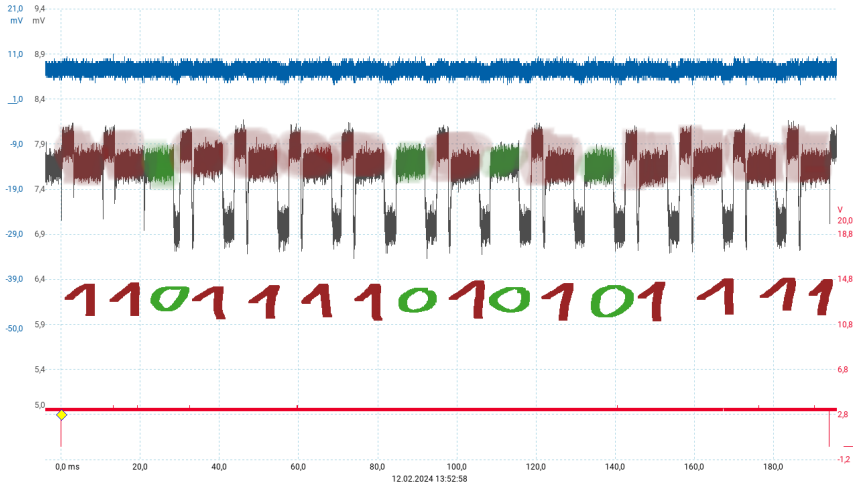

Figure 8: Voltage curve with an exponent of 0xDEAF and added delay

Figures 5-8 depict the voltage curves across the shunt resistors. In figures 5 and 6 an exponent of 10101 was used. While it is quite difficult to a pattern when no extra delay is added after each arithmetic operation, the pattern becomes quite obvious with an added delay.
Figures 7 and 8 show the process with an exponent of 0xDEAF = 1101 1110 1010 1111.

5 Conclusion

The robustness and validity of the implemented RSA algorithm has been verified by successful execution of the NIST test vectors. It has also been shown that the addition of the Chinese Remainder Theorem in the decryption process significantly improves the performance. As a last point security aspects have been discussed and counter measurements to specific side-channel attacks have been implemented.

References

[1] National Institute of Standards and Technology (NIST). Cryptographic algorithm validation program, 2024. https://csrc.nist.gov/projects/cryptographic-algorithm-validation-program.

[2] National Institute of Standards and Technology (NIST). RSADP Decryption Operation Primitive Component Test Vectors. Available at https://csrc.nist.gov/CSRC/media/Projects/Cryptographic-Algorithm-Validation-Program/documents/components/RSADPtestvectors.zip, 2024.

[3] H. Neumann. Kryptographie. Lecture slides, 2018. Accessed: February 8, 2024.

[4] H. Neumann. Rsa lecture slides, 2023. Accessed: February 8, 2024.

[5] Elena Schmidt. Absicherung von ST-Controllern gegen Produktpiraterie, 2021.

[6] The FLINT team. *FLINT: Fast Library for Number Theory*, 2023. Version 3.0.0, https://flintlib.org. Accessed: February 8, 2025.