M1 Algebraic Algorithms for Cryptology: Homework

Hatim Ridaoui

March 2025

This document presents the solutions for Algebraic Algorithms for Cryptology homework.

1 Exercice 1: Implementation of $\mathbb{Z}/n\mathbb{Z}$ and RSA

1.1 (a) Maximum value for n in `uint64_t`

The elements of $\mathbb{Z}/n\mathbb{Z}$ are represented using `uint64_t` in C.

Initially, we assume that the largest possible n is:

$$n_{\max} = 2^{64} - 1$$

However, modular arithmetic involve multiplication before applying modulo n, meaning intermediate values can be as large as:

$$(x \times y) \mod n$$

For a `uint64_t`, the largest possible product befor modulo is :

$$(2^{64} - 1) \times (2^{64} - 1) \approx 2^{128}$$

which exceeds 64-bit storage and causes overflow. Therefore, for safe computation without multi-precision arithmetic, the maximum practical choice for n should be:

$$n_{\max} \approx 2^{32}$$

This ensures that intermediate values remain within a 64-bit representation.

Note. Although we argue that $n_{\max} \approx 2^{32}$ is safe for 64-bit multiplication, in practice we should be cautious. Any additional shift or partial operation in our implementation can push intermediate values beyond 64 bits, causing overflow. For genuinely large moduli or for a margin of safety, multi-precision arithmetic (see part *(h)*) is preferred.

1.2 (b) Implementation of arithmetic operations

Check code files

1.3 (c) Modular exponentiation function

Modular exponentiation computes:

$$\text{modexp}(x, k, n) = x^k \mod n$$

using an efficient method to avoid large intermediate values. After a quick research, we found that the standard approach is the **exponentiation by squaring** algorithm, which runs in $O(\log k)$ time.

Check code files

We ensure that calculations remain within bounds by taking modular reductions at each step. The bitwise right shift

$$k >> 1$$

divides the exponent, reducing complexity.

1.4 (d) RSA textbook definition

Textbook RSA consists of the following :

- **Key Generation**:
  - The public key is $K_{\text{pub}} = (n, e)$ where $n = p \cdot q$, with p and q large prime numbers.
  - The private key is $K_{\text{priv}} = d$, where d is the inverse of e modulo $\phi(n)$, computed as:

$$d = e^{-1} \mod \phi(n)$$

    where $\phi(n) = (p-1)(q-1)$ is Euler's totient function (Indicatrice d'Euler, vue en CPGE).

- **Encryption**: For a message m, the ciphertext is computed as:

$$C = m^e \mod n$$

- **Decryption**: For a ciphertext C, the original message is recovered using:

$$m = C^d \mod n$$

The security of RSA relies on the difficulty of integer factorization: given n, it is hard to retrieve p and q, and reconstruct $\phi(n)$ to compute d.

1.5 (e) Demo of $\mathbb{Z}/n\mathbb{Z}$ Functions

Check code files

1.6 (f) RSA Keygen, enc/dec functions

In a separate file `rsa.c`, we implemented:

- Key Generation (`rsa_keygen`): takes two primes p and q to compute

$$n = p \cdot q, \quad \phi(n) = (p-1)(q-1), \quad e, \quad d = e^{-1} \bmod \phi(n).$$

- Encryption (`rsa_encrypt`): given a message m, computes $m^e \bmod n$.
- Decryption (`rsa_decrypt`): given a ciphertext c, computes $c^d \bmod n$.

Check code files

1.7 (g) RSA Demo with the Provided Primes

We wrote two separate test files:

- `test-rsa.c`: uses one example pair of primes to illustrate RSA encryption and decryption.
- `test-rsa-extended.c`: uses all the primes listed in the assignment, verifying correctness for each pair.

To handle intermediate products safely, we used **128-bit multiplication** where necessary, avoiding overflow when computing $(x \times y) \bmod n$.

1.8 (h) Implementation of Multi-Precision Integers

When our modulus n gets really large, using `uint64_t` will inevitably cause overflow in multiplications. To avoid this, we built a simple multi-precision system that stores numbers in arrays of 32-bit digits. We then implemented the core operations on these arrays.

Now, intermediate products can grow beyond 64 bits without overflow, making it possible to handle very large RSA moduli (such as 1024-bit or even larger).

1.9 (i) Redoing the exercises

See code files

2 Exercice 2: Implementation of Binary Galois Fields $\mathbf{F}_{128}$

2.1 (a) Recall how the field $\mathbf{F}_{128}$ is defined

1. Since $\mathbf{F}_{128}$ has 2^7 elements, each element can be seen as a polynomial
$$a(x) = a_0 + a_1x + a_2x^2 + \cdots + a_6x^6$$
where each $a_i \in \{0, 1\}$ and the polynomial is taken modulo some irreducible polynomial of degree 7 over $\mathbf{F}_2$. Practically, we store these polynomials in a 7-bit (or 8-bit) representation.

2. In characteristic 2, addition corresponds to bitwise XOR of coefficients. For two polynomials $a(x)$ and $b(x)$,
$$a(x) + b(x) = (a_0 \oplus b_0) + (a_1 \oplus b_1)x + \cdots + (a_6 \oplus b_6)x^6.$$
Since $-1 = 1$ in $\mathbf{F}_2$, subtraction is the same as addition.

3. To multiply two polynomials $a(x)$ and $b(x)$, we do the usual polynomial multiplication over $\mathbf{F}_2$, then reduc the result modulo the chosen irreducible polynomial of degree 7. Concretely,
$$c(x) = a(x) \cdot b(x) \bmod p(x),$$
where $p(x)$ is our irreducible polynomial of degree 7, ensuring the result stays within degree < 7.

4. Every non-zero element $a(x)$ in $\mathbf{F}_{128}$ has a multiplicative inverse $a(x)^{-1}$ such that
$$a(x) \cdot a(x)^{-1} \equiv 1 \bmod p(x).$$
We find this inverse via the Extended Euclidean Algorithm for polynmials over $\mathbf{F}_2$, solving
$$a(x)\, x'(x) + p(x)\, y(x) = 1.$$
The polynomial $x'(x)$ in this equation is the inverse of $a(x)$ modulo $p(x)$.

2.2 (b) Which Polynomials Construct $\mathbf{F}_{128}$?

A polynomia $p(x)$ over $\mathbf{F}_2$ of degree $n = 7$ is irreducible if it cannot be factored into non-trivial polynomials in $\mathbf{F}_2[x]$ of lower degree. Concretely, one standard test is:

$$\gcd\big(p(x),\, x^{2^n} - x\big) = 1, \quad \text{and for each divisor } d \text{ of } n,\ \gcd\big(p(x),\, x^{2^d} - x\big) = 1.$$

If all these conditions hold, $p(x)$ is irreducible.

Polynomials Given:

$$\begin{aligned}
P_1 &= X^{128} + X + 1 && \text{(degree 128)},\\
P_2 &= X^8 + X + 1 && \text{(degree 8)},\\
P_3 &= X^7 + X + 1 && \text{(degree 7)},\\
P_4 &= X^7 + X^4 + X + 1 && \text{(degree 7)},\\
P_5 &= X^8 + X^4 + X^3 + X + 1 && \text{(degree 8)}.
\end{aligned}$$

Since we need a degree-7 polynomial for $\mathbf{F}_{2^7}$, the only candidates are P_3 and P_4. Both can be confirmed irreducible by checking they do not factor in $\mathbf{F}_2[x]$ and by verifying that each divides $x^{128} - x$ but not $x^{2^d} - x$ for any proper divisor d of 7. In practical cases, we also rely on standard tables of known irreducible polynomials. This confirms that:

$$P_3 = X^7 + X + 1 \quad \text{and} \quad P_4 = X^7 + X^4 + X + 1$$

are indeed irreducible over $\mathbf{F}_2$. Hence, either P_3 or P_4 can be used to construct $\mathbf{F}_{128}$, where elements are polynomials of degree < 7 taken modulo one of these irreducible polynomials.

Other Polynomials: - P_1 is degree 128, which would define $\mathbf{F}_{2^{128}}$ instead of $\mathbf{F}_{128}$. - P_2 and P_5 are degree 8, corresponding to $\mathbf{F}_{2^8}$ (256 elements), not $\mathbf{F}_{128}$.

2.3 (c) Field Operations in `f128.c`

See code files

2.4 (d) Demonstration File: `f128-demo.c`

See code files

2.5 (e) Zech-Log representation

A Zech-log representation of $\mathbf{F}_{128}$ speeds up multiplication and inversion by storing each nonzero field element a in terms of a logarithm to a fixed generator α, so that

$$a = \alpha^{\log(a)}.$$

Once we have a table for log (mapping elements to exponents) and an antilog table for exp (mapping exponents back to elements), multiplication reduces to:

$$a \cdot b = \alpha^{\log(a)} \times \alpha^{\log(b)} = \alpha^{(\log(a)+\log(b)) \bmod (2^7-1)},$$

and inverses become negation of exponents:

$$a^{-1} = \alpha^{-(\log(a))} = \alpha^{(2^7-1)-\log(a)}.$$

A "Zech-log" specifically encodes the idea of handling $\alpha^k + 1$ through short table lookups, further optimizing additions like $x + 1$. This approach significantly accelerates field arithmetic when many multiplications or inversions are needed.

2.6 (f) Additional Condition for the Polynomial

Besides being irreducible, the chosen polynomial must also be primitive. A polynomial $p(x)$ of degree 7 is primitive if one of its roots α is a generator of the entire multiplicative group $\mathbf{F}_{128}^*$. This means:

$$\alpha^{127} = 1, \quad \text{and for any proper divisor } d \text{ of } 127,\ \alpha^d \neq 1.$$

In other words, α has order 127, the full size of the nonzero elements in $\mathbf{F}_{128}$. This ensures that the "logarithm" base α cycles through all non-zero elements exactly once.

Why all polynomials used here are primitive We rely on standard references listing which degree-7 irreducible polynomials are also primitive. Each polynomial suggested for $\mathbf{F}_{128}$ in this assignment is known to have a root of order 127, thus meeting the primitivity condition. Hence, when we build the field with such a polynomial, its root is a valid generator for the multiplicative group, guaranteeing that the Zech-log method covers all non-zero elements and enabling efficient multiplication and inversion.

2.7 (g) Alternate implementation with zech Logs in `f128-zech.c`

See code files

2.8 (h) Second Demo in `f128-zech-demo.c`

See code files

Execution and Tests For full instructions on compiling and running each part, please see the `README.md`.