Aufgabenblatt Kryptografie mit öffentl. Schlüsseln

Lösen Sie die nachfolgenden Aufgaben und bereiten Sie diese bis zum nächsten Lehrveranstaltungstermin vor.

LB-KÖS 00. (nicht abzugeben)

(*Code::Blocks*-Template *GMP project*)

Schreiben Sie ein Programm unter Zuhilfenahme der *GMP*, das zwei als Argumente übergebene Zahlen multipliziert und das Ergebnis ausgibt. Überprüfen Sie Ihr Programm mit den Beispielzahlenpaaren `2 3` und `x x`, wobei `x` die größtmögliche Zahl ist, die in einem vorzeichenbehafteten 64-Bit-Integer gespeichert werden kann.

LB-KÖS 01. (*Code::Blocks*-Template *GMP project*)

Die Verschlüsselung c einer Nachricht m mit dem öffentlichen Schlüssel (e, N) ist in RSA wie folgt definiert: $c \equiv m^e \pmod{N}$. Die Entschlüsselung mit dem privaten Schlüssel (d, N) ist analog wie folgt definiert: $m \equiv c^d \pmod{N}$.
Schreiben Sie ein Programm unter Zuhilfenahme der *GMP*, das eine Ver- oder Entschlüsselung mit RSA implementiert. Das Programm soll drei Parameter in der folgenden Reihenfolge entgegennehmen: die zu ver- bzw. entschlüsselnde Nachricht m **als Zahl**, einen Exponenten (e bzw. d) und den Modulus (N). Der ver- bzw. entschlüsselte Wert soll (**ohne** weitere Ausgaben) als Zahl auf `std::cout` ausgegeben werden. Verwenden Sie für Ihre Implementierung die Funktion `mpz_powm` und testen Sie diese mit $m = 7, e = 29, d = 85$ und $N = 391$.
Hinweis: Verwenden Sie die Methode `mpz_class::get_mpz_t`, um die Instanzen der `mpz_class`-Klasse an die Funktion `mpz_powm` zu übergeben. Details zu den GMP*-Funktionen entnehmen Sie der Dokumentation unter `https://gmplib.org/manual/Function-Index.html#Function-Index`. `mpz_class`-Instanzen können während des Debuggens ausgewertet werden, indem im* Watch*-Fenster `variable_name.get_str(10)` eingegeben wird, wobei `variable_name` durch den Variablennamen der Instanz ersetzt werden muss.*

LB-KÖS 02. (*Code::Blocks*-Template *GMP project*)

Ein RSA-Schlüsselpaar, bestehend aus dem öffentlichen Schlüssel (e, N) und dem privaten Schlüssel (d, N) kann wie folgt erzeugt werden:

1. Wählen Sie zwei voneinander verschiedene Primzahlen p und q, d.h. $p, q \in \mathbb{P}$, wobei $p \neq q$.
2. Berechnen Sie $N = pq$.
3. Berechnen Sie $\varphi(N) = (p-1)(q-1)$.

4. Wählen Sie für den öffentlichen Schlüssel eine ganze Zahl e zwischen 1 und $\varphi(N)$ (jeweils ohne die beiden Grenzen), die zu $\varphi(N)$ teilerfremd ist, d.h. $\mathrm{ggT}(e, \varphi(N)) = 1$.

5. Berechnen Sie für den privaten Schlüssel d als die Inverse von e modulo $\varphi(N)$, d.h. $d \equiv e^{-1} \pmod{\varphi(N)}$.

Schreiben Sie ein Programm, das **keine** Parameter entgegennimmt, ein RSA-Schlüsselpaar wie oben angegeben erzeugt und den öffentlichen sowie den privaten Schlüssel nach **exakt** folgendem Muster (Groß-/Kleinschreibung, Leerzeichen, etc.) auf `std::cout` ausgibt:

```
Public key: (29, 391)
Private key: (85, 391)
```

Verifizieren Sie die Schlüssel, indem Sie diese verwenden, um eine Nachricht mit Ihrem Programm aus Beispiel 01. zu ver- und wieder zu entschlüsseln. Die Schlüsselgröße (Bitlänge von N, die durch die Längen von p und q beeinflusst wird) soll exakt 2.048 Bit betragen. Falls die Schlüssellänge nicht 2.048 Bit beträgt, muss ein neuer Schlüssel erzeugt werden. Erzeugen Sie gegebenenfalls so lange neue Schlüssel, bis Länge exakt erreicht ist.
Hinweis: Verwenden Sie die beiden Funktionen `gmp_randinit_default` und `gmp_randseed_ui`, um bei Programmstart einen Zufallszahlengenerator zu initialisieren. Erzeugen Sie mit diesem und der Funktion `mpz_urandomb` eine Zufallszahl und wenden Sie `mpz_nextprime` an, um eine Primzahl zu erhalten. Für die Bestimmung der Länge verwenden Sie die Funktion `mpz_sizeinbase`.
Für die Wahl von e iterieren Sie durch den angegebenen Wertebereich, bis Sie einen Wert gefunden haben, der das spezifizierte Kriterium erfüllt. Für die Berechnung des größten gemeinsamen Teilers verwenden Sie die Funktion `mpz_gcd`. Zum Berechnen der Inversen im Modulus verwenden Sie `mpz_invert`.
Nutzen Sie für arithmetische und Vergleichsoperationen wo immer möglich die entsprechenden überladenen Operatoren, z.B. `+`, `<`, `==` etc.

LB-PKC 03. (*Code::Blocks*-Template *GMP project*)

Modifizieren Sie Ihr Programm aus Beispiel 01. so, dass es Zeichenfolgen anstatt numerischer Werte ver- und entschlüsselt. Um Probleme mit nichtdruckbaren Zeichen zu vermeiden, dürfen die Nachricht (beim Verschlüsseln) bzw. der Chiffretext (beim Entschlüsseln) nicht als Kommandozeilenparameter übergeben werden, sondern müssen stattdessen aus einer Datei gelesen werden, deren Pfad als Kommandozeilenparameter übergeben wird. Analog dürfen der Chiffretext (beim Verschlüsseln) bzw. die Nachricht (beim Entschlüsseln) nicht auf der Konsole ausgegeben werden, sondern müssen stattdessen in eine Textdatei geschrieben werden, deren Pfad als Kommandozeilenargument übergeben wird. Beispielaufruf: `input.txt output.txt $e $N`.
Für die Umwandlung zwischen Zeichenfolgen und numerischen Werten gibt es mehrere Möglichkeiten. Eine der einfachsten besteht darin, jedes Zeichen als

Zahl zwischen 0 und 255 zu behandeln (ASCII-Zeichen nutzen nur die untere Hälfte dieses Bereichs aus, wenn jedem Zeichen seine Position in der ASCII-Tabelle zugewiesen wird, Chiffretexte verwenden den vollen Bereich). Zum Beispiel wird das Zeichen `A` in den Wert 65 umgewandelt und umgekehrt. Die Werte mehrerer umgewandelter Zeichen können als eine große Zahl in der Basis 256 interpretiert werden, z.B. wird die Zeichenfolge `ABC` in $65 \cdot 256^2 + 66 \cdot 256^1 + 67 \cdot 256^0 = 4.276.803$ konvertiert und umgekehrt.
Hinweis: Verwenden Sie die Klassen `std::ifstream` und `std::ofstream` (beide benötigen den `fstream`-Header), um aus Dateien zu lesen und in sie zu schreiben. Dateien mit Chiffretextfolgen müssen dabei immer als Binärdateien behandelt werden, indem der entsprechende Stream-Dateimodus verwendet wird. Der Einfachheit halber können alle Dateien als Binärdateien behandelt werden.