IMS
Institut für Mikroelektronische Systeme
Leibniz Universität Hannover

Programmierpraktikum Technische Informatik (C++)
Programmiertutorium Vorlesung 3

Pullen Sie von ppti-common. Sie finden dann in ihrem Repository den Ordner `tutorials/Lecture03/`, in dem die folgenden Aufgaben zu bearbeiten sind.

Teilaufgabe 1

Schreiben Sie die Klasse `RNG` zum Erzeugen von Pseudozufallszahlen. Die Klasse enthält eine private vorzeichenlose Variable `x`, die im Konstruktor definiert wird. Außerdem soll sie die öffentliche Funktion `generateNumber()` bekommen, welche auf Basis von x eine neue Zufallszahl erzeugt. Der dafür verwendete Algorithmus soll den neuen Wert mit Hilfe der Formel $x = x \operatorname{xor} b(x)$ berechnen. Dabei ist $b(x)$ das Ergebnis eines Bitshifts von x vorgegebenen Werten. Zur Berechnung eines neuen x wird die Formel dreimal hintereinander angewandt, wobei die folgenden Werte für b verwendet werden sollen:

- 13 nach rechts
- 7 nach links
- 17 nach rechts

Am Ende wird `x` zurückgegeben.

Teilaufgabe 2

Für bestimmte Anwendungsgebiete werden große Primzahlen benötigt. Zur Erzeugung dieser Primzahlen werden häufig randomisierte Tests, die eine eingegebene Zahl entweder als Nichtprimzahl identifizieren, mit einer gewissen Wahrscheinlichkeit aber auch Zahlen fälschlicherweise als prim erkennen. Durch eine wiederholte Anwendung lässt sich diese Wahrscheinlichkeit beliebig reduzieren. Einer dieser Tests, der sogenannte Fermat-Test, soll im Folgenden programmiert werden!

a) Schreiben Sie die Funktion `modPow(a,b,n)`. Diese soll das Ergebnis x von $a^b \mod n$ für große Exponenten berechnen und zurückgeben. Am Anfang wird $x \leftarrow 1$ gesetzt. Dann wird in jedem Schritt zunächst geprüft, ob $lsb(b) = 1$ gilt. In diesem Fall wird $x \leftarrow x \cdot a \mod n$ gesetzt, ansonsten bleibt x in diesem Schritt unangetastet.

Appelstraße 4 | 30167 Hannover
fon 0511 762-19691 | fax 0511 762-19694
www.ims.uni-hannover.de | info@ims.uni-hannover.de

Weiterhin wird in jedem Schritt $a \leftarrow a^2 \mod n$ gesetzt und b halbiert. Dies wird wiederholt, solange $b \neq 0$ ist.

b) Schreiben Sie die Funktion `fermatTest(p, a)`. Diese Funktion soll einen Fermatschen Primzahltest für p mit Zeugen a durchführen. Der Primzahltest nach Fermat ist die Anwendung des kleinen Satz von Fermat, der besagt, dass, falls für ein beliebiges $a \neq p$ die Formel $a^{p-1} \mod p \neq 1$ erfüllt ist, p keine Primzahl sein kann. Entsprechend soll die Funktion `true` zurückgeben, falls $a^{p-1} \mod p = 1$ gilt und p somit eine Primzahl sein könnte, und ansonsten `false`.

c) Schreiben Sie die Funktion `isPrime(p, iterations)`. Diese soll die bereits geschriebene Funktion `fermatTest` verwenden um zu überprüfen, ob p eine Primzahl ist. Da der Fermattest die Primalität von p nicht garantieren kann, sondern lediglich mit einer gewissen Wahrscheinlichkeit bestimmt, dass p nicht prim ist, muss dieser Test wiederholt angewendet werden um eine gewisse Sicherheit zu haben, dass die Zahl tatsächlich eine Primzahl ist. Entsprechend soll die Funktion `fermatTest` `iterations`-mal aufrufen, wobei für `a` jeweils eine neue Zufallszahl übergeben wird, die durch einen in `isPrime` erstellten `RNG` erzeugt werden soll. Wenn alle Fermat-Tests für p erfolgreich sind, soll `true` zurückgegeben werden, sonst `false`.

Hinweise

- $lsb(x)$ steht für **l**east **s**ignificant **b**it und gibt den Wert der letzten Bitstelle des Arguments zurück, also 1, falls x ungerade ist und 0, falls es gerade ist.

Teilaufgabe 3

RSA ist ein asymetrisches Verschlüsselungsverfahren. Die meisten Verschlüsselungsverfahren verwenden den selben Schlüssel um eine Nachricht zu ver- und entschlüsseln. Bei asymmetrischen Verfahren sind die Schlüssel zum Ver- und Entschlüsseln hingegen verschieden und nicht ohne weiteres auseinander ableitbar. Dies hat den entscheidenden Vorteil, dass der Schlüssel zum Verschlüsseln nicht geheim gehalten werden muss, sondern beliebig verteilt werden kann. Daher nennt man diese Verfahren auch Public-Key-Verfahren und den Schlüssel für die Verschlüsselung auch Public-Key oder öffentlichen Schlüssel. Der Schlüssel zum Entschlüssel ist analog der Private-Key oder private Schlüssel. Dieses Verfahren soll in dieser Aufgabe in einer einfachen Form implementiert werden. Dafür sollen die Lösungen der bisherigen Aufgabenteile zur Erzeugung der für RSA benötigten zufälligen Primzahlen verwendet werden.

a) Schreiben Sie die Funktion `advancedEuclid(x,y)`, die zur Berechnung des größten gemeinsamen Teilers (ggT) zweier ganzer Zahlen `x` und `y` dient. Dazu soll der erweiterten euklidischen Algorithmus verwendet werden. Dieser nutzt das Lemma von Bézot, das besagt, dass es ganze Zahlen s und t geben muss, für die gilt $ggT(x,y) = s \cdot x + t \cdot y$. Der Algorithmus berechnet diese Linearkombination, um den

ggT zu ermitteln. Zunächst wird s auf 1 und t auf 0 gesetzt. Zudem werden die beiden Hilfvariablen u und v benötigt, wobei u mit 0 und v mit 1 initialisiert wird. Dann werden die folgenden Schritte wiederholt, solange $y \neq 0$ gilt:

- Berechne q als Ergebnis der Ganzzahldivision von x und y
- Setze $s \leftarrow u$, $t \leftarrow v$, $u \leftarrow s - q \cdot u$, $v \leftarrow t - q \cdot v$
- Setze $x \leftarrow y$, $y \leftarrow x \mod y$

Als Ergebnis sollen x,s und t als Tuple zurückgegeben werden, wobei x am Ende des Algorithmus der gesuchte ggT ist und s und t die Faktoren für die Linearkombination darstellen.
Hinweis: Die Funktion `std::tie(a,b,...)` erzeugt ein Tupel von Referenzen auf seine Argumente und kann verwendet werden, um mehreren Variablen in einem Schritt gleichzeitig etwas zuzuweisen.

b) Schreiben Sie die Funktion `findPrime`. Diese erhält eine Referenz auf einen Zufallszahlengenerator `RNG` und einen maximalen Suchwert `n` als Argumente und soll eine zufällig erzeugte Primzahl kleiner `n` zurückgeben. Dazu sollen solange Zufallszahlen kleiner `n` erzeugt und mit `isPrime` auf Primalität getestet werden, bis eine Primzahl gefunden wurde. Diese Primzahl soll dann zurückgegeben werden.

c) Schreiben Sie nun die Klasse RSA mit dem folgenden Aufbau:

- Eine private Variable `decryptexp`, folgend d
- Eine öffentliche Variable `encryptexp`, folgend e
- Eine öffentliche Variable `rsamodul`, folgend N
- Einen öffentlichen Konstruktor
- Eine öffentliche Funktion `encrypt`
- Eine öffentliche Funktion `decrypt`

Dem Konstruktor wird eine Referenz auf einen Zufallszahlengenerator `RNG` übergeben. N wird im Konstruktor als Produkt zweier unterschiedlicher, zufällig erzeugter Primzahlen p_1 und p_2 berechnet. Dabei sollen p_1 und p_2 jeweils kleiner als 256 und $N = p_1 \cdot p_2$ größer als 255 sein. Entsprechend muss die Erzeugung in einer Schleife durchgeführt werden, bis die gestellten Bedingungen erfüllt sind.

Weiterhin werden die Exponenten d und e im Konstruktor gesetzt. Diese können anhand der vorgegebenen Funktion `calcExponents` berechnet werden, die den Zufallszahlengenerator und $\phi(N)$ übergeben bekommt, wobei sich $\phi(N)$ als $(p_1 - 1) \cdot (p_2 - 1)$ berechnet. Die Funktion `calcExponents` gibt dabei ein Tupel zurück, dessen erste Komponente e und dessen zweite Komponente d ist.

Die Funktion `encrypt` bekommt eine vorzeichenlose Ganzzahl `m` als Nachricht übergeben und gibt eine verschüsselte Ganzzahl `c` zurück, die nach der Formel $c = m^e \mod N$ berechnet wird. `decrypt` soll eine übergebene verschlüsselte Zahl wieder entschlüsseln, wofür die Formel $m = c^d \mod N$ verwendet wird.