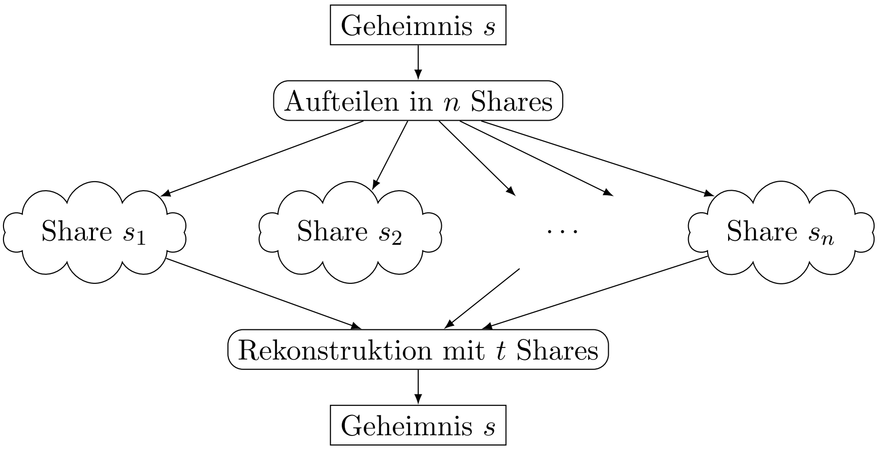
AG IT-Sicherheit
Secret-Sharing

1 Einführung und Orientierung



Unter (t, n)-Secret-Sharing versteht man ein Verfahren, ein Geheimnis (z.B. einen kryptografischen Schlüssel, ein Passwort, eine Safe-Kombination oder auch eine ganze Datei) so in n Teile (*Shares* genannt) aufzuteilen, dass man mindestens t Teile kombinieren muss, um das ursprüngliche Geheimnis wiederherstellen zu können. Falls nur $t - 1$ oder weniger Shares vorhanden sind, soll es informationstheoretisch unmöglich sein, etwas über das Geheimnis zu erfahren: alle möglichen Geheimnisse sind dann immer noch gleich wahrscheinlich.

Es ist sofort einleuchtend, dass Secret-Sharing nur für $n \geq t \geq 2$ sinnvoll ist. Im einfachsten Fall haben wir $t = n$ (dann werden alle Shares zur Rekonstruktion benötigt), aber $t < n$ ist immer dann sinnvoll, wenn die Wiederherstellung trotz des Ausfalls einiger Shares immer noch funktionieren soll.

Eine mögliche Anwendung eines $(2, 4)$-Secret-Sharing besteht zum Beispiel darin, die Shares einer Datei mit wichtigem Inhalt auf 4 Cloud-Provider verteilt abzuspeichern (etwa Google, Amazon, Microsoft und Dropbox). Dann müssten für einen erfolgreichen Angriff zwei Benutzerkonten kompromittiert werden oder zwei Cloud-Anbieter miteinander kooperieren, während gleichzeitig der Ausfall zweier Anbieter für die Wiederherstellung kein Problem darstellt.

1.1 Einfaches (n, n)-Secret-Sharing

Für diesen Spezialfall existiert ein einfaches Verfahren, das auf binärem XOR basiert. Zur Erinnerung: die XOR-Verknüpfung zweier Bits ist definiert als

$$\begin{array}{c|cc} \oplus & 0 & 1 \\ \hline 0 & 0 & 1 \\ 1 & 1 & 0 \end{array},$$

und kann trivial auf ganze Folgen von Bytes verallgemeinert werden. In Java entspricht dies dem Operator `^`, der ganz einfach für alle Elemente eines `byte`-Arrays iteriert werden kann.

Wir werden weiterhin eine Zufallszahlenquelle benötigen. In Java gibt es dafür die Klasse `java.security.SecureRandom`, deren Methode `nextBytes` ein Byte-Array mit neuen Zufallszahlen füllt.

Ein Geheimnis s der Länge l Bytes kann dann wie folgt auf n Shares $s_1, \ldots, s_n$ verteilt werden:

1. Für $i = 1, \ldots, n-1$: berechne eine neue zufällige Bytesequenz r_i der Länge l und setze $s_i := r_i$.
2. Setze $s_n := s \oplus \bigoplus_{i=1}^{n-1} s_i = s \oplus s_1 \oplus \cdots \oplus s_{n-1}$.

Alle n Shares sind jeweils ebenfalls l Bytes lang. Die Rekonstruktion des Geheimnisses s aus den n Shares $s_1, \ldots, s_n$ erfolgt analog:

1. Setze $s := \bigoplus_{i=1}^{n} s_i = s_1 \oplus \cdots \oplus s_n$.

Wegen $s_i \oplus s_i = 0$ sieht man die Korrektheit sofort ein (XOR ist sein eigenes Inverses). Sind weniger als n Shares vorhanden, kann das fehlende Share jedes Bit mit Wahrscheinlichkeit $1/2$ kippen – man ist also genauso schlau wie ohne Kenntnis der $n-1$ Shares.

1.2 Shamirs (t, n)-Secret-Sharing

Soll jede Teilmenge von t Shares, $2 \leq t \leq n$, die Rekonstruktion des Geheimnisses ermöglichen, benötigt man ein anderes Verfahren, das 1979 vom Turing-Preisträger Adi Shamir entwickelt wurde. Während das XOR-Verfahren mit Eingaben beliebiger Länge arbeiten kann, erfordert Shamirs Verfahren Geheimnisse s, die sich als Zahl in der Form $0 \leq s < p$ für eine große Primzahl p darstellen lassen. (In der Praxis ist das keine Einschränkung, siehe Abschnitt 1.)

Ein Geheimnis s dieser Form kann dann wie folgt auf n Shares $s_1, \ldots, s_n$ verteilt werden, von denen jede Teilmenge von mindestens t Shares zur Rekonstruktion ausreicht:

1. Für $i = 1, \ldots, t-1$: berechne eine neue Zufallszahl a_i, $0 \leq a_i < p$.
2. Betrachte das Polynom $f(x) = s + a_1 x + a_2 x^2 + \cdots + a_{t-1} x^{t-1}$ und berechne für $i = 1, \ldots, n$ das Share $s_i := (x_i, y_i) = (i, f(i) \bmod p)$.

Das i-te Share s_i besteht also aus dem Punkt mit der x-Koordinate i und der y-Koordinate $f(i) \bmod p$, wobei das Polynom modulo p evaluiert wird.

Zur Rekonstruktion von s benötigen wir den Wert $f(0) \bmod p$, den wir durch Interpolation der vorliegenden Punkte errechnen können: t oder mehr Punkte bestimmen das Polynom f vom Grad $t-1$ eindeutig. Folglich können wir die wohlbekannte Lagrange-Interpolation verwenden, um aus einer beliebigen Teilmenge von wenigstens t Shares das Polynom und damit auch $f(0) \bmod p$ wiederherzustellen. Angenommen, wir verfügen über die k Shares $s_1 = (x_1, y_1), \ldots, s_k = (x_k, y_k)$, wobei $t \leq k \leq n$. Dann können wir das Geheimnis s rekonstruieren als

$$s := f(0) \bmod p = \sum_{i=1}^{k} v_i \bmod p$$

mit

$$v_i = y_i \cdot \prod_{\substack{1 \leq j \leq k, \\ j \neq i}} \frac{-x_j}{x_i - x_j} \bmod p.$$

Bei der Implementierung ist hier zu beachten, dass alle Berechnungen modulo p ausgeführt werden müssen. Hierbei (und bei den großen Zahlen) hilft die Klasse `java.math.BigInteger` enorm, welche wie folgt verwendet werden kann:

```
// Gegeben BigIntegers a, b, c und p,
// berechne d = a / (b-c) = a * 1/(b-c)  modulo p
BigInteger d = a.multiply(b.subtract(c).modInverse(p)).mod(p);
```

Division wird also durch Multiplikation mit dem Inversen erreicht. Am Ende jeder Berechnung muss modulo p reduziert werden.

2 Aufgaben (1. Tag)

Der Aufgabenstellung beiliegend finden Sie ein Programmskelett, welches Sie verwenden können, aber natürlich nicht müssen. Lediglich die geforderte Funktionalität sollte übereinstimmen.

Aufgabe 1 (Einfaches (n, n)-Secret-Sharing mit XOR, 2+2=4 Punkte)
Wir betrachten zunächst das Verfahren aus Abschnitt 1.1.

a) Implementieren Sie dieses Verfahren (Sharing und Wiederherstellung) für allgemeines n und für Geheimnisse, die in Form von `byte[]`-Arrays beliebiger Länge vorliegen. Testen Sie die Korrektheit Ihrer Implementierung an einigen Beispielen. Die Datei `XorSecretSharing.java` enthält einen Vorschlag für die Schnittstellen für das Sharing und die Wiederherstellung.

b) Schreiben Sie ein einfaches Programm, welches die folgenden Funktionalitäten bietet:
   a) Aufteilung einer Datei `F` mit dem in a) implementierten Secret-Sharing-Verfahren in n Share-Dateien `F-1`, `F-2`, ..., `F-`n.
   b) Rekonstruktion der Datei `F` aus den n Einzeldateien.

Aufgabe 2 (Shamirs (t, n)-Secret-Sharing, 4+2=6 Punkte)
Wir implementieren jetzt Shamirs Verfahren aus Abschnitt 1.2. Aus Gründen der Einfachheit lassen wir die Umwandlung von binären Daten oder Dateien in Ganzzahlen und zurück für heute zunächst weg.

a) Implementieren Sie dieses Verfahren (Sharing und Wiederherstellung) für allgemeines t und n und für Geheimnisse, die bereits in Form von `BigInteger`s vorliegen. Als Primzahl nehmen wir $p = 2^{256} + 297$, so dass wir im Prinzip Geheimnisse bis zu $256/8 = 32$ Byte Länge als `BigInteger` kodieren und verarbeiten könnten. Für die Generierung der zufälligen Koeffizienten a_i ist folgender `BigInteger`-Konstruktor hilfreich:

   ```
   BigInteger ai = new BigInteger(p.bitLength(), rng);
   ```

   wobei `rng` ein `SecureRandom`-Objekt ist. Für die Evaluierung des Polynoms empfiehlt sich das Horner-Schema[1].

[1] `https://de.wikipedia.org/wiki/Horner-Schema`

Die Dateien `ShamirSecret.java` und `ShamirSecretSharing.java` enthalten einen Vorschlag für die Schnittstellen für das Sharing und die Wiederherstellung sowie ein paar andere Vorarbeiten (z.B. die Definition von p als `BigInteger`). Beachten Sie auch, dass die x-Koordinaten für die Shares $s_1, s_2, \ldots, s_n$ wirklich bei 1 und nicht bei 0 beginnen müssen, da sonst $s_1 = f(0) \bmod p = s$ wäre, was dem Zweck des Secret-Sharings widerspricht.

b) Demonstrieren Sie die Korrektheit Ihrer Implementierung an einigen Beispielen, in denen Sie `BigInteger`s sharen und rekonstruieren (oder noch besser: schreiben Sie einen vernünftigen JUnit-Test für Ihre Shamir-Secret-Sharing-Klasse).