

# Warum der Wert von $n$ in $2^n$ wichtig ist

Fabio Oesch & Jan Fässler

4. November 2013

Es wird beschrieben, wie man 2 Briefe mit dem selben Hashwert erstellt.  
Diese haben eine unterschiedliche Kontonummer.

---

## Inhaltsverzeichnis

<b>1</b>	<b>Aufgabenstellung</b>	<b>2</b>
<b>2</b>	<b>Software Aufbau</b>	<b>2</b>
2.1	HashingMachine . . . . .	2
2.1.1	preprocess . . . . .	2
2.1.2	create . . . . .	2
2.2	LabOne . . . . .	3
2.2.1	main . . . . .	3
2.2.2	resolveCombination . . . . .	3
2.2.3	createVariation . . . . .	3
2.2.4	generateHashes . . . . .	4

# 1 Aufgabenstellung

Konstruieren Sie zwei Briefe an Alice, einen (original) für Bob und einen (gefälscht) für Alice, die aber den gleichen Hashwert haben. Da die Fälschung etwas für Sie einbringen soll, ersetzen Sie im Brief an Alice die Kontonummer 222-1101.461.12 durch Ihre eigene: 202-1201.262.10. Sie haben freilich bei der gleichen Bank ein Konto mit dieser Nummer eröffnet.

## 2 Software Aufbau

Das Projekt wurde in 2 Klassen unterteilt. Die Klasse HashingMachine ist für das Hashing zuständig. Die Klasse LabOne ist für das einlesen der Daten und das Durchprobieren der Hashingmethoden zuständig.

### 2.1 HashingMachine

Diese Klasse hasht den Vorgegebenen Text mit der Hashfunktion die wir von der Aufgabe erhalten haben.

#### 2.1.1 preprocess

Die Methode preprocess(byte[] input) strukturiert den Input wie er nachher benötigt wird.

##### Listing 1: Structure the input

```
1 for (int i = 0; i < input.length; i++) out[i] = input[i];
2 if (r > 0) out[input.length] = -128;
3 for (int i = 1; i < r; i++) out[input.length + i] = 0;
4 for (int i = 0; i < 8; i++) out[out.length - 8 + i] = length
  [i];
```

Die erste Linie kopiert den Input in den Output. Linien 2 und 3 sind für das Padding verantwortlich. Der erste Teil des Paddings fügt 10...0 hinzu und der zweite füllt den Rest mit Nullen auf. Die Linie 4 kopiert die Länge in den Output.

#### 2.1.2 create

##### Listing 2: Hashing

```
1
2 byte[] hash = iv.clone();
3
```

```
4 for (int i = 0; i < input.length; i += 8) {
5     try {
6         desOut = new byte[16];
7         cipher.init(true, new KeyParameter(hash));
8         cipher.processBytes(input, i, 8, desOut, 0);
9         cipher.doFinal(desOut, 0);
10        for (int j = 0; j < hash.length; j++)
11            tempState[j] = (byte) ((desOut[j] ^ desOut[j + 8]) ^ hash[j]);
12    } catch (CryptoException ce) { System.err.println(ce); }
13
14    // swap
15    byte[] tmp = tempState;
16    tempState = hash;
17    hash = tmp;
18 }
19
20 ByteBuffer buffer = ByteBuffer.wrap(hash);
21 buffer.order(ByteOrder.LITTLE_ENDIAN);
22 long result = buffer.getLong();
23 return (int) (result >>> 32) ^ Integer.reverse((int) result);
```

---

Auf der Zeile 2 wird der cipher mit dem Initialvektor initialisiert.

Zwischen Zeile 4 - 18 wird der Blockcipher ausgeführt. Das heisst es wird der alte Output genommen und ein neuer Hashwert erzeugt. Da desOut, die Ausgabe von der Hashfunktion, doppelt so gross ist, müssen wir diesen noch verkleinern. Dies machen wir mit  $desOut[j] \oplus desOut[j + 8]$ .

Ab Zeile 20 - 23 wird noch  $H$ , unser Output, in die Hälften  $H_1$  und  $H_2$  geteilt. Wobei  $H_2$  in der umgekehrten Reihenfolge mit  $H_1$  XOR'd. Dies ist nun der Output  $h(m)$ .

## 2.2 LabOne

In der Klasse LabOne wird die ganze Arbeit gemacht. Es werden zwei Template Dateien eingelesen und eine Datei mit den Optionen welche die Möglichkeiten für die Lücken in den Template Dateien zur Verfügung stellt.

### 2.2.1 main

In der main Funktion werden alle Einstellungen konfiguriert und das Suchen gestartet.

### 2.2.2 resolveCombination

Diese Funktion bekommt einen Integer und ein Template String. Sie schaut den Integer Bitweise an und nimmt je nach dem ob das LSBit eine 1 oder eine 0 ist die eine oder andere Option und setzt sie zu einem neuen String zusammen. Dieser wird dann zurückgegeben.

### 2.2.3 createVariation

Diese Funktion bekommt auch einen Integer und einen Template String. Diese werden danach direkt an die resolveCombination Funktion übergeben. Der neue Text der von dieser zurückgegeben wird, wird dann mittels der HashingMachine in einen Hash verwandelt.

### 2.2.4 generateHashes

Diese Funktion erstellt entweder zufällig oder der Reihe nach einen Block von Combinationen von den Original- und Fake Texten und speichert ihre Hash Werte ab.

#### Listing 3: Hashing Knacken

```
1 for (int i = 0; i < count; i++) {
2     if (useRandom) {
3         do { originalCombination = rand.nextInt(); } while (hashesOriginal.
            containsValue(originalCombination));
4         do { fakeCombination = rand.nextInt(); } while (hashesFake.
            containsValue(fakeCombination));
5     } else {
6         originalCombination++;
7         fakeCombination++;
8     }
9     hash = createVariation(originalCombination, templateOriginal);
10    hashesOriginal.put(hash, originalCombination);
11    hash = createVariation(fakeCombination, templateFake);
12    hashesFake.put(hash, fakeCombination);
13 }
```

### 2.2.5 checkCollisions

Methode welche überprüft ob es in der Liste der Hash Werte von den Fake-Texte eine Kollision haben mit den Hash Werte aus den Original Texten. Falls einer gefunden wird, wird ein Zähler hochgezählt und die gefundene Kollision abgelegt.

### 2.2.6 createAllVariation

Diese Funktion erstellt immer wieder einen neuen Block von Hash Werte und überprüft danach ob es in den erzeugten Hashs Kollisionen gibt.