Aufgabe 2.1

Vorgehensweise Um die Aufgaben zu bearbeiten, betrachten wir zunächst die Definition aus der Vorlesung, um zu prüfen, ob die Anforderungen an eine Hashfunktion erfüllt sind:

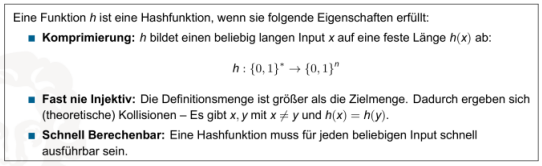

Abbildung 1: Definition Hashfunktion

Im Anschluss betrachten wir die erweiterte Definition aus der Vorlesung für kryptografische Hashfunktionen, die aus folgenden Eigenschaften besteht:

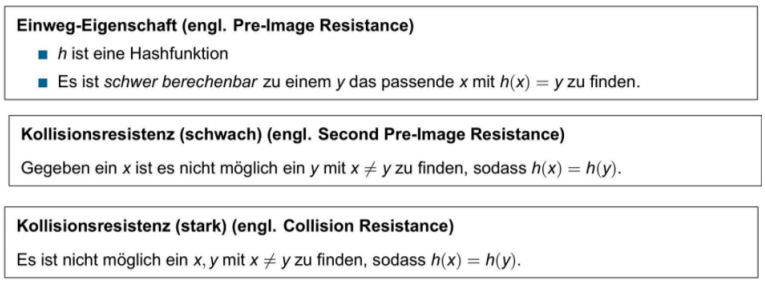

Abbildung 2: Deinition kryptografische Hashfunktionen

a) $h(x) = x \mod 7$
Komprimierung:
Ist erfüllt, da wir durch das Modulo 7, egal wie hoch die Eingabe ist, eine Zahl zwischen 0 und 6 erhalten werden.
Fast nie injektiv:
Ist erfüllt, die mögliche Definitionsmenge ($=\infty$) ist für diese Funktion als größer anzusehen, verglichen mit der Zielmenge (0,1,2,3,4,5,6).
Schnell berechenbar:
Ist erfüllt, da es sich beim Modulo um eine einfache Rechenoperation handelt, ist diese auch schnell berechenbar.
Da wir gezeigt haben, dass es sich um eine Hashfunktion handelt, müssen wir nur noch die Kriterien für eine kryptografische Hashfunktion prüfen.
Einweg-Eigenschaft:
Ist nicht erfüllt, es ist möglich die Funktion umzukehren, um dann durch einfache Iteration zum x zu gelangen. Gehen wir von einem $h(x) = 3$ aus, dann können wir eine Gegenfunktion $f(n) = 3 + n * 7$ dazu bilden, die wir durchiterieren können. So kommen wir leicht auf das dazugehörige x.
Kollisionsresitent Schwach:
Ist nicht erfüllt, zu einem gegebenen x, lässt sich einfach ein y finden, zu dem gilt $x \neq y$ und $h(x) = h(y)$. Nehmen wir ein Beispiel. $h(12) = 12 \mod 7 = 5$ finden wir viele beliebige x, die zum selben Ergebnis führen. Durch folgende Funktion finden wir sehr einfach sehr viele x die zum

gleichen Ergebnis führen: $f(n) = 5 + n * 7$.
Kollisionsresistent Stark:
Auch hier können wir eine einfache Kombination aus x und y finden, bei der gilt $x \neq y$ und $h(x) = h(y)$. Auch hier können wir beispielhaft die Funktion $f(n) = 5 + n * 7$ verwenden. Mithilfe der Funktion können wir beliebig viele x berechnen, sodass gilt $h(x) = h(y)$ (beispielsweise $x = 12$ und $y = 19$).
Dadurch das diese Eigenschaften nicht erfüllt sind, können wir daraus schließen, dass es sich bei der Funktion zwar um eine Hashfunktion, aber **nicht** um eine kryptografische Hashfunktion handelt.
b) $g(x) = x \mod 12$
Zunächst überprüfen wir, ob es sich bei der Funktion um eine Hashfunktion handelt. Dazu betrachten wir die oben beschriebenen Kriterien.
Komprimierung:
Ist erfüllt, da wir durch das Modulo 12, egal wie hoch die Eingabe ist eine Zahl zwischen 0 und 11 erhalten werden.
Fast nie injektiv:
Ist erfüllt, die mögliche Definitionsmenge ($=\infty$) ist für diese Funktion als größer anzusehen, verglichen mit der Zielmenge (Zahlen zwischen 0 und 11).
Schnell berechenbar:
Ist erfüllt, da es sich beim Modulo um eine einfache Rechenoperation handelt, ist diese auch schnell berechenbar. Da wir gezeigt haben, dass es sich um eine Hashfunktion handelt, müssen wir nur noch die Kriterien für eine kryptografische Hashfunktion prüfen.
Einweg-Eigenschaft:
Siehe Begründung in a).
Kollisionsresitent Schwach:
Siehe Begründung in a).
Kollisionsresistent Stark:
Siehe Begründung in a).
Dadurch das diese Eigenschaften nicht erfüllt sind, können wir daraus schließen, dass es sich bei der Funktion zwar um eine Hashfunktion, aber **nicht** um eine kryptografische Hashfunktion handelt.

Aufgabe 2.2

Das Hashen mit benutzerspezifischen Informationen schützt besser gegen verschiedene Angriffe. Beispiele: Schutz vor Rainbow-Table-Angriffen: Bei Rainbow-Table-Angriffen werden vorgefertigte Tabellen genutzt, um Hashes effizient zu knacken. Durch die Verwendung führt dies zu individuelleren Hashes für jedes Passwort, insbesondere wenn Benutzer das gleiche Passwort verwenden. Der Angreifer müsste demnach zu jedem Hash eine neue Tabelle erstellen, was unpraktisch wäre. Erschweren von Brute-Force-Angriffen: Durch die Individualisierung erhöht sich gleichzeitig die Anzahl der möglichen Hashes.

Aufgabe 2.3

Alice und Bob wählen beide ihre Option. Beide können dann eine Hashfunktion $h_j = h(Option_j)$. Da es lediglich drei Optionen gibt, wäre dies einfach durch Brute-Forcing zu knacken. Durch das Hinzufügen eines Salts vor dem hashen durch beispielsweise eine Zufallszahl kann dieses Problem gelöst werden. Nachdem sie ihre Wahl offengelegt haben, kann der andere Spieler sicherstellen, dass

der ursprüngliche Zug nicht verändert wurde, indem er den bekanntgegebenen Zug hasht und mit dem zuvor erhaltenen Hash vergleicht.

Aufgabe 2.4

a) Der Wert, um das Rätsel zu lösen ist:

$$x = 0000f03052da432dae2eb04d0b2099e86e9ca8c2ba64b17e4c0accb1850516f1$$

Dafür wurden 834961 Iterationen benötigt die in 3.3620524406433105s durchgeführt wurden.
Das Suchrätsel wurde in Python entwickelt. Folgend ist der Code abgebildet:

```
import hashlib
import time
startTime = time.time()
puzzleID = 'BBSE_E01'
d = '0000f'
iteration = 0

while(True):
    hash=puzzleID+str(iteration)
    hash=hash.encode()
    hash=hashlib.sha256(hash).hexdigest()
    if (hash[:5]==d):
        break
    iteration+=1
print('Iterationen: '+str(iteration)+' Hash: ' +str(hash)+
'Time: '+ str(time.time()-startTime))
```

b)
Da Computer A der schnellste ist, dadurch dass er mehr Hashes in der Sekunde ausführen kann, gewinnt er, da er das Ziel als erster erreicht.
c)
Um die Gewinnchance zu erhöhen, könnten die anderen Computer mithilfe einer anderen Strategie ihre Gewinnchance erhöhen. Wenn Sie beispielsweise nach jeder Iteration eine Zufallszahl als Iterationsfaktor nehmen $iteration = RandomZahl()$. Da A in jedem Fall, wenn es sich um eine lineare Strategie handelt, wie bei $x = x + 1$ gewinnt, und die Gewinnchance bei B und C bei der Verwendung der Strategie bei null liegt, wird die Gewinnchance durch Zufallszahlen vergrößert.
d)
Man könnte das Rätsel Hashing-Leistungsabhängig gestalten. Demnach wäre der Gewinner nicht derjenige Computer, welcher das Rätsel in kürzester Zeit löst, sondern immer auch abhängig von der Hashing-Leistung. Beispielsweise, wenn A das Rätsel in einer Sekunde löst und B das Rätsel in einer Sekunde löst, dann wäre kein gleichstand, sondern B hätte gewonnen, da dieser weniger Hashing-Leistung hat und für das gleiche Rätsel mehr Zeit bekommt. Man könnte auch das Rätsel so gestalten, dass die Computer nur eine gewisse Anzahl an Versuchen bekommen, um das Rätsel zu lösen. Dadurch würde derjenige Computer gewinnen, welcher die Effizienteste Methode zur Lösung vorweisen kann und nicht der mit der schnellsten Hash-Leistung.

Aufgabe 2.5

Folgend würde ich den Merkle Tree mit 5 Blättern aufbauen:

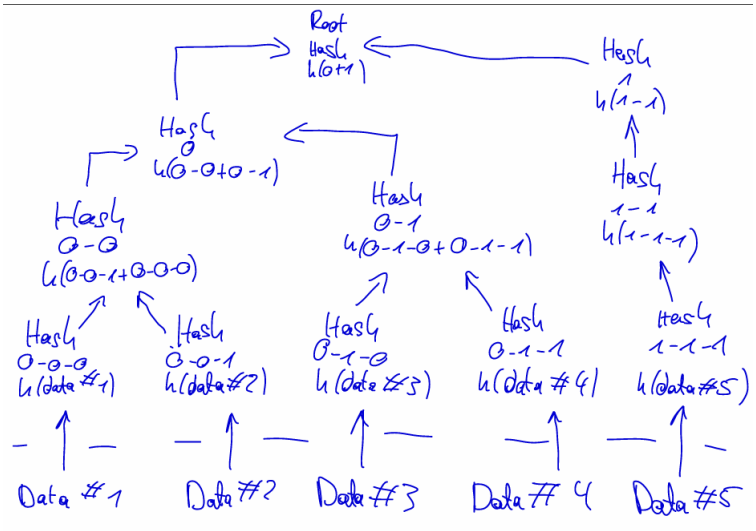

Abbildung 3: Merkle Tree mit 5 Blättern