Hashing

GF IF

Andreas Schärer

KSR

Erinnerung: Dictionaries

```
data = {
    'Genthod': {'area': 2.81, 'inhabitants': 2893, 'canton': 'GE'},
    'Gams': {'area': 22.28, 'inhabitants': 3587, 'canton': 'SG'},
    'Rorbas': {'area': 4.5, 'inhabitants': 2885, 'canton': 'ZH'},
    'Wohlen bei Bern': {'area': 36.25, 'inhabitants': 9240, 'canton
    'Kaisten': {'area': 18.09, 'inhabitants': 2754, 'canton': 'AG'}
    'Denens': {'area': 3.29, 'inhabitants': 742, 'canton': 'VD'}
    }
```

- Dictionary mit Infos zu allen CH-Gemeinden
- Info zu «Lupfig»:
  - `data['Lupfig']`
  - `# -> {'area': 8.45, 'inhabitants': 3157, 'canton': 'AG’}`
- Wo ist Info zu «Lupfig», also `{'area’:` …} auf Computer gespeichert?
- Speicherort muss *nicht gefunden* werden, wie wenn man mit Listen arbeitet …
- … sondern wird **berechnet** mithilfe von Key.
- Vorteil: Sehr schnell, (praktisch keine) Verlangsamung bei sehr grossen Datensätzen
- Doch wie macht es das?
- **Hashing** resp. **Hashfunktionen**

Hash

- «to hash»: zerhacken, verstreuen
- Idee:
  - Wende eine **Hash-Funktion** auf einen Input an, …
  - … z.B. auf einen String.
  - Hash-Funktion gibt **Hash** zurück, …
  - … typischerweise (Hex-) Zahl
  - Ist eine Art ‘**Signatur**’ für diesen Input zurück.
  - Hash ist typischerweise eine Zahl mit *fixer Bit-Länge*, z.B. 160 Bit

Einfache Hash-Funktion

- Algorithmus:
  - Summiere von allen Zeichen von Strings die ASCII-Werte zusammen …
  - … und rechne das Resultat % 16.
  - Resultat ist eine Zahl im Bereich 0,1,…,15
  - Resp. Hex-Zahl 0,1,2,…,F
- Beispiel 1: «ACDC»:
  - hash(«ACDC») = (65 + 67 + 68 + 67) % 16
  - = 267 % 16
  - = 11 (weil 267 // 16 = 16 Rest 11)
  - = 0xB (Hex)
- Beispiel 2: «KSR»
  - hash(«KSR») = 0x0

String	Hash
«ACDC»	0xB
«KSR»	0x0
«MATHEMATIK»	0x5
«MATH»	0xA
«COMPUTER»	0xF
«BODENSEE»	0x5
«INFORMATIK»	0x4
«INFORATIK»	0x7

Kleinste Änderung im Input
-> komplett anderer HASH

Unterschiedliche Inputs mit gleichem Hash!

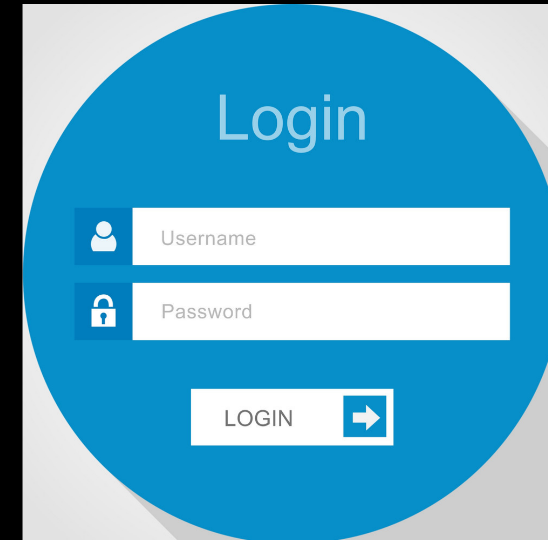
Hash

- Hashes sind **nicht eindeutig**!
- Unendlich viele Inputs produzieren gleichen Hash
- Ist also *nicht umkehrbar!*
  - *«MATHEMATIK» -> 0x5*
  - *«BODENSEE» -> 0x5*
  - 0x5 -> ??? (unendlich viele Möglichkeiten)
- Man verliert also Information, wenn man Hashfunktion anwendet
- Daher eignet sich *nicht* als *Verschlüsselungsmethode*
- Aber wozu sollen Hashes nützlich sein?

String	Hash
«ACDC»	0xB
«KSR»	0x0
«MATHEMATIK»	0x5
«MATH»	0xA
«COMPUTER»	0xF
«BODENSEE»	0x5
«INFORMATIK»	0x4
«INFORATIK»	0x7

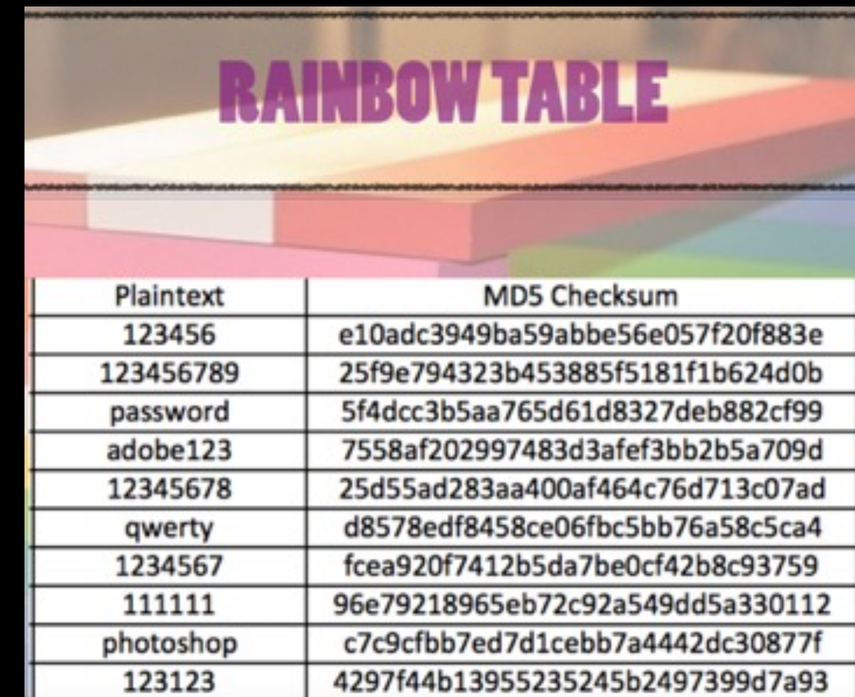
Hashes für Passwortspeicherung

- Website speichert in seiner Datenbank die Passwörter seiner Benutzerinnen (hoffentlich) *nicht im Klartext* sondern dessen Hashes
- Loggt man sich nun mit seinem Passwort ein, wird vom eingegebenen Passwort der Hash berechnet …
- … und mit demjenigen in Datenbank verglichen.
- D.h. man könnte sich auch Zugriff zur Website verschaffen mit einem ganz anderen Passwort, welches gleichen Hash hast.
- Dies ist bei professionellen Hash-Algorithmen aber so gut wie unmöglich.



Hashes für Passwortspeicherung

- Sorgt für mehr Sicherheit:
  - Betreiber der Website kennt also die Passwörter seiner Benutzer:innen nicht
  - Ebenso wenig Hacker, falls Website gehackt wird
- Probleme:
  - Websites verwenden alle die gleichen Standard Hash-Algorithmen
  - Menschen sind naiv (verwenden unsichere Passwörter)
- Vorgehen von Hackern:
  - Nehme Tabelle mit beliebten Passwörtern …
  - … und generiere Hashes davon mit bekannten Hash-Algorithmen
  - Vergleiche diese mit erbeuteten Hashes von gehackter Website
  - Solche vorgenerierten Hash-Tabellen heissen R**ainbow Tables**
- Deshalb weitere Sicherheitsmassnahme: «**Salt**»
  - Generiere «Salt» für jede Benutzerin: Zufallszahl, gespeichert auch in Datenbank
  - Addiere Salt zu Passwort, bevor Hash berechnet wird
  - Benutzerinnen mit identischen Passwörtern -> unterschiedliche Hashes
  - Rainbow Tables nutzlos


RAINBOW TABLE

Plaintext	MD5 Checksum
123456	e10adc3949ba59abbe56e057f20f883e
123456789	25f9e794323b453885f5181f1b624d0b
password	5f4dcc3b5aa765d61d8327deb882cf99
adobe123	7558af202997483d3afef3bb2b5a709d
12345678	25d55ad283aa400af464c76d713c07ad
qwerty	d8578edf8458ce06fbc5bb76a58c5ca4
1234567	fcea920f7412b5da7be0cf42b8c93759
111111	96e79218965eb72c92a549dd5a330112
photoshop	c7c9cfbb7ed7d1cebb7a4442dc30877f
123123	4297f44b13955235245b2497399d7a93

Hashes für Passwortspeicherung

- Schlechte Datenbank:

Username	Password
heiri@bluewin.ch	passwort123
ruthild@outlook.com	a37Z_1iaO9
maximilian@gmx.de	sauerkraut
gertrude@gmail.com	passwort123

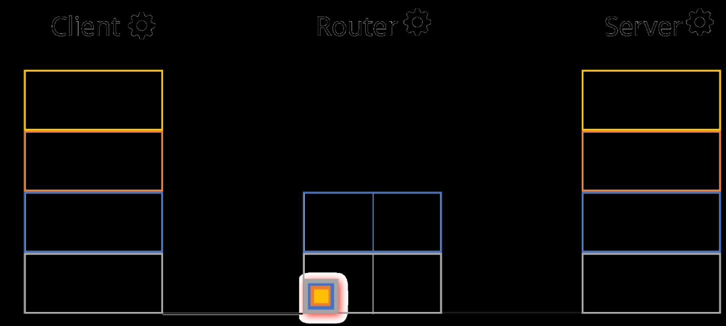
- Bessere Datenbank:

Username	Password
heiri@bluewin.ch	55789e79eca2f9a1e0786388b869f34f28a64ccbc37eb85ceeb031fd9677e06e
ruthild@outlook.com	be10c942d355ac2dc694d1036f907ac81731cf1ef812f33362b1682b34a2fd79
maximilian@gmx.de	f52da4906b06344b7289dc6302ec2f6df31857fc257aa061b01c3dd22d83124f
gertrude@gmail.com	55789e79eca2f9a1e0786388b869f34f28a64ccbc37eb85ceeb031fd9677e06e

- Gute Datenbank:

Username	Salt	Password
heiri@bluewin.ch	b_6Z/4NgK4|F5L	a8132ec57c4c2adbf21b73e04bb738dc9e37c4c8965c0c7ef4513131b4520889
ruthild@outlook.com	}|C#TXs7jYpy3|t_	3b8d5154344311dd562954ccb7aff61b35df1dd0d922a4dd21ee1b83ec1b26ae
maximilian@gmx.de	N&/&d7qWeHHm#$XF	c14d1d7fb83c16fdbbab42098642e72a28016b016c6d1f84ef1a48bebfee0e83
gertrude@gmail.com	<t/UpgUZG,-;+3@N	6c8adba84577cf1df4e349935af5125b21fe634cc74ee3a234f16d71d17a37bd

Datenintegrität



- Übermitteln von Datei
- Versender berechnet Hash von Datei, genannt **Prüfsumme / Checksum** …
- … und übermittelt diese Zusammen mit Datei
- Empfänger berechnet selbst Checksum von erhaltener Datei …
- … und vergleicht mit derjenigen von Versender
- Falls identisch weiss man, dass Datei korrekt und komplett übermittelt wurde:
    - Nichts ging verloren
    - Wurde nicht manipuliert

Dictionaries

- Python verwendet intern **Hash-Tables** für die Implementierung von Dictionaries
- Beispiel (Gemeinde-Auftrag): `data['Lupfig']`
- Python berechnet Hash von Key (hier 'Lupfig')
- Hash gibt Speicherort von zugehörigem Wert an, hier: `{'area': 8.45, 'inhabitants': 3157, 'canton': 'AG'}`
- Vorteil Dictionary vs. Liste:
  - Speicherort wird nicht gesucht …
  - … sondern berechnet
  - -> praktisch unabhängig von Grösse von Datensatz
- Mehr zu Hash-Tables in Aufgabe

Bekannte Hash-Algorithmen

- **MD5**:
  - «Message Digest»
  - Produziert immer 128bit Hash
  - Früher Standard, mittlerweile aber nicht mehr sicher
  - Ist heute möglich, für gegebenen Hash ein Passwort zu finden, der gleichen Hash produziert
- **SHA-2**:
  - «Secure Hash Algorithm 2»
  - Verschiedene Versionen mit unterschiedlichen Längen: z.B. 256bit oder 512bit
  - Heutiger Standard
  - Gilt (noch?) als sicher
  - In Zukunft wird sich aber wohl SHA-3 durchsetzen