

Kryptografische Hash Funktionen

Dozent: Prof. Dr. Michael Eichberg

Kontakt: michael.eichberg@dhbw-mannheim.de

Basierend auf: *Cryptography and Network Security - Principles and Practice, 8th Edition, William Stallings*

Version: 2.2



Folien: [HTML] <https://delors.github.io/sec-hashfunktionen/folien.de.rst.html>

[PDF] <https://delors.github.io/sec-hashfunktionen/folien.de.rst.html.pdf>

Fehler melden:

<https://github.com/Delors/delors.github.io/issues>

1. HASHFUNKTIONEN - GRUNDLAGEN

Hashfunktionen

- Eine Hashfunktion H akzeptiert eine beliebig lange Nachricht M als Eingabe und gibt einen Wert fixer Größe zurück: $h = H(M)$.
- Wird oft zur Gewährleistung der Datenintegrität verwendet. Eine Änderung eines beliebigen Bits in M sollte mit hoher Wahrscheinlichkeit zu einer Änderung des Hashwerts h führen.
- Kryptographische Hashfunktionen werden für Sicherheitsanwendungen benötigt.
Mögliche Anwendungen:
 - Authentifizierung von Nachrichten
 - Digitale Signaturen
 - Speicherung von Passwörtern

Beispiel: Berechnung von Hashwerten mittels MD5

`md5("Hello") = 8b1a9953c4611296a827abf8c47804d7`

`md5("hello") = 5d41402abc4b2a76b9719d911017c592`

`md5("Dieses Passwort ist wirklich total sicher
und falls Du es mir nicht glaubst, dann
tippe es zweimal hintereinander blind
fehlerfrei ein.")
= 8fcf22b1f8327e3a005f0cba48dd44c8`

Warnung

Die Verwendung von MD5 dient hier lediglich der Illustration. In realen Anwendung sollte MD5 nicht mehr verwendet werden.

Sicherheitsanforderungen an kryptografische Hashfunktion I

Variable Eingabegröße:

H kann auf einen Block beliebiger Größe angewendet werden.

Pseudozufälligkeit:

Die Ausgabe von H erfüllt die Standardtests für Pseudozufälligkeit.

Einweg Eigenschaft:

Es ist rechnerisch/praktisch nicht machbar für einen gegebenen Hashwert h ein N zu finden so dass gilt: $H(N) = h$

(🚩 *Preimage resistant; one-way property*)

Sicherheitsanforderungen an kryptografische Hashfunktion II

Schwache Kollisionsresistenz:

Es ist rechnerisch nicht machbar für eine gegebene Nachricht M eine Nachricht N zu finden so dass gilt: $M \neq N$ mit $H(M) = H(N)$

(🇺🇸 *Second preimage resistant; weak collision resistant*)

Starke Kollisionsresistenz:

Es ist rechnerisch unmöglich ein paar (N, M) zu finden so dass gilt:
 $H(M) = H(N)$.

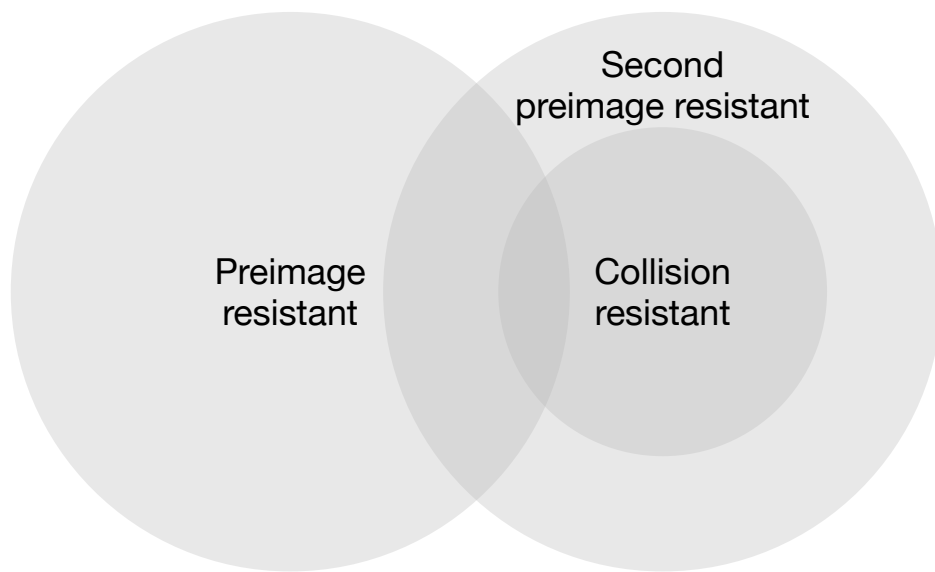
(🇺🇸 *Collision resistant; strong collision resistant*)

Hintergrund

Im Deutschen wird auch von Urbild-Angriffen gesprochen. In dem Fall ist *preimage resistance* (d.h. die Einweg Eigenschaft) gleichbedeutend damit, dass man nicht effektiv einen „Erstes-Urbild-Angriff“ durchführen kann. Hierbei ist das Urbild die ursprüngliche Nachricht M , die *gehasht* wurde.

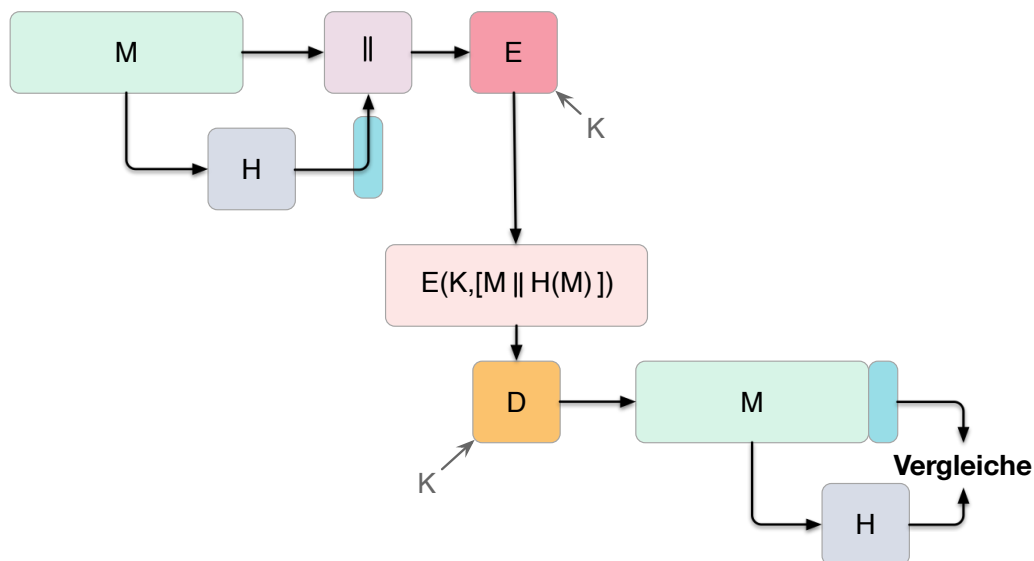
Second preimage resistance ist dann gleichbedeutend damit, dass man nicht effektiv einen „Zweites-Urbild-Angriff“ durchführen kann. Es ist nicht möglich zu einer Nachricht M eine zweite Nachricht N (d.h. ein zweites Urbild) zu finden, die für eine gegebene Hashfunktion den gleich Hash aufweist.

Beziehung zwischen den Sicherheitsanforderungen an Hashfunktionen

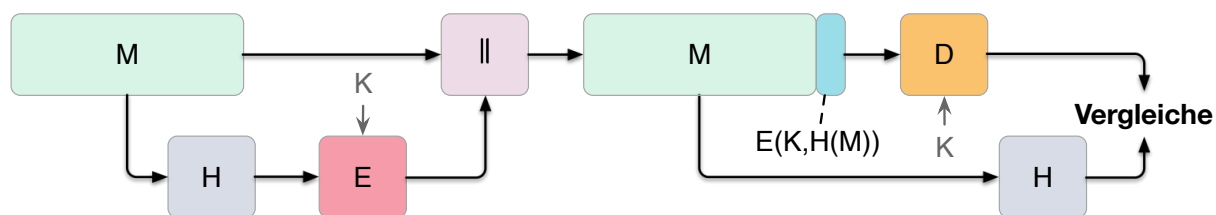


Nachrichtenauthentifizierung - vereinfacht

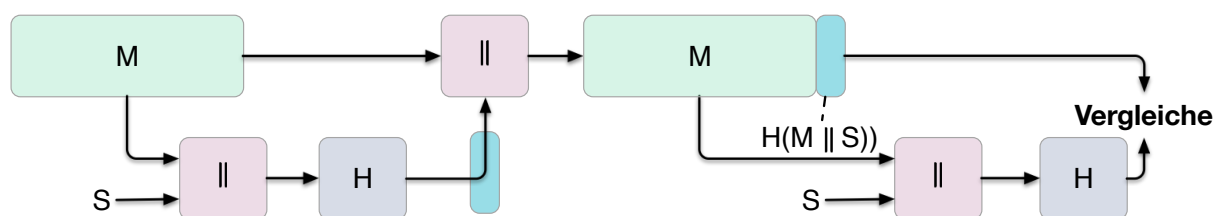
Nachrichten können auf verschiedene Weisen authentifiziert werden, so dass *Man-in-the-Middle-Angriffe* (MitM)^[1] verhindert werden können.



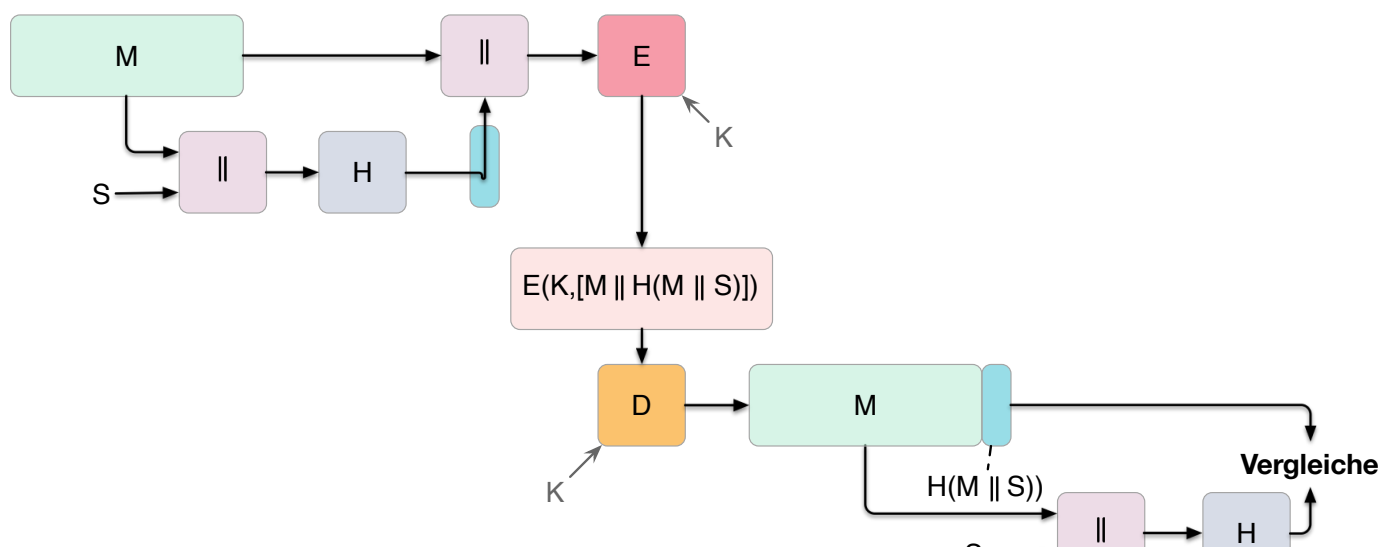
1



2



3



[1]  *Man* ist hier geschlechtsneutral zu verstehen.

Szenarien

Im ersten Szenario wird der Hash an die Nachricht angehängt und als ganzes verschlüsselt. Wir erhalten Vertraulichkeit und Authentizität.

Im zweiten Szenario wird der Hash der Nachricht berechnet und dann verschlüsselt. Der Empfänger kann den Hash berechnen und mit dem entschlüsselten Hash vergleichen. Wir erhalten Authentizität, aber keine Vertraulichkeit.

Im dritten Szenario wird an die Nachricht ein geteiltes Secret angehängt und alles zusammen gehasht. Die Nachricht wird dann mit dem Ergebnis der vorhergehenden Operation zusammen verschickt.

Im letzten Szenario werden alle Ansätze kombiniert.

Legende

M:	die Nachricht
H:	die Hashfunktion
E:	der Verschlüsselungsalgorithmus
D:	der Entschlüsselungsalgorithmus
K:	ein geheimer Schlüssel
S:	eine geheime Zeichenkette
 :	die Konkatenation von zwei Werten (d. h. das Aneinanderhängen von zwei Werten)

Hinweis

Bei *Man-in-the-Middle-Angriffen* handelt es sich um einen Fachbegriff und häufig wird zum Beispiel Eve oder Mallory verwendet, um die Person zu bezeichnen, die den Angriff durchführt. Gelegentlich wird auch *Adversary-in-the-Middle* oder *Person-in-the-Middle* verwendet.

Message-Digests

Im allgemeinen Sprachgebrauch wird auch von  *Message Digests* gesprochen.

Digitale Signaturen - vereinfacht

Digitale Signaturen dienen dem Nachweis der Authentizität einer Nachricht und der Integrität der Nachricht. Jeder, der einen öffentlichen Schlüssel hat, kann die Signatur überprüfen, aber nur der Besitzer des privaten Schlüssels kann die Signatur erstellen.

Legende

M:	die Nachricht
H:	die Hashfunktion
E:	der Verschlüsselungsalgorithmus
D:	der Entschlüsselungsalgorithmus
PR_a :	der private Schlüssel von a
PU_a :	der öffentliche Schlüssel von a
 :	die Konkatenation von zwei Werten (d. h. das Aneinanderhängen von zwei Werten)

Anforderungen an die Resistenz von Hashfunktionen

	Preimage Resistant	Second Preimage Resistant	Collision Resistant
Hash + Digitale Signaturen	✓	✓	✓
Einbruchserkennung und Viruserkennung		✓	
Hash + Symmetrische Verschlüsselung			
Passwortspeicherung	✓		
MAC	✓	✓	✓

Einbruchserkennung und Viruserkennung - Hintergrund

Bei der Einbruchserkennung und Viruserkennung ist *second preimage* Resistenz erforderlich. Andernfalls könnte ein Angreifer seine Malware so schreiben, dass diese einen Hash wie eine vorhandene gutartige Software hat und so verhindern, dass die Malware auf eine schwarze Liste gesetzt werden kann, ohne den Kollateralschaden, dass auch die gutartige Software fälschlicherweise als Malware erkannt wird.

Aufwand eines Kollisionsangriffs

Ein Kollisionsangriff erfordert weniger Aufwand als ein *preimage* oder ein *second preimage* Angriff.

Dies wird durch das Geburtstagsparadoxon erklärt. Wählt man Zufallsvariablen aus einer Gleichverteilung im Bereich von 0 bis $N - 1$, so übersteigt die Wahrscheinlichkeit, dass ein sich wiederholendes Element gefunden wird, nach \sqrt{N} Auswahlen $0,5$. Wenn wir also für einen m -Bit-Hashwert Datenblöcke zufällig auswählen, können wir erwarten, zwei Datenblöcke innerhalb von $\sqrt{2^m} = 2^{m/2}$ Versuchen zu finden.

Beispiel

Es ist relativ einfach, ähnliche Meldungen zu erstellen. Wenn ein Text 8 Stellen hat, an denen ein Wort mit einem anderen ausgetauscht werden kann, dann hat man bereits 2^8 verschiedene Texte.

Es ist relativ trivial(1), vergleichbare(2) Nachrichten(3) zu schreiben(4). Wenn ein Text 8 Stellen hat, an denen ein Ausdruck(5) mit einem vergleichbaren (6) ausgetauscht werden kann, dann erhält(7) man bereits 2^8 verschiedene Dokumente(8).

Effizienzanforderungen an kryptografische Hashfunktionen

Effizienz bei der Verwendung für Signaturen und zur Authentifizierung:

Bei der Verwendung zur Nachrichtenauthentifizierung und für digitale Signaturen ist $H(N)$ für jedes beliebige N relativ einfach zu berechnen. Dies soll sowohl Hardware- als auch Softwareimplementierungen ermöglichen.

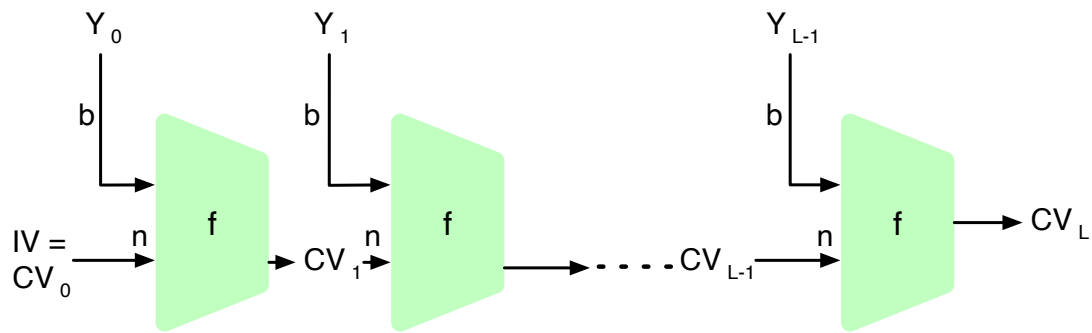
VS.

Brute-Force-Angriffe auf Passwörter erschweren:

Bei der Verwendung für das Hashing von Passwörtern soll es schwierig sein den Hash effizient zu berechnen, selbst auf spezialisierter Hardware (GPUs, ASICs).

Struktur eines sicheren Hash-Codes

(Vorgeschlagen von Merkle.)



IV = Initialer Wert (Algorithmus-abhängig)

n = Länge des Blocks

CV_i = Verkettungsvariable

L = Anzahl der Eingabeblocks

Y_i = i -er Eingabeblock

b = Länge des Eingabeblocks

f = Kompressionsfunktion

Diese Struktur liegt insbesondere den Hashfunktionen der SHA-2 Familie zugrunde.

XOR als Hashfunktion

Warum ist eine einfache „Hash-Funktion“, die einen 256-Bit-Hash-Wert berechnet, indem sie ein XOR über alle 256-Bit Blöcke einer Nachricht durchführt, im Allgemeinen ungeeignet?

Wir nehmen hier an, dass die Nachricht ein Vielfaches von 256 Bit lang ist. Falls nicht, dann wenden wir Padding an.

XOR als Hashfunktion

Warum ist eine einfache „Hash-Funktion“, die einen 256-Bit-Hash-Wert berechnet, indem sie ein XOR über alle 256-Bit Blöcke einer Nachricht durchführt, im Allgemeinen ungeeignet?

Wir nehmen hier an, dass die Nachricht ein Vielfaches von 256 Bit lang ist. Falls nicht, dann wenden wir Padding an.

Bewertung der Sicherheit

- Eine Nachricht M bestehe aus N 64-bit Blöcken: X_1, \dots, X_n .
- Der Hashcode $H(M)$ ist ein simpler XOR über alle Blöcke:
$$H(M) = h = X_1 \oplus X_2 \oplus \dots \oplus X_n.$$
- h wird als der X_{N+1} Block an die Nachricht angehängt und danach wird unter Verwendung des CBC Modus die Nachricht inkl. des Hashcodes verschlüsselt ($C = Y_1, \dots, Y_{N+1}$).
- Gegen welche Art von Manipulation ist diese Konstruktion *nicht* sicher?

Studieren Sie ggf. noch einmal den CBC Modus.

Bewertung der Sicherheit

- Eine Nachricht M bestehe aus N 64-bit Blöcken: X_1, \dots, X_n .
- Der Hashcode $H(M)$ ist ein simpler XOR über alle Blöcke: $H(M) = h = X_1 \oplus X_2 \oplus \dots \oplus X_n$.
- h wird als der X_{N+1} Block an die Nachricht angehängt und danach wird unter Verwendung des CBC Modus die Nachricht inkl. des Hashcodes verschlüsselt ($C = Y_1, \dots, Y_{N+1}$).
- Gegen welche Art von Manipulation ist diese Konstruktion *nicht* sicher?

Studieren Sie ggf. noch einmal den CBC Modus.

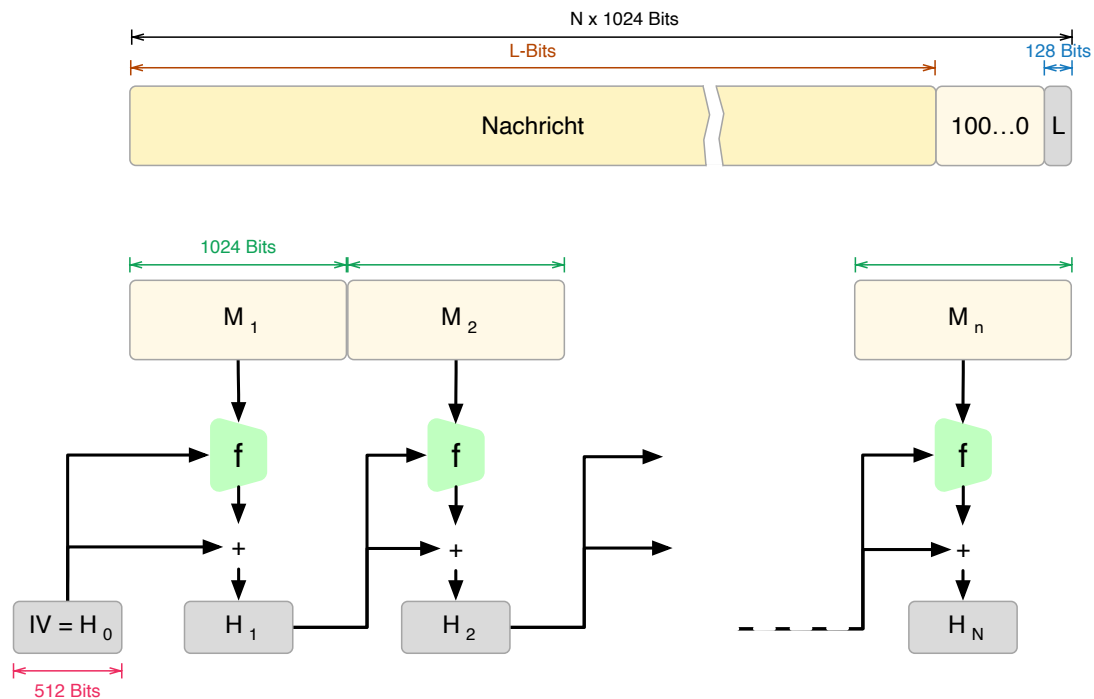
Irrelevanz von Second-Preimage-Resistenz und Kollisionssicherheit

Warum sind *Second-Preimage-Resistenz* und Kollisionssicherheit von nachgeordneter Relevanz, wenn der Hash-Algorithmus zum Hashing von Passwörtern verwendet wird?

Irrelevanz von Second-Preimage-Resistenz und Kollisionssicherheit

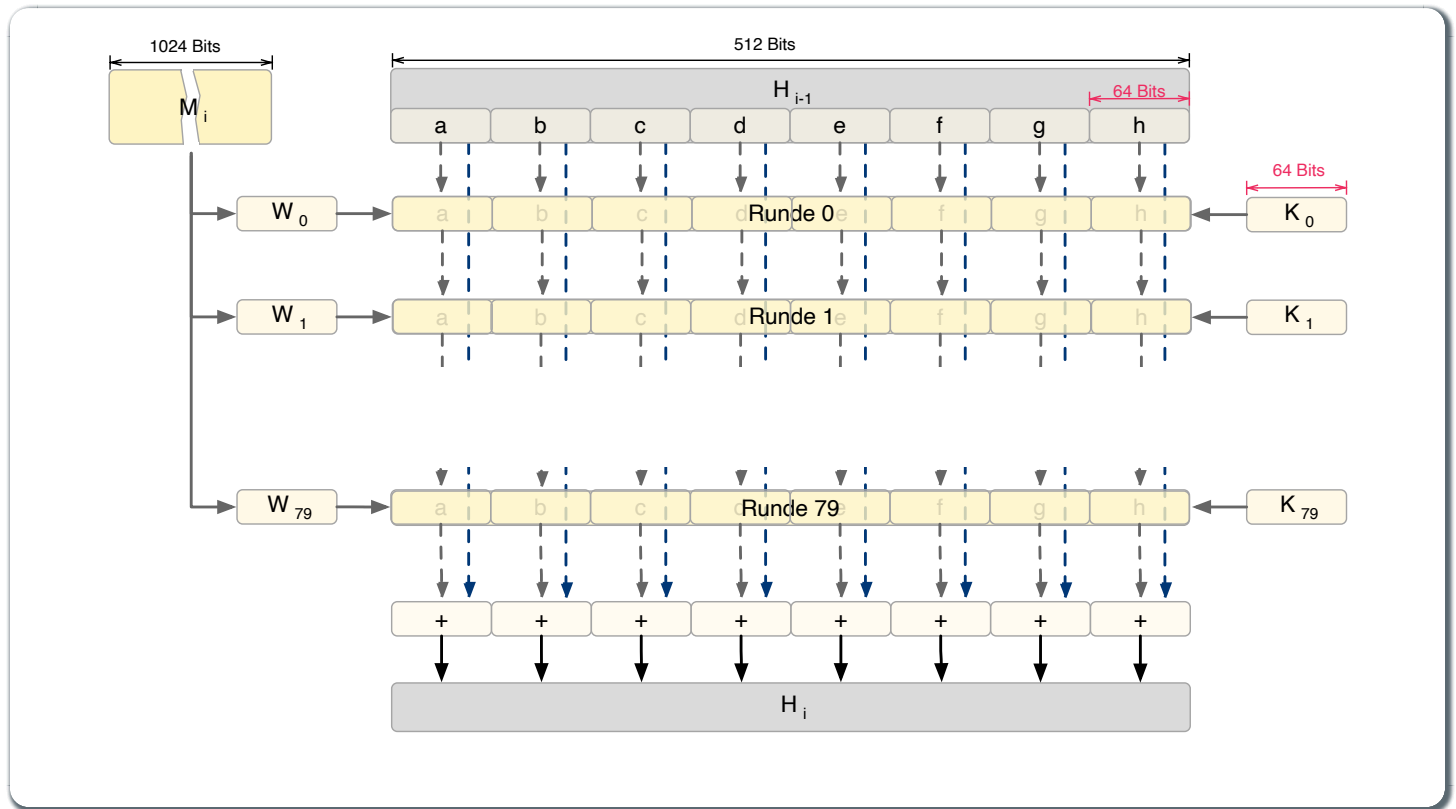
Warum sind *Second-Preimage-Resistenz* und Kollisionssicherheit von nachgeordneter Relevanz, wenn der Hash-Algorithmus zum Hashing von Passwörtern verwendet wird?

SHA 512 - Übersicht



- SHA-512 nimmt eine Nachricht beliebiger Größe und gibt einen 512-Bit-Hashwert zurück.
- Der IV von SHA-512 besteht aus den folgenden acht 64-Bit-Zahlen:
`6a09e667f3bcc908`
`bb67ae8584caa73b`
`3c6ef372fe94f82b`
`a54ff53a5f1d36f1`
`510e527fade682d1`
`9b05688c2b3e6c1f`
`1f83d9abfb41bd6b`
`5be0cd19137e2179`
- Die Addition erfolgt Wortweise modulo 2^{64} .
- Die Nachricht wird in 1024-Bit-Blöcke unterteilt. Die Nachricht wird - unabhängig von der tatsächlichen Länge - immer aufgefüllt (`padding`) und auf eine Länge $l \equiv 896 \pmod{1024}$ Bits gebracht.
- Das Padding besteht aus einem Bit mit Wert 1, gefolgt von der notwendigen Anzahl Nullen.
- Am Ende wird die Länge der Nachricht als 128-Bit-Wert angehängt, um ein Vielfaches von 1024 zu erhalten.

SHA-512 Verarbeitung eines 1024-Bit-Blocks



Die Additionen erfolgen Modulo 2^{64} .

Berechnung der W_i

W_0 bis W_{15} sind die ersten 16 Wörter des 1024-Bit-Blocks. Die restlichen 64 Wörter werden wie folgt berechnet.

$$w_t = \sigma_1(w_{t-2}) + w_{t-7} + \sigma_0(w_{t-15}) + w_{t-16}$$

Mit:

$$\sigma_0(x) = (x \ggg 1) \oplus (x \ggg 8) \oplus (x \gg 7)$$

$$\sigma_1(x) = (x \ggg 19) \oplus (x \ggg 61) \oplus (x \gg 6)$$

Legende

\gg ist die Rechtsverschiebung bei der die linke Seite mit Nullen aufgefüllt wird.

\ggg ist die zyklische Rechtsverschiebung.

\oplus steht für die bitweise XOR-Operation.

Rundenfunktion

$$T_1 = h + Ch(e, f, g) + \left(\sum_1^{512} e\right) + K_t + W_t$$

$$T_2 = \left(\sum_0^{512} a\right) + Maj(a, b, c)$$

$$h = g$$

$$g = f$$

$$f = e$$

$$e = d + T_1$$

$$d = c$$

$$c = b$$

$$b = a$$

$$a = T_1 + T_2$$

Weiterhin gilt:

t ist die Schrittnummer

$$Ch(e, f, g) = (e \wedge f) \oplus (\neg e \wedge g)$$

$$Maj(a, b, c) = (a \wedge b) \oplus (a \wedge c) \oplus (b \wedge c)$$

$$\sum_0^{512} a = a \ggg 28 \oplus a \ggg 34 \oplus a \ggg 39$$

$$\sum_1^{512} e = e \ggg 14 \oplus e \ggg 18 \oplus e \ggg 41$$

Berechnung der Konstanten

Die Konstanten K sind die ersten 64 Bits der Bruchteile der Kubikwurzeln der ersten 80 Primzahlen.

Konzeptionell: $((\sqrt[3]{n_{te} \text{ Primzahl}} - \lfloor \sqrt[3]{n_{te} \text{ Primzahl}} \rfloor) < 64) \cdot \text{toInt}().\text{toString}(16)$

Die SHA-512 Konstanten

```
428a2f98d728ae22 7137449123ef65cd b5c0fbcfec4d3b2f e9b5dba58189dbbc 3956c25bf348b538 59f111f1b605d019 923f82a4af194f9b
ab1c5ed5da6d8118 d807aa98a3030242 12835b0145706fbe 243185be4ee4b28c 550c7dc3d5ffb4e2 72be5d74f27b896f 80deb1fe3b1696b1
9bdc06a725c71235 c19bf174cf692694 e49b69c19ef14ad2 efbe4786384f25e3 0fc19dc68b8cd5b5 240ca1cc77ac9c65 2de92c6f592b0275
4a7484aa6ea6e483 5cb0a9dcdbd41fbd4 76f988da831153b5 983e5152ee66dfab a831c66d2db43210 b00327c898fb213f bf597fc7beef0ee4
c6e00bf33da88fc2 d5a79147930aa725 06ca6351e003826f 142929670a0e6e70 27b70a8546d22ffc 2e1b21385c26c926 4d2c6dfc5ac42aed
53380d139d95b3df 650a73548baf63de 766a0abb3c77b2a8 81c2c92e47edae6 92722c851482353b a2bfe8a14cf10364 a81a664bbc423001
c24b8b70d0f89791 c76c51a30654be30 d192e819d6ef5218 d69906245565a910 f40e35855771202a 106aa07032bbd1b8 19a4c116b8d2d0c8
1e376c085141ab53 2748774cdf8eeb99 34b0bcb5e19b48a8 391c0cb3c5c95a63 4ed8aa4ae3418acb 5b9cca4f7763e373 682e6ff3d6b2b8a3
748f82ee5defb2fc 78a5636f43172f60 84c87814a1f0ab72 8cc702081a6439ec 90befffa23631e28 a4506cebd82bde9 bef9a3f7b2c67915
c67178f2e372532b ca273eceeaa26619c d186b8c721c0c207 eada7dd6cde0eb1e f57d4f7fee6ed178 06f067aa72176fba 0a637dc5a2c898a6
113f9804bef90dae 1b710b35131c471b 28db77f523047d84 32caab7b40c72493 3c9ebe0a15c9bebc 431d67c49c100d4c 4cc5d4becb3e42b6
597f299cfc657e2a 5fcb6fab3ad6faec 6c44198c4a475817
```

Exemplarischer Code zur Berechnung der Konstanten

Der folgende JavaScript Code demonstriert, wie die Konstanten für SHA-512 berechnet werden bzw. wurden. Die Präzision von Standard JavaScript Gleitkommazahlen (64-Bit Double) ist jedoch nicht ganz ausreichend, um die Konstanten vollständig zu berechnen.

```
1 const primes = [
2   2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59,
3   61, 67, 71, 73, 79, 83, 89, 97, 101, 103, 107, 109, 113, 127,
4   131, 137, 139, 149, 151, 157, 163, 167, 173, 179, 181, 191, 193,
5   197, 199, 211, 223, 227, 229, 233, 239, 241, 251, 257, 263, 269,
6   271, 277, 281, 283, 293, 307, 311, 313, 317, 331, 337, 347, 349,
7   353, 359, 367, 373, 379, 383, 389, 397, 401, 409 ];
8
9 function* genSHA512Constants() {
10   let i = 0;
11   while(i < 80) {
12     const p = primes[i];
13     const cubeRootP = Math.cbrt(p); // == p ** (1/3);
14     yield (cubeRootP - Math.floor(cubeRootP));
15     ++i;
16   }
17 }
18 for(const c of genSHA512Constants()) {
19   console.log(c.toString(16));
20 }
```

Der folgenden Java Code demonstriert, wie die Konstanten für SHA-512 berechnet werden können. Wir verwenden hier die Klasse *BigDecimal*, um die Konstanten zu berechnen, da Java keinen `long double` Typ (mit

128 Bit) kennt.

```
1 /**
2  * Compute the cube root using BigDecimals and the Newton-Raphson
3  * algorithm.
4  *
5  * @param n the number for which the cube root should be computed.
6  * @param guess the current/initial guess. Can be BigDecimal.ONE.
7  * @param steps the number of steps to be executed. The algorithm is
8  *             iterative and the number of steps determines the
9  *             precision of the result.
10 */
11 BigDecimal cbrt(BigDecimal n, BigDecimal guess, int steps) {
12     if (steps == 0) return guess;
13     final var newGuess =
14         guess.add(
15             guess.pow(3).add(n.negate()).divide(
16                 guess.pow(2).multiply(new BigDecimal(3)),
17                 MathContext.DECIMAL128
18             ).negate()
19         );
20     return cbrt(n, newGuess, steps - 1);
21 }
22 /**
23  * Given a prime number get the first 64 bits of the fractional
24  * part of the cube root.
25  */
26 String shaConstant(int prime) {
27     final var cubeRoot = cbrt(new BigDecimal(prime), BigDecimal.ONE, 16);
28     // "extract" the fractional by computing modulo 1
29     final var fractionalPart = cubeRoot.remainder(BigDecimal.ONE);
30     // To extract the first 64 bits we effectively do a shift-left
31     // by 64 which we simulate by multiplying with 2^64
32     final var bits = fractionalPart.multiply(BigDecimal.TWO.pow(64));
33     // to get the HEX representation we use BigInteger's toString
34     // method as a convenience method
35     return bits.toBigInteger().toString(16);
36 }
```

2. *MESSAGE AUTHENTICATION CODES* (MACs)

Hinweis

Message Authentication Codes könnte ins Deutsche mit Nachrichtenauthentifizierungscodes übersetzt werden, dies ist aber nicht üblich.

Im allgemeinen (IT-)Sprachgebrauch wird von *MACs* gesprochen.

HMAC (Hash-based Message Authentication Code)

Auch als *keyed-hash message authentication code* bezeichnet.

$$\begin{aligned}HMAC(K, m) &= H((K' \oplus opad) || H((K' \oplus ipad) || m)) \\ K' &= \begin{cases} H(K) & \text{falls } K \text{ größer als die Blockgröße ist} \\ K & \text{andernfalls} \end{cases}\end{aligned}$$

H ist eine kryptografische Hashfunktion.

m ist die Nachricht.

K ist der geheime Schlüssel (*Secret Key*).

K' ist vom Schlüssel K abgeleiteter Schlüssel mit Blockgröße (ggf. *padded* oder *gehasht*).

$||$ ist die Konkatenation.

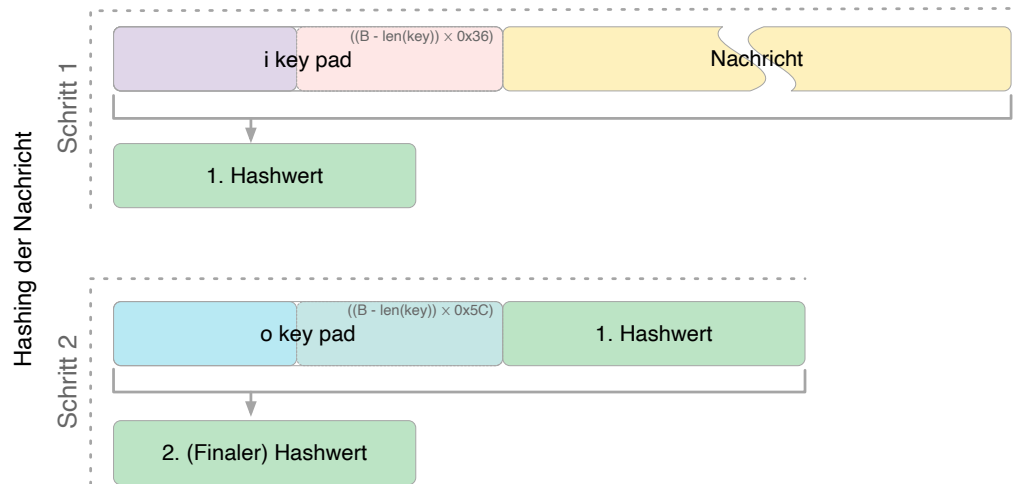
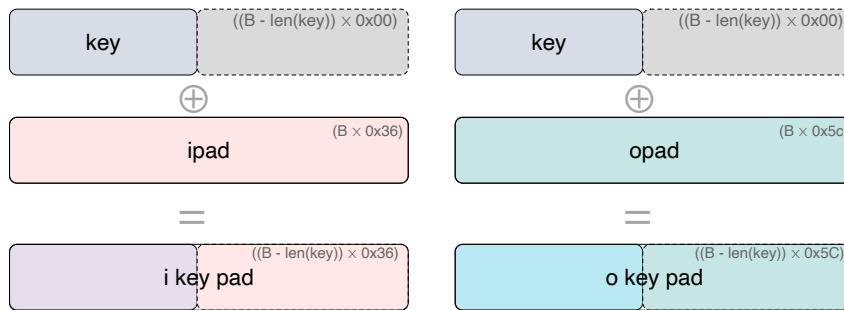
\oplus ist die XOR Operation.

opad ist das äußere Padding bestehend aus Wiederholungen von 0x5c in Blockgröße.

ipad ist das innere Padding bestehend aus Wiederholungen von 0x36 in Blockgröße.

Ableitung der Schlüssel, die jeweils in die Hashberechnung eingehen.

Die Blockgröße sei B bytes und der Schlüssel (Key) sei kleiner.



Padding und Hashing

Im Rahmen der Speicherung von Passwörtern und *Secret Keys* ist die Verwendung von Padding Operationen bzw. das Hashing von Passwörtern, um Eingaben in einer wohl-definierten Länge zu bekommen, üblich. Neben dem hier gesehenen Padding, bei dem $0x00$ Werte angefügt werden, ist zum Beispiel auch das einfache Wiederholen des ursprünglichen Wertes, bis man auf die notwendige Länge kommt, ein Ansatz.

Diese Art Padding darf jedoch nicht verwechselt werden mit dem Padding, dass ggf. im Rahmen der Verschlüsselung von Nachrichten notwendig ist, um diese ggf. auf eine bestimmte Blockgröße zu bringen (zum Beispiel bei ECB bzw. CBC Block Mode Operations.)

HMAC Berechnung in Python

Implementierung

```
import hashlib
pwd = b"MyPassword"
stretched_pwd = pwd + (64-len(pwd)) * b"\x00"

ikeypad = bytes(map(lambda x : x ^ 0x36 , stretched_pwd)) # xor with ipad
okeypad = bytes(map(lambda x : x ^ 0x5c , stretched_pwd)) # xor with opad

hash1 = hashlib.sha256(ikeypad+b"JustAMessage").digest()
hmac = hashlib.sha256(okeypad+hash1).digest()

# hmac =
# b'\xab\xa0\xd9\xe2\x8a\xc8\x081\xe\x1b\x1d,
# \x8f\xa6\xd6L\x94\xab\x89\x9a\x89*\xc7\x0f_no\xc1xdc6\xfc'
```

HMAC ist auch direkt als Bibliotheksfunktion verfügbar.

```
import hashlib
import hmac

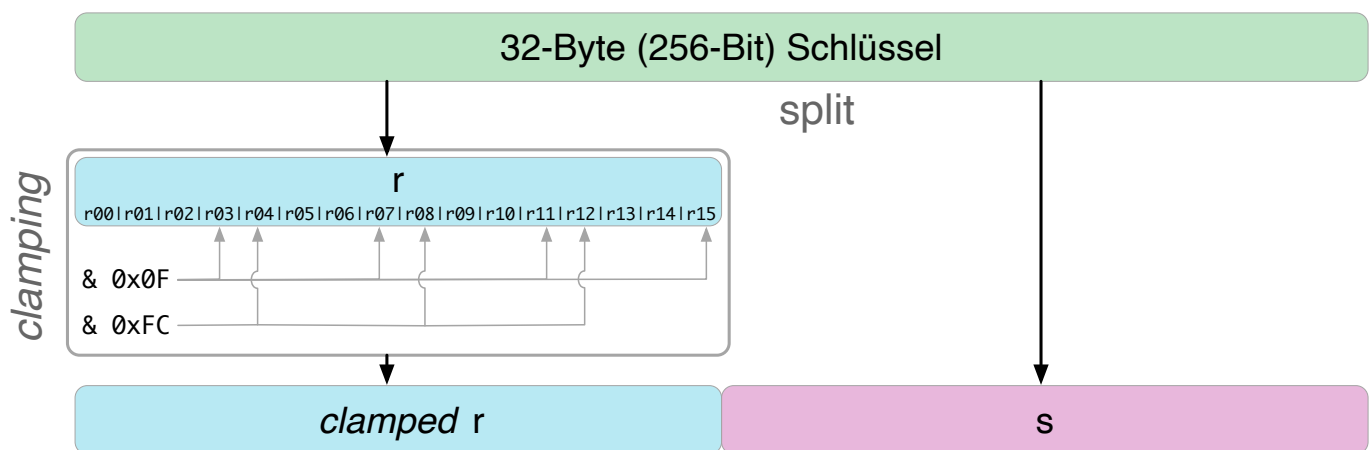
hash_hmac = hmac.new(
    b"MyPassword",
    b"JustAMessage",
    hashlib.sha256).digest()

hash_hmac =
b'\xab\xa0\xd9\xe2\x8a\xc8\x081\xe\x1b\x1d,
\x8f\xa6\xd6L\x94\xab\x89\x9a\x89*\xc7\x0f_no\xc1xdc6\xfc'
```

MAC: Poly 1305

- Ein MAC Algorithmus für die Einmalauthentifizierung von Nachrichten.
- Entwickelt von Daniel J. Bernstein.
- Basierend auf einem 256-Bit-Schlüssel und einer Nachricht wird ein 128-Bit-Tag berechnet.
- In Verbindung mit *ChaCha20* in einer Reihe von Protokollen verwendet.

1



Aufteilung des Schlüssels

2

Verarbeitung der Nachricht

- initialisiere den Akkumulator a mit 0
- die Nachricht wird in Blöcke von 16 Byte aufgeteilt und als *little-endian* Zahl verarbeitet; d. h. ein Block hat 16 Oktette (16×8 Bit)
- Füge dem Block n ein Bit jenseits der Anzahl der Oktette des aktuellen Blocks hinzu $= n'$; d. h. im Falle eines 16-Byte-Blocks wird die Zahl 2^{128} addiert
(Danach haben wir ggf. eine 17-Byte-Zahl.)
- Addiere n' aus dem letzten Schritt zum Akkumulator a und multipliziere mit **clamped r**
- Aktualisiere den Akkumulator mit dem Ergebnis *modulo* P mit $P = 2^{130} - 5$:
$$a = ((a + n') \times \text{clamped r}) \bmod P$$

3

Beispiel

```
000 43 72 79 70 74 6f 67 72 61 70 68 69 63 20 46 6f Cryptographic Fo
016 72 75 6d 20 52 65 73 65 61 72 63 68 20 47 72 6f rum Research Gro
032 75 70 up
```

Sei $\text{clamped } r = 806d5400e52447c036d555408bed685$

Sei $s = 1bf54941aff6bf4afdb20dfb8a800301$

Verarbeitung des ersten Blocks

$$\begin{aligned}a &= 00 \\n &= 6f4620636968706172676f7470797243 \\n' &= 016f4620636968706172676f7470797243 \\a + n' &= 016f4620636968706172676f7470797243 \\(a + n') \times \text{clamped } r &= b83fe991ca66800489155dcd69e8426ba2779453994ac90ed284034da565ecf \\a = ((a + n') \times \text{clamped } r) \bmod P &= 2c88c77849d64ae9147ddeb88e69c83fc\end{aligned}$$

1

Verarbeitung des letzten Blocks

$$\begin{aligned}a &= 2d8adaf23b0337fa7cccfb4ea344b30de \\n &= 7075 \\n' &= 017075 \\a + n' &= 2d8adaf23b0337fa7cccfb4ea344ca153 \\(a + n') \times \text{clamped } r &= 16d8e08a0f3fe1de4fe4a15486aca7a270a29f1e6c849221e4a6798b8e45321f \\a = ((a + n') \times \text{clamped } r) \bmod P &= 28d31b7caff946c77c8844335369d03a7\end{aligned}$$

2

Abschluss

Addiere auf den Wert des Akkumulators a den Wert s .

Und somit ist der Tag $2a927010caf8b2bc2c6365130c11d06a8$ (als Zahl im little-endian Format)

3

Hinweise

In dieser Diskussion betrachten wir jeden Block der Nachricht als „große Zahl“.

$$P = 2^{130} - 5 = 3ffb$$

Dadurch, dass wir den Block als Zahl in *little-endian* Reihenfolge interpretieren, ist das hinzufügen des Bits jenseits der Anzahl der Oktette gleichbedeutend damit, dass wir den Wert 0x01 am Ende des Blocks hinzufügen.

Zusammenfassung

- Ein Hashwert dient der Integritätssicherung von Nachrichten.
- Ein Mac dient der Authentifizierung von Nachrichten.
- Ein Mac sichert auch immer die Integrität der Nachricht.

Es ist somit möglich die Integrität einer Nachricht zu sichern ohne Authentizität zu gewährleisten, aber nicht umgekehrt.

- Ein Mac erlaubt es dem Empfänger eine gefälschte Nachricht zu erkennen aber ggf. auch zu erstellen (🚩 *to forge a message*).
- Eine Signatur basiert auf einem Hashwert und einem privaten Schlüssel.
- Der Empfänger kann bei einer signierten Nachricht, diese nicht verändern und als eine Nachricht des Senders ausgeben.
- Nur für Nachrichten, die signiert sind, gilt somit die Nichtabstreitbarkeit (🚩 *non-repudation*).