Hashfunktionen

Soviel Mathematik wie nötig, sowenig wie möglich

Wolfgang Dautermann

FH JOANNEUM

Chemnitzer Linuxtage 2010

1 Einleitung

2 Grundlagen von Hashfunktionen

3 Anwendungen von Hashfunktionen

4 „Angriffe“

5 kryptographische Hashfunktionen
- Message Digest 5 (MD5)
- Message Digest 4 (MD4)
- Secure Hash Algorithm 1 (SHA1)
- Secure Hash Algorithm 3

6 Gefahr von Kollisionen

7 Zusammenfassung

Grundlagen von Hashfunktionen

Ein Hash ist ein Algorithmus, der aus einer großen Datenmenge eine sehr kleine Zusammenfassung/Identifikation (einen Fingerabdruck) generiert.

- Aus einer (theoretisch[1]) beliebig langen Eingabe (Bitfolge) soll eine Ausgabe konstanter Länge erzeugt werden.
  $f : \{0,1\}^* \rightarrow \{0,1\}^n$
- klar definierter Algorithmus, keine Geheimnisse
- einfach zu berechnen (soll auch auf embedded devices machbar sein, geringer Speicher & CPU-Bedarf).

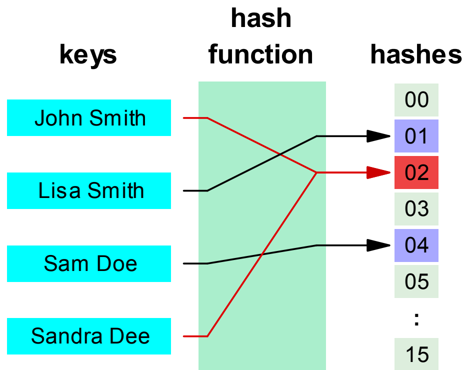
[1] SHA1 ist limitiert auf Inputgröße $< 2^{64}$ Bit $= 2^{56}$ Byte $=$ 64 Petabyte $\approx$ 64.000 TB

Grundlagen von Hashfunktionen

to hash: zerhacken, zerkleinern, faschieren

Grundlagen von Hashfunktionen

Abbildung einer großen Eingabemenge auf eine kleinere Ausgabemenge mittels einer Hashfunktion.



Anwendungen von Hashfunktionen

- Digitale Zertifikate (SSL: sha1 bzw. md5)
- Digitale Signaturen: In der Praxis unterschreibt man meist nicht die Nachricht, sondern ihren Hashwert (*Hash-then-sign*)
- Passwortverschlüsselung (Einwegverschlüsselung)
  - pam_unix: md5, sha2
  - htpasswd (Apache): md5, sha1
  - Postgres: md5
  - Weblogsystem Serendipity: md5, sha1
  - Mediawiki: md5
  - Mysql: sha1 (zweimal angewendet)
- Verifikation von Downloads (üblich: md5)

Anwendungen von Hashfunktionen

- Versionskontrollsysteme
  - Subversion: md5
  - Mercurial: sha1
  - Git: sha1
- Rsync: md4 (und zuvor eine einfachere schnellere Hashfunktion)
- IPv6: sha1, md5
- Datenblockverifikation bei Filesharingprogrammen (z.B. Bittorrent: sha1)
- Integritätsprüfung (ZFS: sha1)
- Openssl: md2, md4, md5, sha, sha1, sha224, sha256, sha384, sha512, mdc2, ripemd160
- Openssh HashKnownHosts: sha1

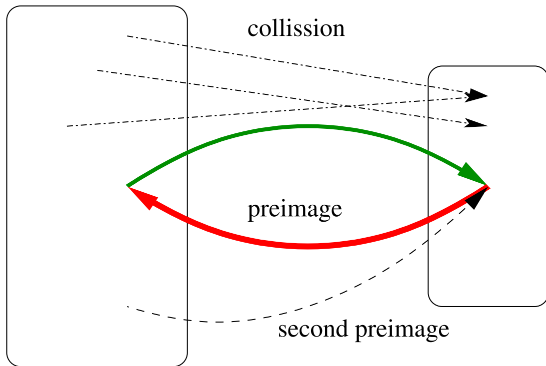
Kriterien für gute Hashfunktionen

- Geringe Kollisionswahrscheinlichkeit
- Gleichverteilung der Hashwerte.
- Effizienz: schnell berechenbar, geringer Speicherverbrauch, die Eingabedaten nur einmal lesen.
- Jeder Ergebniswert soll möglich sein (Surjektivität).
- Hashwert viel kleiner als Eingabedaten (Kompression)

Weitere Kriterien für kryptographische Hashfunktionen

- Chaos: „ähnliche“ Eingabedaten sollen sehr unterschiedliche Hashwerte erzeugen (ändern eines Bits soll ca. die Hälfte der Ausgabebits ändern)
- Preimage-Resistenz: Aus einem gegebenen Hash Value h soll es schwierig sein, eine Message M zu finden, die $h = hash(M)$ erfüllt.
- Second preimage Resistenz: Mit einer gegebenen Message M_1 soll es schwierig sein, eine andere Message M_2 ($M_1 \neq M_2$) zu finden, so dass gilt: $hash(M_1) = hash(M_2)$.
- Kollissionsresistenz: Es soll schwierig sein, 2 verschiedene Messages ($M_1 \neq M_2$) zu finden mit $hash(M_1) = hash(M_2)$.
- [Beinahe-Kollissionsresistenz: Es soll schwierig sein, 2 verschiedene Messages ($M_1 \neq M_2$) zu finden mit $hash(M_1)$ unterscheidet sich nur in wenigen Bits von $hash(M_2)$.]

Weitere Kriterien für kryptographische Hashfunktionen



zum Vergleich: Hashes in Programmiersprachen

Kriterien (für kryptographische Hashes) nicht notwendig

Hashes in Perl

```
%alter = ('Alice' => 28,
          'Bob'   => 30,
          'Eve'   => 30);
```

Auch in anderen Programmiersprachen (z.B. Ruby, Python) vorhanden.

„Angriffe“ (RFC4270[3])

Brechen der vorher definierten Kriterien (mit weniger Rechnenleistung als zu erwarten ist)

- Preimage-Angriff: Brechen der One-Way Eigenschaft, bei Passwörtern z.B. mit Wörterbuchattacken. (Abhilfe: Salted Hash)
- Second preimage Angriff: M_1 ist bekannt, Eigenschaften des Algorithmus könnten genutzt werden, um M_2 zu finden.
- Kollissionsangriff ist einfacher: (Geburtstagsparadoxon). Wenn die Rechenleistung kleiner als erwartet ist, ~~sollte man sich Gedanken machen~~ gilt die Hashfunktion als „geknackt“... (SHA1: 160 Bit Hashsize: $\Rightarrow 2^{80}$ Operationen sollten für einen Kollissionsangriff durchschnittlich nötig sein, aber ein Angriff mit 2^{52} Operationen wurde gefunden[2]

---

[2] Siehe `http://eprint.iacr.org/2009/259` – Paper wurde zurückgezogen, Aufwandsabschätzung ev. nicht korrekt. Nächstbestes Resultat: 2^{63} Operationen.

[3] RFC4270: „Attacks on Cryptographic Hashes in Internet Protocols“

Salted Hash

Das bloße Hashen eines Passworts und Speichern des Hashes hat folgende Nachteile:

- gleiche Passworte (Vorname der Freundin, „geheim“, „Passwort“, „123456“, ...) liefern denselben Hash (und das kann auffallen oder z.b. mit Google oder anderen online-Hashcrackern entschlüsselt werden).
- Wörterbuchattacken (Hashes aller Wörter[4] der dt. Sprache, ...) plausibel.

Lösung durch Beifügung eines „Salt“:
Statt MD5(Passwort) speichere MD5(Salt Passwort) oder z.B. MD5(Salt MD5(Passwort)).

[4] Duden: 125.000 Einträge

Salted Hash

Das Salt ist ein Zufallswert (dieser muss z.B. in der Datenbank oder `passwd/shadow`-Datei gespeichert werden) oder z.B. der Username.
Vorteile:

- Salt | Passwort erhöht die Komplexität des zu hashenden Begriffs (und vermindert die Wahrscheinlichkeit des Erfolgs einer Wörterbuchattacke.
- Wenn ein Urbild (Preimage) des Hashwerts tatsächlich gefunden wurde, nützt es nur, wenn es mit dem Salt anfängt (unwahrscheinlich) – der User kann nur den 2. Teil (Passwort) eingeben.

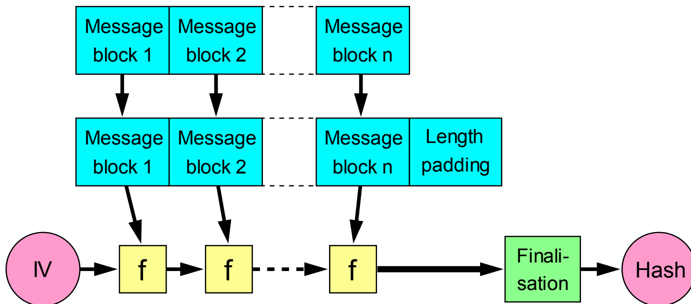
Übersicht über gängige Hashfunktionen

Name	Hashgrösse	Jahr
MD2	128	1989
MD4	128	1990
MD5	128	1992
MD6[5]	variabel < 512	2008
SHA(0)	160	1992 - durch SHA1 ersetzt
SHA1	160	1995
SHA224 (SHA2)	224	2004
SHA256 (SHA2)	256	2001
SHA384 (SHA2)	384	2001
SHA512 (SHA2)	512	2001
SHA3	?	Wettbewerb läuft

[5] eingereicht, aber zurückgezogen für SHA3

Merkle-Damgård Hashkonstruktion

Übliches Konstruktionsverfahren bekannter Hashverfahren (z.B.: MD4, MD5, SHA1, SHA2



1. Message Padding (vielfaches von 512 Byte)
2. Sequentielles Verarbeiten der Datenblöcke
3. (ev.) Endverarbeitung, Ausgabe

Message Padding

(nahezu) identisch bei MD4, MD5, SHA1, SHA2

Message muss ein Vielfaches von 512 Byte lang sein.

- Anhängen eines 1-Bits
- Auffüllen mit 0-Bits, bis letzter Block = 448 Bits lang
- Anhängen der Messagelänge $|M|$ (64 Bit lang)[6]
- Wird **immer** gemacht (auch wenn $|M| = n * 512$ Bits)

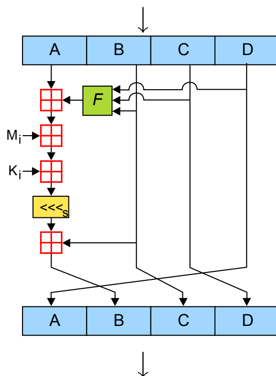
Warum kann man nicht einfach 0-en anhängen? Wieso muss das immer gemacht werden?

---

[6] Bei MD4/MD5 nur die 64 niedrigsten Bits falls $|M| >= 2^{64}$ - bei SHA1/SHA2 Längenbeschränkung auf $|M| < 2^{64}$

Message Digest 5 (MD5, RFC1321)[7]

Initialisierung / Zwischenhash (128 Bit)



nächster Zwischenhashwert / Ausgabe (128 Bit)

[7] Graphik aus Wikipedia

Message Digest 5 (MD5, RFC1321)

- Variableninitialisierung A, B, C, D fixe Konstanten
- Funktionen F folgendermassen definiert:

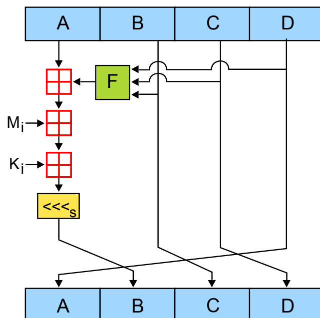
$$\begin{aligned} F_1(X,Y,Z) &= (X \wedge Y) \vee (\neg X \wedge Z) \\ F_2(X,Y,Z) &= (X \wedge Z) \vee (Y \wedge \neg Z) \\ F_3(X,Y,Z) &= X \oplus Y \oplus Z \\ F_4(X,Y,Z) &= Y \oplus (X \vee \neg Z) \end{aligned}$$

- 512-Bit Nachrichtenblock wird in 16 32-Bit Blöcke $w[i]$ geteilt.
  - for i = 0 to 15: $F = F_1$, 16 Runden mit $M[i] = w[i]$
  - for i = 16 to 31: $F = F_2$, 16 Runden mit $M[i] = w[(5i+1)mod16]$
  - for i = 32 to 47: $F = F_3$, 16 Runden mit $M[i] = w[(3i+5)mod16]$
  - for i = 48 to 63: $F = F_4$, 16 Runden mit $M[i] = w[(7i)mod16]$
- K_i vordefinierte Rundenkonstanten, Shiftanzahl S rundenabhängig.

Message Digest 5 (MD5, RFC1321)

- 64 Runden
- 128 Bit Hashwert
- Kollisionen inzwischen einfach errechenbar $\longrightarrow$ gebrochen!
- noch häufig verwendet.

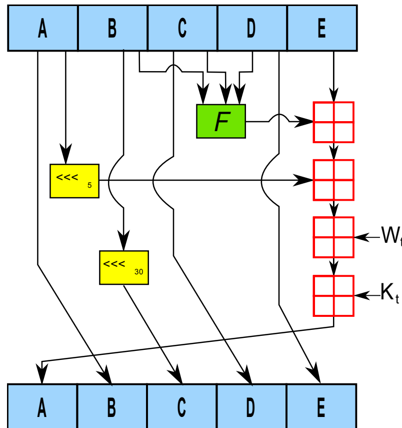
Message Digest 4 (MD4, RFC1320)[8]



MD4 besteht aus 48 Runden (3 verschiedene Gruppen zu je 16 Operationen)

[8] Graphik aus Wikipedia

Secure Hash algorithm 1 (SHA1, RFC3174)[9]



[9] Graphik aus Wikipedia

Secure Hash algorithm 1 (SHA1, RFC3174)

- längerer Hash $\longrightarrow$ 5 (32 Bit) interne Variablen = 160 Bit
- Designed von der NSA, approved vom National Institute of Standards and Technology (NIST) (ebenso SHA-0 (zurückgezogen, minimale Modifikation zu SHA-1) und SHA-2)
- SHA-1 besteht aus 80(!) Runden dieser Operationen.
- Standardhashfunktion für viele Anwendungen in der Informatik (siehe Einleitung)
- `http://csrc.nist.gov/publications/fips/fips180-3/fips180-3_final.pdf`

Secure Hash algorithm 1 (SHA1, RFC3174)

Probleme

- Kollisionen sind mit weniger Aufwand berechenbar, als sie sein sollten (2^{69} (ev. 2^{52}(?)) statt 2^{80} Operationen).
- ist (noch) nicht kritisch (Aufwand nach wie vor zu groß), aber der Algorithmus hat Schwächen.
- Kollisionen in Varianten mit reduzierter Rundenanzahl schon gefunden.

Secure Hash algorithm 1 (SHA1, RFC3174)

Probleme - kritisch?

- Schwächen von SHA1 sind (noch) nicht so kritisch, dass Kollisionen sofort errechenbar sind.
- Für Passworte ist das Problem bei Verwendung eines Salt auch noch kein Problem.
- zum Fälschen von (schon signierten) Verträgen etc. müsste ein second preimage Angriff ausgeführt werden – noch wesentlich schwieriger.
- Also alles in Butter?

Secure Hash algorithm 1 (SHA1, RFC3174)

Aber:

- Diese gefundenen Schwächen (Kollisionen in 2^{69}, 2^{63}, 2^{52}(?) (anstatt 2^{80}) sind nur eine Obergrenze. Es können durchaus noch größere Schwächen gefunden werden.
- SHA2 ist vermutlich besser (längere Hashes), aber auch weniger analysiert (und ähnlich aufgebaut wie SHA1).
- DAHER: Entwicklung von SHA3:

Secure Hash Algorithm 3

Internationaler Wettbewerb - Cryptographic Hash Algorithm Competition

NIST has opened a public competition to develop a new cryptographic hash algorithm, which converts a variable length message into a short “message digest“ that can be used for digital signatures, message authentication and other applications. The competition is NIST’s response to recent advances in the cryptanalysis of hash functions. The new hash algorithm will be called “SHA-3“ and will augment the hash algorithms currently specified in FIPS 180-2, Secure Hash Standard. Entries for the competition must be received by October 31, 2008. The competition is announced in the Federal Register Notice published on November 2, 2007; (`http://www.nist.gov/hash-competition`)

Secure Hash Algorithm 3 Competition

http://ehash.iaik.tugraz.at/wiki/The_SHA-3_Zoo

- 191 Einreichungen (http://ehash.iaik.tugraz.at/wiki/SHA-3_submitters)
- 51 Einreichungen in Runde 1
- 14 Einreichungen in Runde 2
- Schaun wir, wer den Wettbewerb gewinnt — und wie lange SHA-3 halten wird, bevor er geknackt wird und wir SHA-4 suchen?

HMAC: Keyed-Hashing for Message Authentication (RFC2104)

Verwendung von geheimen Schlüsseln um Messages (N) sicher zu authentifizieren.

$$HMAC_K(N) = H\left((K \oplus opad) \mid\mid H\left((K \oplus ipad) \mid\mid N\right)\right)$$

- $H()$ Hashfunktion
- opad / ipad Konstanten
- $\oplus$ XOR
- $\mid\mid$ Verkettung

Wirkt für einen bekannten Schlüssel wie eine normale Hashfunktion, kann ansonsten nur verifiziert werden, wenn der Schlüssel bekannt ist.

Implementierung in Hardware (Beispiel)

VIA C7-M Processor – includiert einen Security Coprozessor: Padlock (aus dem Werbetext:)

1. Secure Hash Algorithm: SHA-1 and SHA-256 – Throughput at rates of up to 5 gigabits per second.
2. AES Encryption: ECB, CBC, CFB, and OFB modes – Another method of encrypting information at rates of up to 25 gigabits per second.
3. Montgomery Multiplier: An invaluable tool to assist the encryption of information using the RSA Public key algorithm.
4. NX Execute Protection: When enabled, this feature prevents most worms from proliferating on your device.
5. Random Number Generator: Two random number generators can create unpredictable random numbers at a rate of 1600K to 20M per second.

Supported ab Linux 2.6.18, OpenSSL 0.9.8

Gefahr von Kollisionen

Zufällig gefundene Kollision zufälliger Bitfolgen sind ein Problem? Ja!

Eine einmal gefundene Kollision (eines 512 Bit Blocks) kann (vorn und hinten) um weitere Datenblöcke erweitert werden:

$$M_0, M_1, \dots, M_i, \dots, M_n$$
$$M_0, M_1, \dots, N_i, \dots, M_n$$

Wenn der (Zwischen)hashwert nach den (unterschiedlichen!) Datenblöcken $M_i \neq N_i$ gleich ist, ändert er sich auch bei nachfolgenden (gleichen) Datenblöcken nicht mehr!
Und?

Gefahr von Kollisionen II

Und das kann ausgenutzt werden:

Gutes oder böses Programm?

```
int main()
{
    char * gutoderboese = "[sinnloserstring1]";
    /* oder [sinnloserstring2] mit gleicher MD5-Summe */
    if (strcmp(gutoderboese,"[sinnloserstring1]")) {
        gutes_programm()
    } else {
        boeses_programm()
    };
}
```

Gefahr von Kollisionen III

- Wenn der String korrekt positioniert und korrekt gewählt ist, haben beide Programme (sowohl mit `"[sinnloserstring1]"` als auch `"[sinnloserstring2]"`) denselben Hashwert – und könnten unentdeckt(!) ausgetauscht werden.
- Abhängig vom Wert des Strings macht das Programm aber was anderes.
- Do-it-yourself: „evilize“-library:
  `http://www.mathstat.dal.ca/~selinger/md5collision/`
  $\Rightarrow$ Hashfunktionen mit errechenbaren Kollisionen nicht verwenden!
- Auch möglich z.B. bei Postscript-Dokumenten: Dokument (Hash des Dokuments) wird signiert und kann nachträglich ausgetauscht werden.

Zusammenfassung

Hashfunktionen sind in vielen Bereichen der modernen Soft- und Hardwaretechnik unersetzlich

- Ständig neue Angriffsmethoden und Rechenleistungen verlangen nach widerstandsfähigen Hashfunktionen.
- Die öffentliche Ausschreibung von SHA-3 lässt auf eine gegen viele (jetzt bekannten und hoffentlich auch zukünftigen) Angriffe immunen Hashfunktion hoffen (ähnlich AES).
- **Attacks always get better, they never get worse!**

Fragen? Feedback?

Vielen Dank für Ihre Aufmerksamkeit

Wolfgang Dautermann
wolfgang.dautermann [AT] fh-joanneum.at