Cryptographic hash function

A hash function is a function that takes an arbitrary block of data and returns a fixed-size unique bit string representation.

Related terms: hash, message digest, fingerprint, checksum
Cryptographic hash function

Properties:

- Easy to compute hash value (fast)
- Hard to restore message from hash (one-way)
- Hard to find messages with the same hash (collision resistant)
- Similar messages have a very different hashes (avalanche effect)

Attacks:

- Collision attack:
  \[ \text{hash}(x) = \text{hash}(y) \mid \text{find any } x \text{ and } y \text{ such that } x \neq y \]
- (First) Preimage attack:
  \[ \text{hash}(x) = h \mid \text{given } h, \text{ find any } x \]
- Second preimage attack:
  \[ \text{hash}(x) = \text{hash}(y) \mid \text{given } x \text{ and } \text{hash}(x), \text{ find } y \neq x \]
“In cryptography, a **brute-force attack**, or exhaustive key search [..] might be used when it is not possible to take advantage of other weaknesses in an encryption system. It consists of systematically checking all possible keys or passwords until the correct one is found.”

https://en.wikipedia.org/wiki/Brute-force_attack

E.g.: If a cryptosystem which has 56-bit key can be brute-forced using $2^{56}$ operations\(^1\) then the cryptosystem has security level of 56 bits.

- $2^{128}$ operations infeasible
- $2^{80}$ become feasible

Note, $2^{81}$ operations take twice the time of $2^{80}$ ($2^{128}$ vs $2^{256}$).

---

1\(^{\text{The term “operation” is not defined.}}\)
Cryptographic hash functions

- **MD5** – 128-bit output
  - collision attack in $2^{18}$ (brute-force $2^{64}$)
  - Chosen-prefix collision attack

- **preimage and second preimage attack in** $2^{73}$ (brute-force $2^{128}$)

- **SHA-1** – 160-bit output
  - theoretical practical collision attack in $2^{63.1}$ (brute-force $2^{80}$)

- **SHA-256** – 256-bit output

- **SHA-512** – 512-bit output

- **SHA-3** – 224/256/384/512-bit output
Data identification and integrity verification

• Integrity and authenticity of distributed files

![Image of winMd5Sum software](image)

• Disk imaging in digital forensics
• Distribution of MS Windows updates
• Remote file comparison (rsync)
Commitment scheme

Here is the proof of my clairvoyant powers – next U.S. president will be SHA256(x)=dd2a4a379c04b944834c66cc797815f0...

• Binding – due to collision resistance of SHA256
• Hiding – due to one-wayness of SHA256

```python
>>> for candidate in ['Joe Biden', 'Hillary Clinton', 'Donald Trump']:
...     print hashlib.sha256(candidate).hexdigest(), candidate
...     
...     d99d0b129d5864e1813438a885034452... Joe Biden
dd2a4a379c04b944834c66cc797815f0... Hillary Clinton
e4f2e1f0e2ae4d3ce7018cf3b4f3577c... Donald Trump
```

• Improve hiding property by adding randomness

```python
>>> prediction = 'Hillary Clinton|' + os.urandom(16)
>>> hashlib.sha256(prediction).hexdigest()
'c20634bfe2b34c0958e9e96ec8ea40c0...' 
>>> prediction
'Hillary Clinton|p\xe0\xce\xcbZaYE\xb7-f\x99\x9f\xa5%\xf5'
```

• Function must be secure against chosen-prefix collision attack
Coin flipping over phone

How to generate random number from 0 to 99 over phone:

1. Alice: “my commitment value (SHA256(x)) is 108c995b953c8a35561103e2014cf828…”

2. Bob: “my value is 84”

3. Alice: “my value was 65“

4. Bob checks if
   SHA256(“65”) = “108c995b953c8a35561103e2014cf828…”

Random number generated:

   65 + 84 = 149 mod 100 = 49
Server-side password storage

**Law & Disorder / Civilization & Discontents**

Sony hacked yet again, plaintext passwords, e-mails, DOB posted

The hackers of Lulz Security have broken into yet more Sony websites, this ...

by Peter Bright - June 3 2011, 4:06am EEST

---

Hello, Iam Idahc a Lebanese Hacker

I was Bored and I play the game of the year: "hacker vs Sony"

I hacked little database of 120 user with an sql injection......

site: http://apps.pro.sony.eu/

username password mobile office email website

<table>
<thead>
<tr>
<th>User</th>
<th>Password</th>
<th>Mobile</th>
<th>Office</th>
<th>Email</th>
<th>Website</th>
</tr>
</thead>
<tbody>
<tr>
<td>ascul</td>
<td>+44 7818 92</td>
<td>dam@3d</td>
<td></td>
<td></td>
<td>csie.unav</td>
</tr>
<tr>
<td>-JafarAbd ourfar</td>
<td>145899</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sterChamp manter</td>
<td>1 1522</td>
<td></td>
<td></td>
<td></td>
<td>masteringenlo</td>
</tr>
<tr>
<td>arth</td>
<td>+44 7887 83</td>
<td>44 788</td>
<td></td>
<td></td>
<td>techniche.com</td>
</tr>
<tr>
<td>o alro</td>
<td>+44 075927</td>
<td>44 (6)</td>
<td></td>
<td></td>
<td>the.com</td>
</tr>
<tr>
<td>bern</td>
<td>+34 630 163</td>
<td>91 63</td>
<td></td>
<td></td>
<td>yahoo.es</td>
</tr>
<tr>
<td>marie c668763c07r</td>
<td>465cf</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>the Arghfc</td>
<td>+3069</td>
<td></td>
<td></td>
<td></td>
<td><a href="http://www.argie">www.argie</a>.</td>
</tr>
<tr>
<td>van vanbar</td>
<td>+32 41</td>
<td>5 +32</td>
<td></td>
<td></td>
<td>elde@scarlet.</td>
</tr>
<tr>
<td>loc locbon</td>
<td>+44 (6)</td>
<td>66545</td>
<td></td>
<td></td>
<td>lisli@virgin</td>
</tr>
<tr>
<td>mal malbru</td>
<td>+33 6</td>
<td>8 +33</td>
<td></td>
<td></td>
<td>bit.com</td>
</tr>
<tr>
<td>oJohansen senato</td>
<td>91355</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Solution – store password hashes in database
  - Compare received plaintext password with hash from db
Server-side password storage

```
mysql> SELECT name, password FROM users;
+--------+------------------------------------------+
| name   | password                               |
+--------+------------------------------------------+
| Jeff   | b1b3773a05c0ed0176787a4f1574ff0075f7521e |
| Katrin | 2730f2c29354932611d328cfff0c9f01e10328ec |
| Mike   | e72e941812b920c908bba17798d5e27eb627912 |
+--------+------------------------------------------+
```

- Solution – add user-specific salt to the password
  
  ```python
db_salt = os.urandom(8).encode('hex')
db_password = hashlib.sha1(password + db_salt).hexdigest()
```
Server-side password storage

mysql> SELECT name, password, salt FROM users;
+--------+------------------------------------------+------------------+
<table>
<thead>
<tr>
<th>name</th>
<th>password</th>
<th>salt</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jeff</td>
<td>0771580376c18f7faeae9de565ff663eeff8c5cc</td>
<td>7d3a5cd7fc28aa9</td>
</tr>
<tr>
<td>Katrin</td>
<td>9c70ccbf02e5b8be46ebedi149326d5d375895187</td>
<td>df9372246bcdf8d0</td>
</tr>
<tr>
<td>Mike</td>
<td>622cd81265db68c3b2616400f312c2a7096f5848</td>
<td>9a737642e2bf40db8</td>
</tr>
</tbody>
</table>
+--------+------------------------------------------+------------------+

• No benefit in building password specific lookup table
• Even the same passwords will have different hash
Server-side password storage

- Brute-force cracking still possible:

  ```python
  >>> hashlib.sha1('qwerty'+'7d3a5ccd7fc28aa9').hexdigest()
  '0771580376c18f7faeeae9de565ff663eef8c5cc'
  ```

<table>
<thead>
<tr>
<th>Hash Function</th>
<th>GPU Speed</th>
</tr>
</thead>
<tbody>
<tr>
<td>MD5</td>
<td>23070.7 M/s</td>
</tr>
<tr>
<td>SHA-1</td>
<td>7973.8 M/s</td>
</tr>
<tr>
<td>SHA-256</td>
<td>3110.2 M/s</td>
</tr>
<tr>
<td>SHA-512</td>
<td>267.1 M/s</td>
</tr>
<tr>
<td>NTLM</td>
<td>44035.3 M/s</td>
</tr>
<tr>
<td>DES</td>
<td>185.1 M/s</td>
</tr>
<tr>
<td>WPA/WPA2</td>
<td>348.0 k/s</td>
</tr>
</tbody>
</table>

Table: GPU speed

- Slow down brute-force by arbitrary factor using iterated hash: $h(h(h(h(x))))$
  - Trade-off: performance vs cost of brute-forcing
  - Goal is to increase asymmetry
Server-side password storage

Task: Calculate the security level of password hashes stored in a company X database. Company’s X password security policy requires password to be exactly 7 lowercase letters.

- Number of letters in English alphabet: 26
- Hash operations required:
  - to brute-force 1-letter password: 26
  - to brute-force 2-letter password: $26 \times 26$
  - to brute-force 3-letter password: $26 \times 26 \times 26$
  - to brute-force 7-letter password: $26^7$

\[
26^7 = 8031810176 \approx 2^{32}
\]

If using 2000 iterations:

\[
2^{32} \times 2000 = 2^{32} \times 2^{11} \approx 2^{43}
\]

Password storage has 43-bit security level.\(^2\)

\(^2\)Assuming user chooses letters randomly.
**Hash-based PRNG**

| sha1(seed)                  | sha1(seed + "0")                  |
| sha1(sha1(seed))            | sha1(seed + "1")                  |
| sha1(sha1(sha1(seed)))      | sha1(seed + "2")                  |
| sha1(sha1(sha1(sha1(seed))))| sha1(seed + "3")                  |
| ...                        | ...                                 |

```python
import hashlib
def hash_prng(seed):
    i = 0
    while True:
        print hashlib.sha1(seed + str(i)).hexdigest()
        i += 1

hash_prng('fookey')
```

- Standardized construction – Hash-DRBG (NIST SP 800-90)
- If we have PRNG we can build stream cipher
Hash chain

- Linking-based time-stamping
- Also known as blockchain
Hash tree (Merkle tree)

- Easy to prove that a node belongs to the tree
- To prove that “Data block 3” is part of the tree $h_{1-4}$:
  - “Data block 3”
  - “$h_4$”
  - “$h_{1-2}$”
HMAC: hash-based Message Authentication Code

- Valid MAC cannot be produced without knowing the key
- Naive implementation $\text{hash}(\text{key} + \text{message})$ is vulnerable!
  - Safe to use HMAC construction (RFC 2104)
- MAC does not guarantee freshness of the message
- Can MAC be used as a digital signature?
Questions

- What are the properties of hash function?
- What attacks cryptographic hash function must resist?
- What does the size of hash function output influences?
- What is a “security level” in cryptography?
- What is commitment scheme useful for?
- Why it is better to store hashed passwords in db?
- How can we increase the security level of password hashing?
- How can we create encryption scheme from hash function?
- What is HMAC useful for?
- Why using MD5/SHA1 for HMAC is not insecure?
Task: HMAC – 3p

Implement a tool that calculates and verifies integrity of a file.

$ python hmac.py
Usage:
-verify <filename>
-mac <filename>

$ python hmac.py -mac somefile
[?] Enter key: secretkey
[+] Calculated HMAC-SHA256: f5e94378cae5a3d0836e145f28807bb7076d28cd22b2481d45f92a904be9d2e8
[+] Writing HMAC DigestInfo to somefile.hmac

$ python hmac.py -verify somefile
[+] Reading HMAC DigestInfo from somefile.hmac
[+] HMAC-SHA256 digest: f5e94378cae5a3d0836e145f28807bb7076d28cd22b2481d45f92a904be9d2e8
[?] Enter key: secretkey
[+] Calculated HMAC-SHA256: f5e94378cae5a3d0836e145f28807bb7076d28cd22b2481d45f92a904be9d2e8
[+] HMAC verification successful!

$ dumpasn1 somefile.hmac
 0 49: SEQUENCE {
 2 13:  SEQUENCE {
 4 9:  OBJECT IDENTIFIER sha-256 (2 16 840 1 101 3 4 2 1)
 15 0:  NULL
 :  }
 17 32:  OCTET STRING
 :  F5 E9 43 78 CA E5 A3 D0 83 6E 14 5F 28 80 7B B7
 :  07 6D 28 CD 22 B2 48 1D 45 F9 2A 90 4B E9 D2 E8
 :  }

DigestInfo

DigestInfo ::= SEQUENCE {
    digestAlgorithm AlgorithmIdentifier,
    digest OCTET STRING
}

AlgorithmIdentifier ::= SEQUENCE {
    algorithm OBJECT IDENTIFIER,
    parameters ANY DEFINED BY algorithm OPTIONAL
}

$ dumpasn1 hashobject
0 33: SEQUENCE {
  2 9: SEQUENCE {
    4 5: OBJECT IDENTIFIER sha1 (1 3 14 3 2 26)
    11 0: NULL
  }
  13 20: OCTET STRING DA 39 A3 EE 5E 6B 4B 0D 32 55 BF ...
}

- Standard structure to store algorithm and calculated digest
- Defined in PKCS#1 v1.5 signature creation (RFC 2313)
- Our hash functions have no parameters (ASN.1 NULL)
Task: HMAC

- Use python's hashlib and hmac library
e.g., hmac.new('somekey', None, hashlib.md5)
- Must support hashing of huge files
  - Read file by 512 byte chunks
  - Feed chunks sequentially to hash.update()
  - Finally execute hash.digest()
- HMAC digest must be written to “.hmac” file using DigestInfo
  ASN.1 structure
  - Use your own ASN.1 encoder
    - Please embed your encoder in your solution
  - For decoding use pyasn1
- MAC'er must use HMAC-SHA256
- Verifier must support HMAC-MD5, HMAC-SHA1 and
  HMAC-SHA256 (algorithm must be read from DigestInfo)
  - OID for MD5: 1.2.840.113549.2.5
  - OID for SHA1: 1.3.14.3.2.26
  - OID for SHA256: 2.16.840.1.101.3.4.2.1
Task: Test Cases

$ echo -e -n "\x01" > file_sha256
$ python hmac.py -mac file_sha256
[?] Enter key: testkey
[+] Calculated HMAC-SHA256: a8be648dd48738b964391a00d4522fe988d10e3d5b2dbf8629a3dcbc0ce93ffd
[+] Writing HMAC DigestInfo to file_sha256.hmac
$ python hmac.py -verify file_sha256
[+] Reading HMAC DigestInfo from file_sha256.hmac
[+] HMAC-SHA256 digest: a8be648dd48738b964391a00d4522fe988d10e3d5b2dbf8629a3dcbc0ce93ffd
[?] Enter key: testkey
[+] Calculated HMAC-SHA256: a8be648dd48738b964391a00d4522fe988d10e3d5b2dbf8629a3dcbc0ce93ffd
[+] HMAC verification successful!

$ wget https://bitbucket.org/appcrypto/2019/raw/master/03/hmac_testcases.tgz
$ tar -zxvf hmac_testcases.tgz

$ python hmac.py -verify file_md5
[+] Reading HMAC DigestInfo from file_md5.hmac
[+] HMAC-MD5 digest: 9e8031ab9d85a5fa0753344bc8c31a2f
[?] Enter key: secretkey
[+] Calculated HMAC-MD5: 9e8031ab9d85a5fa0753344bc8c31a2f
[+] HMAC verification successful!

$ python hmac.py -verify file_sha1
[+] Reading HMAC DigestInfo from file_sha1.hmac
[+] HMAC-SHA1 digest: ebfb4fc1a84d5f9fcbd1b7c8d5d625ac9f5b4c81
[?] Enter key: secretkey
[+] Calculated HMAC-SHA1: ebfb4fc1a84d5f9fcbd1b7c8d5d625ac9f5b4c81
[+] HMAC verification successful!

$ python hmac.py -verify file_sha256
[+] Reading HMAC DigestInfo from file_sha256.hmac
[+] HMAC-SHA256 digest: c40932474350a3f29a9f800e68b6429c64b7526800f8701ae9b4e73db8a3b700
[?] Enter key: secretkey
[+] Calculated HMAC-SHA256: 737f438db779461e6163aa236797099f08b154de6f5741843a549866ae57a5fd
[-] Wrong key or message has been manipulated!
pyasn1 library – decoding DER

$ sudo apt install python-pyasn1
$ python
>>> from pyasn1.codec.der import decoder
>>> der = open('asn1.der').read()
>>> decoder.decode(der)
(<Sequence value object at 0x7f2a0cc3cf10 componentType=<NamedTypes
   object at 0x7f2a0cc128d0 types > tagSet=<TagSet object at
   0x7f2a0cc3cfd0 tags 0:32:16-128:32:0>
   subtypeSpec=<ConstraintsIntersection object at 0x7f2a0cc12850>
   sizeSpec=<ConstraintsIntersection object at 0x7f2a0cc12890>
   payload [<Set value object at 0x7f2a0cc3c890 componentType=[...
   <UTCTime value object at 0x7f2a0cc3ca50 tagSet <TagSet object at
   0x7f2a0cc3c590 tags 0:0:23> encoding us-ascii payload
   [150223010900Z]>, '', '])
>>> decoder.decode(der)[0][0][2]
<Integer value object at 0x7f2a0cc3ced0 tagSet <TagSet object at
   0x7f2a0cc3ca90 tags 0:0:2-128:32:11> payload [65407]>
>>> int(decoder.decode(der)[0][0][2])
65407
>>> decoder.decode(der)[0][0][2].__class__.__name__
'Integer'

• Can’t handle large (>3MB) DER encoded structures
• Can’t handle DER structures with implicit tagging