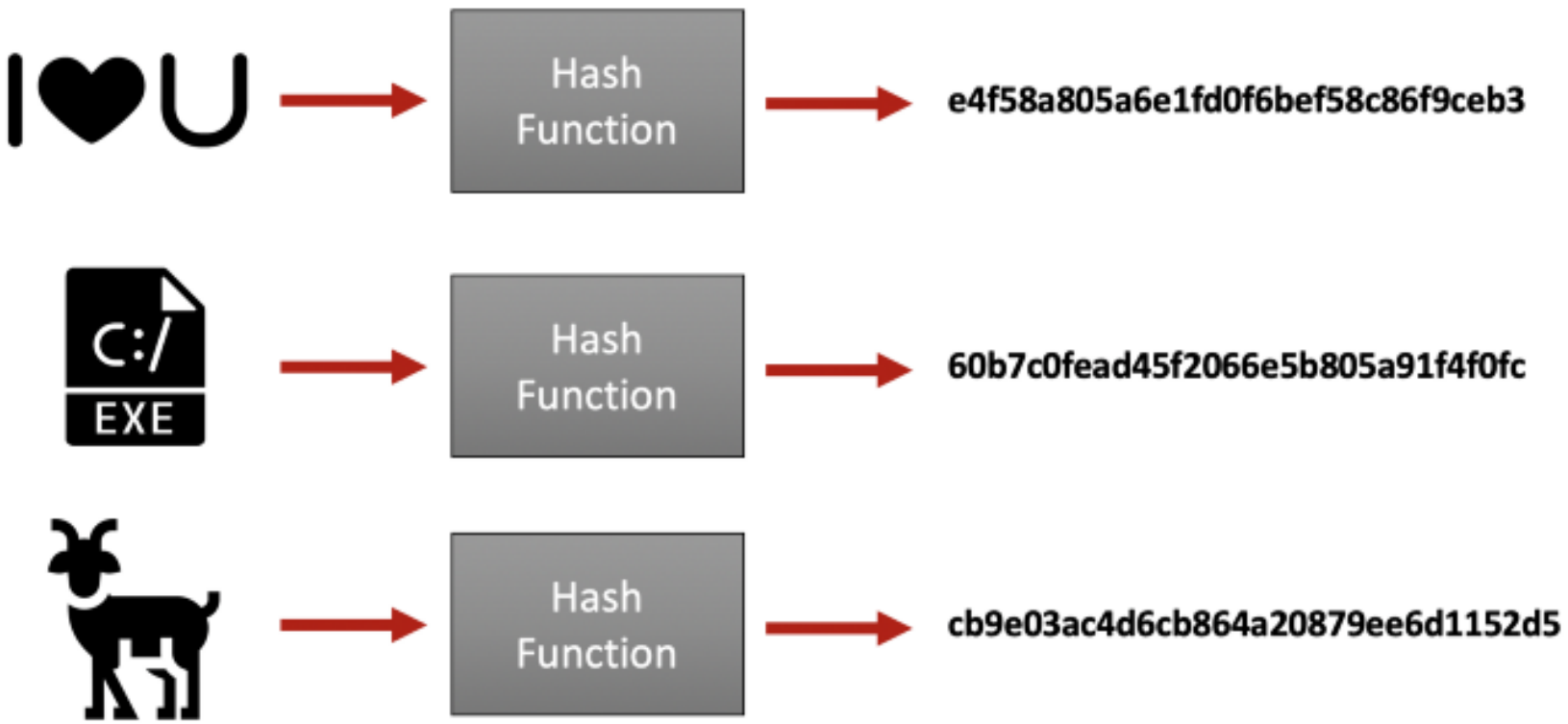
CS4236 Cryptography Theory and Practice Topic 7 - Hashes

Hugh Anderson

National University of Singapore
School of Computing

September, 2022

Hashes...


Hash Function
e4f58a805a6e1fd0f6bef58c86f9ceb3
C:/
EXE
Hash Function
60b7c0fead45f2066e5b805a91f4f0fc
Hash Function
cb9e03ac4d6cb864a20879ee6d1152d5

Outline

The story so far ... where are we?

The last 6 weeks: weekly steps we have taken

1. We had a historical/contextual introduction in Session 1.
2. We progressed from the traditional view of perfect secrecy, through to a game/experiment view: *perfect* indistinguishability.
3. *Perfectly* indistinguishable was relaxed to give *computationally* indistinguishable. An EAV-Secure system was constructed with a PRG.
4. The notion of CPA-secure was developed, where the adversary had access to an encryption oracle. CPA-secure was achieved with a PRF.
5. Modes were described, and CCA-Security was outlined. The “padding oracle” example was described, motivating CCA.
6. The argument was made that integrity could, and should, be separated from confidentiality - the right tool for the right job. We saw MAC; (cryptographic) message authentication codes which were unlikely (i.e. less than negl) to be able to be forged: *Secure-MAC* and *Strong-MAC*. A fixed-size MAC based on a PRF, Construction 4.5, was shown and extended to variable length messages (Constructions 4.7 and 4.11). The MACs were then used for *authenticated* encryption, which when combined gave CCA security and unforgeability.

Hash function? Say what? hash $\neq$ encrypt

Maps data to a value, the reverse may be difficult...

```
hugh@comp:~[508]$ md5sum ss.c
550114bc3cc3359e55ba33abe8983a85  ss.c
hugh@comp:~[511]$ md5sum TXT/cybercom.txt
9ec4c12949a4f31474f299058ce2b22a  TXT/cybercom.txt
```

MD5 and SHA

MD5...

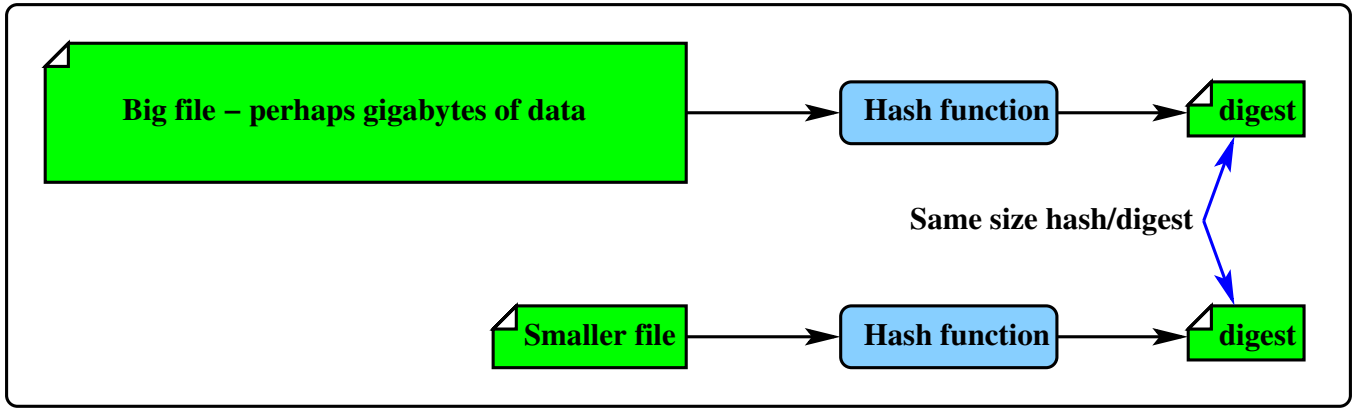
Was designed in 1991 by Ron Rivest (of RSA fame), as a cryptographic hash function. However it has been found to **not** be *collision-resistant* (defined later), and so, despite extensive use in industry today, it should no longer be used as a cryptographic hash.

The SHA family...

Originally designed by NIST in 1993 (SHA-0) but revised in 1995 as SHA-1. Revisions in 2002 led to SHA-2 256/384/512: higher security levels, and with no known weaknesses. SHA-3 was released in 2012 - it is the newest one, with some different design elements.

	SHA-1	SHA-224	SHA-256	SHA-384	SHA-512
Digest (hash) size (bits)	160	224	256	384	512
Message size	$< 2^{64}$	$< 2^{64}$	$< 2^{64}$	$< 2^{128}$	$< 2^{128}$
Block size (bits)	512	512	512	1024	1024
Number of steps	80	64	64	80	80

From checksums to cryptographic hash functions



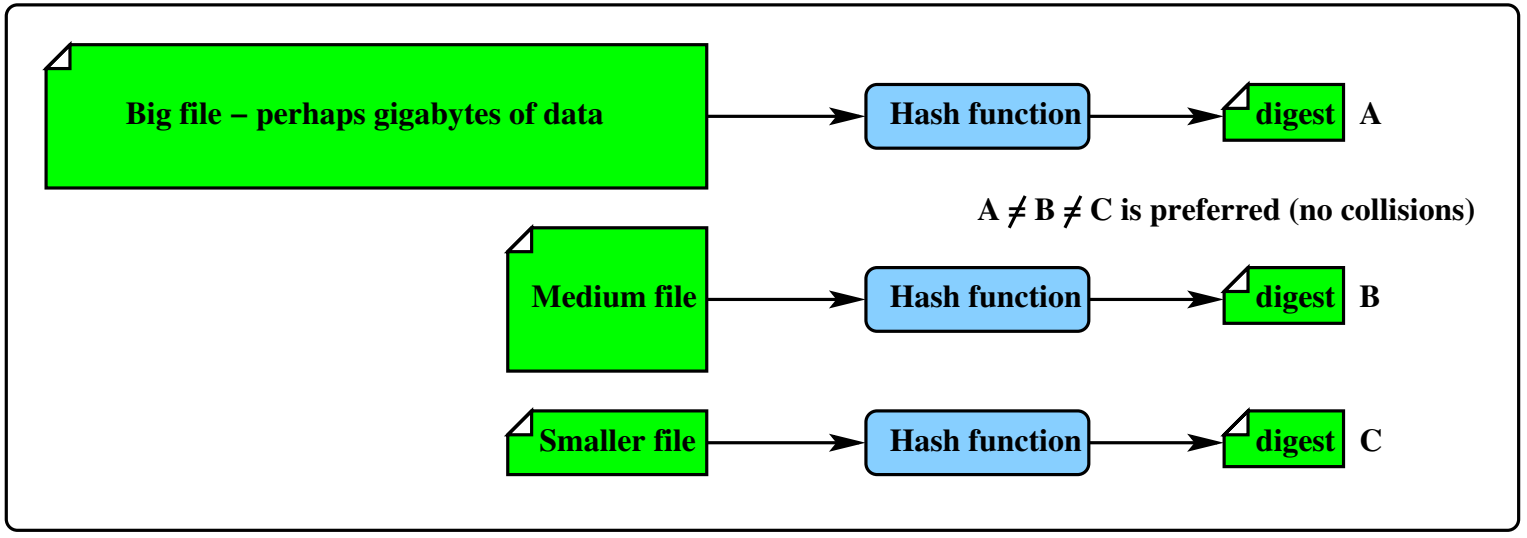
Protecting data (I and A)...

Non-secure ones are commonly called checksums, and they provide protection against noise (as opposed to a malevolent attacker). Examples of (non-cryptographic) checksums are parity bits, and CRCs (Cyclic Redundancy Checksums).

A hash function maps a long message, or a large amount of data, to a (shorter) check message of some sort. We attach the hashed value to the original message, as a check.
Cryptographic functions may be used to construct hash functions.

Here we look at *efficient* functions that are *cryptographically secure*.

Cryptographic hashes



A *hash* is not just a checksum

A cryptographic hash meets extra security requirements. In particular, we want them to be collision resistant (defined on next slide).

A cryptographic hash is an important crypto primitive. It is used in digital signatures : hash-and-sign, protecting password files (salted hashes), Commitment schemes, Proof-of-Work in Bitcoin, and so on.

Properties of cryptographic hash functions

Desirable properties: It should be efficient, public, and...

1. It should be computationally infeasible to find two items x, x' where $x \neq x'$, but both map to $y = \mathcal{H}(x) = \mathcal{H}(x')$ (collision-resistant).
2. Given x, it should be computationally infeasible to find an $x' \neq x$ such that $\mathcal{H}(x') = \mathcal{H}(x)$ (target-collision, or second pre-image resistant).
3. Given y, it should be computationally infeasible to find item x mapping to it: $y = \mathcal{H}(x)$ (one-way property, or pre-image resistant

Note that (1) implies (2), and also a (2) attack implies a (1) attack.

When two messages map to same hash...

1. Can there be no collisions at all? If the number of messages $\#m$ is greater than the number of hashes $\#\mathcal{H}(\ldots)$, then consider the pigeonhole principle - if there are n roosts for $n+1$ pigeons...
2. How likely are collisions? Consider the birthday paradox[a]. What is the probability that at least two of N randomly selected people have the same birthday? It is much more likely than you would suspect...

[a] BTW - It is not really a paradox, just counter-intuitive.

Formal formulation of hash functions

Formulation so we can have hash constructions

The theoretical formulation of *collision resistant* turns out to be problematic. The textbook takes the approach of using a keyed hash function (even though in practice, standard hash functions have no key). The notation for a keyed hash function is

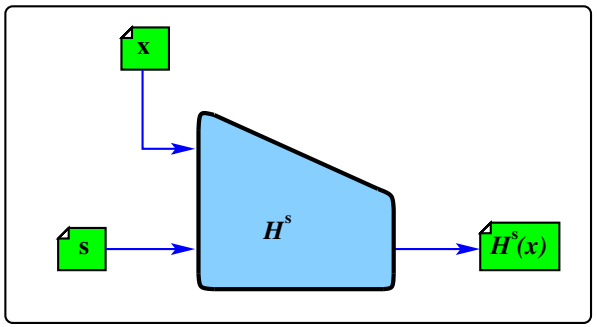
$$\mathcal{H}^s(x)$$

The keyed-hash construction solves a technical issue, described at the bottom of p155 in the textbook. The core of the issue is that we know that there do exist collisions in (say) SHA, and so it is difficult to claim that a probability of a collision is less than negligable. We could imagine an adversary that has a hard-coded collision, and can provide a pair of values to the challenger in an experiment that hash to the same value. However, if we use a keyed hash, then the entropy of the key can uphold the validity of the negligability claim.

In practice, it is common to use unkeyed hashes like SHA-3.

We also will see some form of “pseudo randomness” used in our hash-based constructions.

The definition of hash functions:



Definition 5.1 - Hash function $(\mathrm{Gen}, \mathcal{H})$

Gen: Take in the size info 1^n, output a key s.

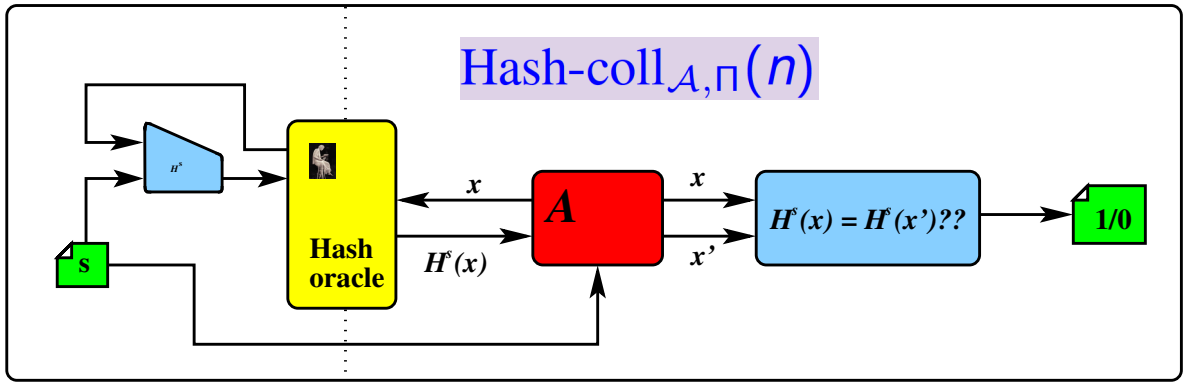
$\mathcal{H}$**:** Input is the key s and an arbitrary long string x in $\{0,1\}^*$. Output is written as $\mathcal{H}^s(x)$, which is a string of length $\ell(n)$.

Fixed length input: compression functions

Some hashes only take in an input of a certain fixed size. In general, the length of the input must be larger than the output for it to be useful, as the goal of the hash is to "compress" the input.

If $\mathcal{H}^s$ is defined only for inputs $x \in \{0,1\}^{\ell'(n)}$ and $\ell'(n) > \ell(n)$, then we say that $(\mathrm{Gen}, \mathcal{H})$ is a fixed-length hash function for inputs of length ℓ', and write it h^s. This is commonly termed a *compression* function.

Collision resistance game: $\text{Hash-coll}_{\mathcal{A},\Pi}$



Game $\text{Hash-coll}_{\mathcal{A},\Pi}$ (Collision-finding)

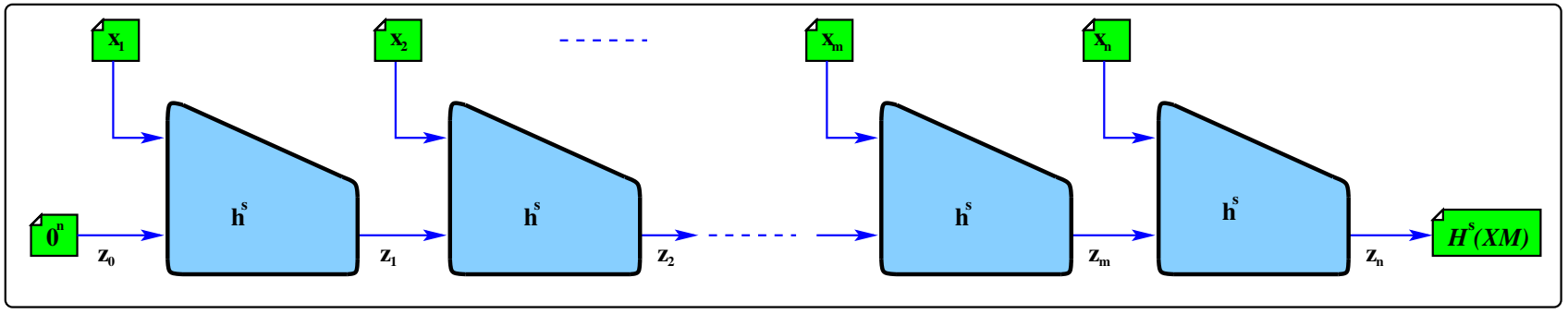
1. Generate key $s \leftarrow \text{Gen}(1^n)$.
2. PPT adversary $\mathcal{A}$, on input s, eventually outputs x, x'.
3. Adversary wins (output is 1) iff $x \neq x'$ and $\mathcal{H}^s(x) = \mathcal{H}^s(x')$.

Definition 5.2 collision resistance

We say that a hash function $(\text{Gen}, \mathcal{H})$ is collision resistant iff for any PPT adversary $\mathcal{A}$, there is a negl s.t.

$$\Pr[\text{Hash-coll}_{\mathcal{A},\Pi}(n) = 1] \leq \text{negl}(n)$$

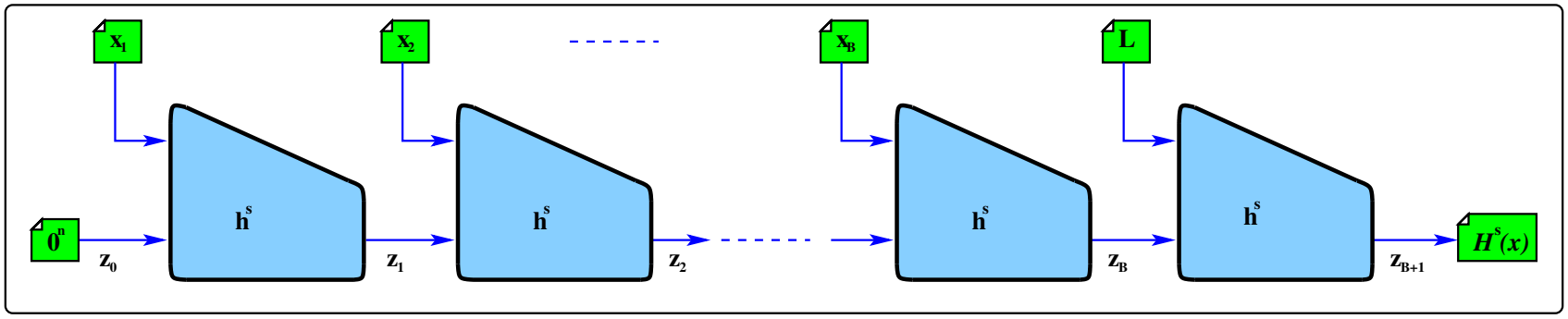
Issues with constructing hash functions



Length extension issues

We saw in CBC (Session 5) how extension attacks could be used in encryption, where a fixed size encryption engine was incorrectly extended to variable length messages. A similar issue appears in using compression functions to build hashes for variable length messages. The length extension problem exists because $h^s(m)$ provides direct information about the intermediate state after the first blocks of m.

Construction for a hash function



Construction 5.3 (p157 - Merkle Damgård)

This construction extends a fixed size hash ($h^s()$ - a *compression* function) to an arbitrary long message.

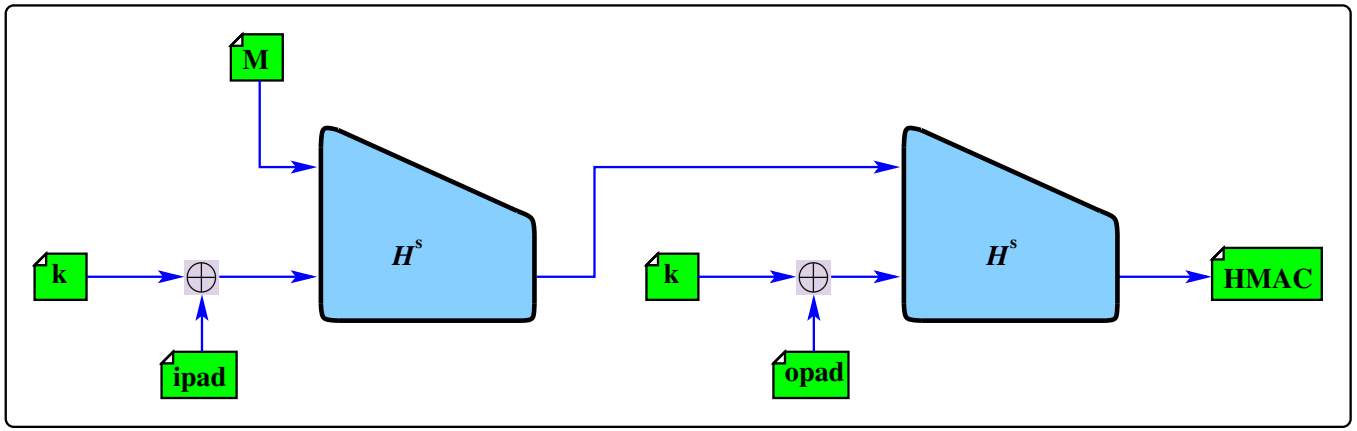
Let (Gen, h) be a fixed-length hash function for inputs of length $2n$ and with output length n. Construct the hash function $(\mathrm{Gen}, \mathcal{H})$ as follows:

Gen: remains unchanged.

$\mathcal{H}^s(x)$**:** on input s and string $x \in \{0,1\}^*$ of length $L < 2^n$:

- Set $B := \lceil \frac{L}{n} \rceil$, the number of blocks in x, and pad x with zeros.The result is $x_1, \ldots, x_B$. Set $x_{B+1} := L$.
- Set $z_0 := 0^n$ - the IV.
- For $i = 1, ..., B+1$: $z_i := h^s(z_{i-1} + x_i)$. Output z_{B+1}.

As used on the Internet...



The HMAC construction

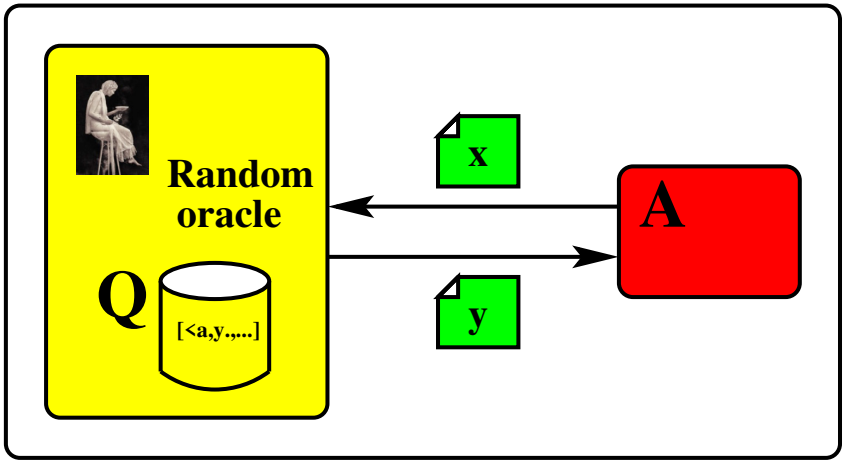
See also RFC2104, and construction 5.7 (p161) - the HMAC, that constructs a Mac using a hash:

$$\mathrm{HMAC}(k, m) = \mathcal{H}^s((k \oplus \mathrm{opad}) \dashv\!\vdash \mathcal{H}^s((k \oplus \mathrm{ipad}) \dashv\!\vdash m)$$

The ipad and opad values are discussed in the discussion on p162. The section on HMAC proves that such a construction is secure.

A practical hash like SHA-3 is unkeyed. So, theoretically it cannot be "collision resistant" under our definition.
If we use a salt as key, it could be possible (and believed to be so). This does not mean that it is not secure if we use the unkeyed hash.

The random oracle



An idealised model, for use in proofs

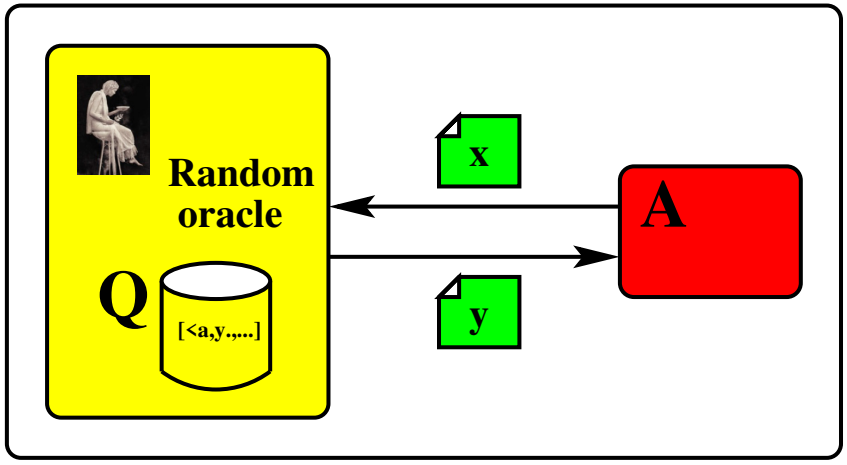
In the definition of the PRF before, we saw similar oracles. The "random oracle" is an alternative way, a methodology, for formulating hash ideas. It makes proofs (sometimes) a little simpler (see page 174). However, a random oracle is not a realizable, actual thing!

It is an idealized model, and so any proofs based on an assumption that (say) the underlying hash is a random oracle are theoretically meaningless. We really should not be saying

...xxx is secure, assuming $\mathcal{H}$ is a random oracle.

Statements that are made without assuming a random oracle are called "*Standard model*". (e.g. xxx is secure under standard model).

The random oracle rules

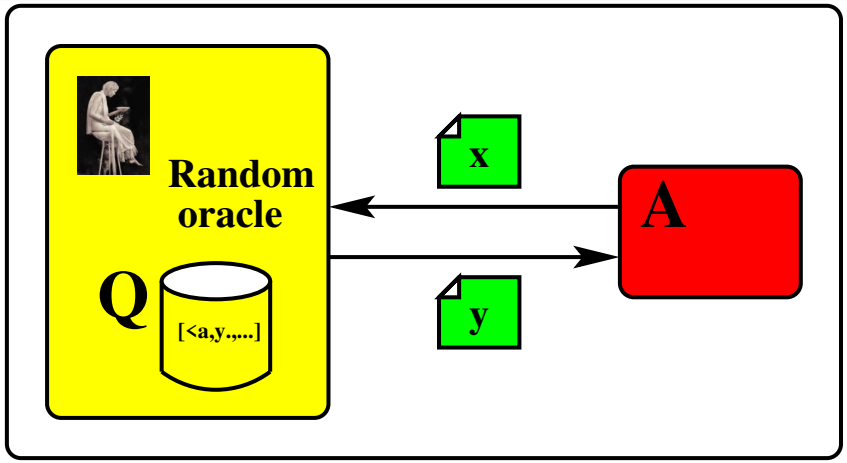


Definition of the behaviour of the random oracle

A random oracle is an oracle that keeps record of queries it had received. The oracle maintains a set Q containing tuples $\langle x, y \rangle$, The first term x is the index, and Q is initially empty. On each query, the oracle either returns a previously queried value (i.e. it is *consistent*), or it returns a new uniform y as follows:

```
Q = dict()
def O(x):
    if x in Q:
        return Q[x]
    else:
        Q[x] = random.getrandbits(...)
        return Q[x]
```

Summary



An idealised model. Better than nothing?

The main purpose of formulating the "*Random Oracle*" is not to prove that a hash is secure.

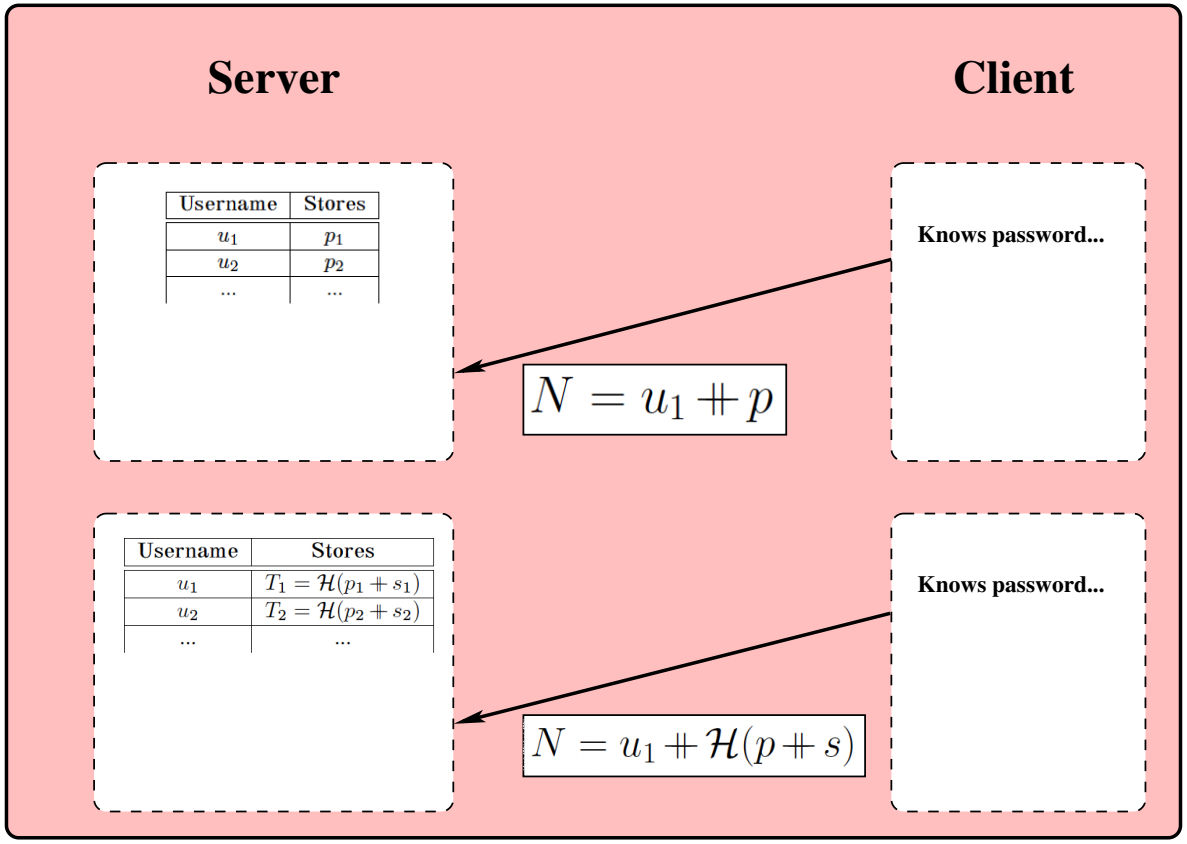
> *We do not make claims such as a function is secure if it is indistinguishable from the random oracle.*

Instead, the Random Oracle is used in proving the security of constructions based on hashes. We might claim that...

> *...construction XYZ is secure if $\mathcal{H}$ is a random oracle.*

In practice, when we implement the construction, we replace the random oracle with a cryptographic hash function, under the hope that it is as good as a random oracle. It can be used in a reduction proof, we can "simulate" it and so on. The textbook goes into detail at pgs 178,179.

Passwords and hashing

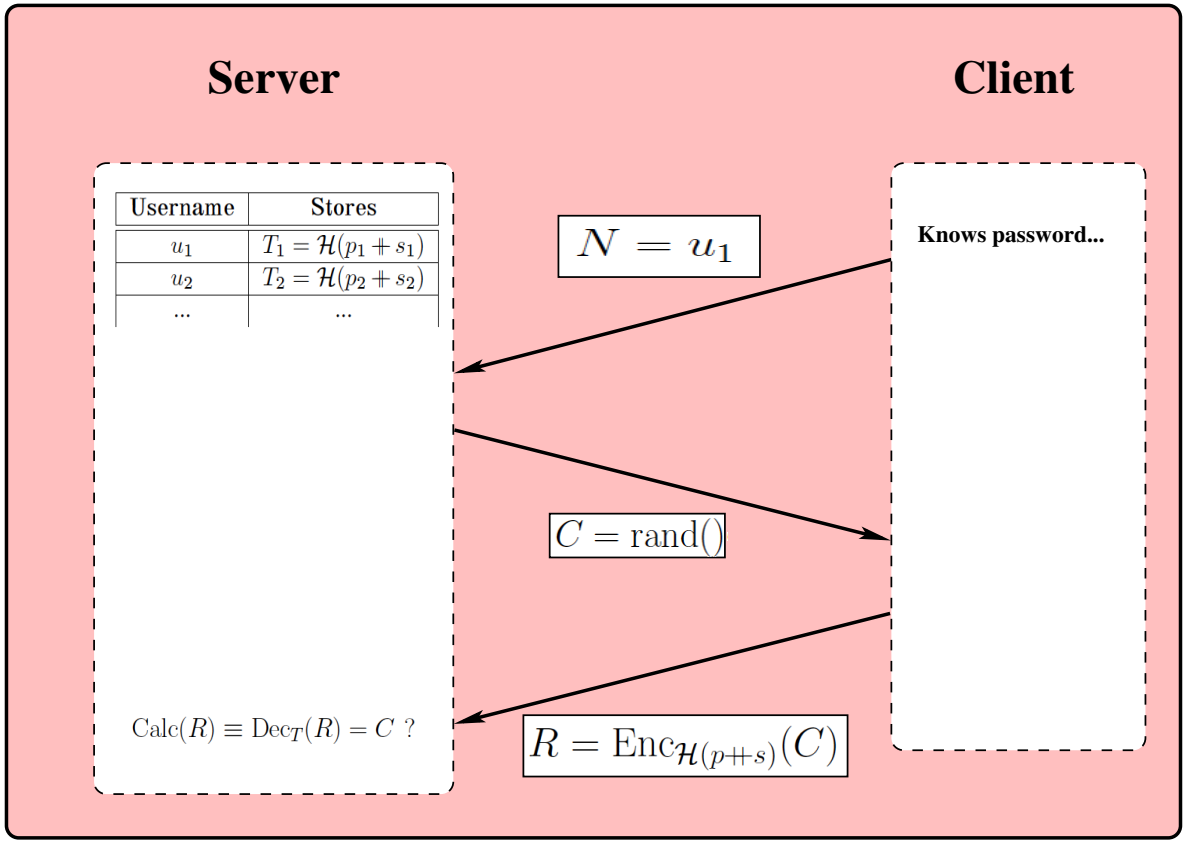


Encryption $\neq$ Hashing... Basic authentication is not much

Assume $\mathcal{H}$ is a cryptographic hash function, p is the password, s is a salt for the password, and $p + s$ as before.

Note that here, a hash is as good as the password.

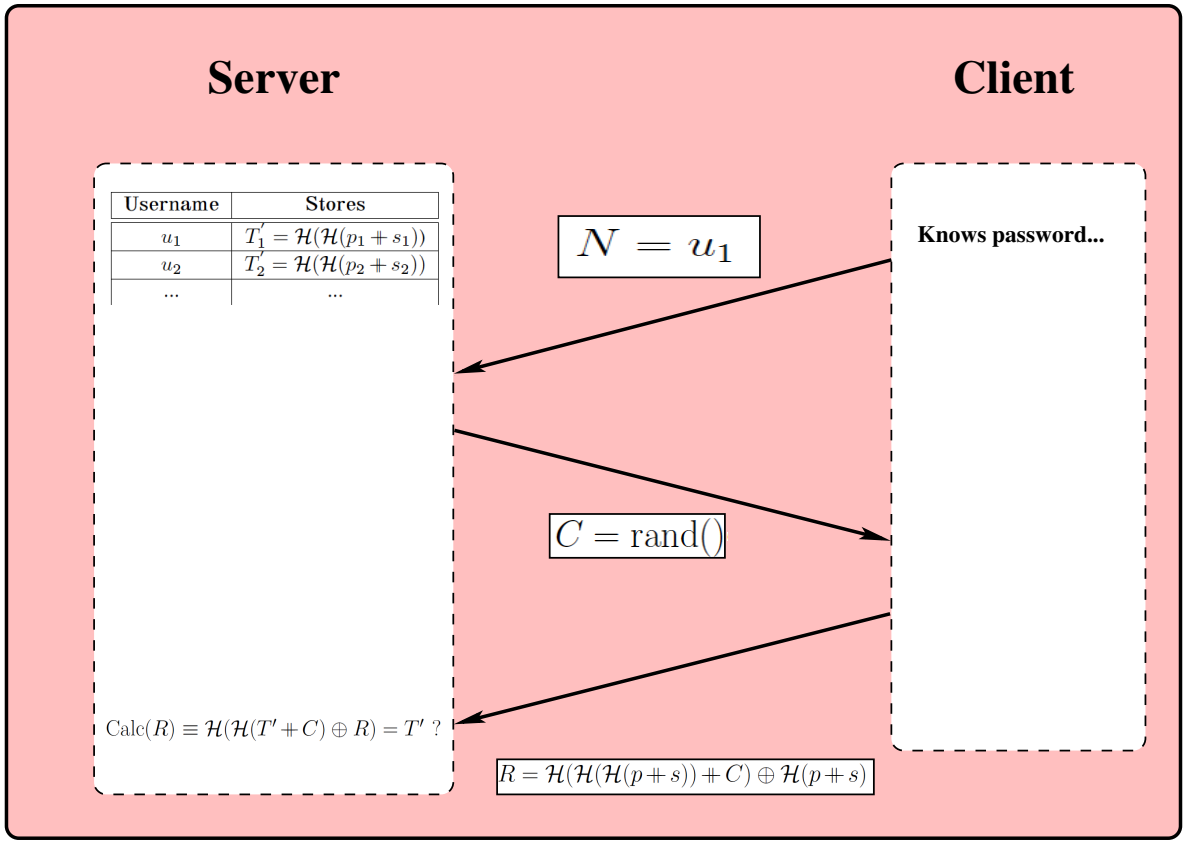
Challenge-response



Encryption $\neq$ Hashing

Note also that here, a hash is (still) as good as the password.

Enhanced challenge-response (just fun)



Encryption $\neq$ Hashing

Note that here, a hash is not as good as the password.

But, it still has issues in the face of an eavesdropper.

Enhanced challenge-response (just fun)

To stop hash being as-good-as a password

The client knows p, s, and when given challenge C, calculates response R:

$$R = \mathcal{H}(\mathcal{H}(\mathcal{H}(p \mathbin{+\!\!+} s)) \mathbin{+\!\!+} C) \oplus \mathcal{H}(p \mathbin{+\!\!+} s)$$

The server does not know p or $\mathcal{H}(p \mathbin{+\!\!+} s)$, but instead has $T' = \mathcal{H}(\mathcal{H}(p \mathbin{+\!\!+} s))$. The server calculates

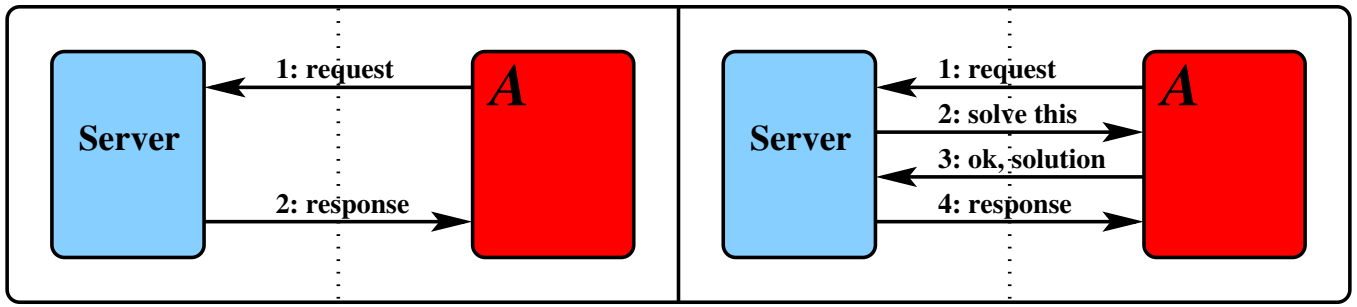
$$\mathrm{Calc}(R) = \mathcal{H}(\mathcal{H}(T' \mathbin{+\!\!+} C) \oplus R)$$

If $\mathrm{Calc}(R) = T'$ then the server agrees that the client knows p, s or (at least) $\mathcal{H}(p \mathbin{+\!\!+} s)$. Expanding out the server's calculation, we see how it works:

$$\begin{aligned} \mathrm{Calc}(R) &= \mathcal{H}(\mathcal{H}(T' \mathbin{+\!\!+} C) \oplus R) \\ &= \mathcal{H}(\mathcal{H}(\mathcal{H}(\mathcal{H}(p \mathbin{+\!\!+} s)) \mathbin{+\!\!+} C) \oplus \mathcal{H}(\mathcal{H}(\mathcal{H}(p \mathbin{+\!\!+} s)) \mathbin{+\!\!+} C) \oplus \mathcal{H}(p \mathbin{+\!\!+} s)) \\ &= \mathcal{H}(\mathcal{H}(p \mathbin{+\!\!+} s)) \\ &= T' \end{aligned}$$

https://openwall.info/wiki/people/solar/algorithms/challenge-response-authentication

Proof-of-work based prevention of DOS



NDSS 1999

In a typical Denial of Service attack, the attacker invests relatively less resource in issuing a request, but the victim has to spend much more resources in answering the request. The puzzle based method tries to flip this asymmetry. An attacker needs to submit a proof-of-work before the server serves the request.

The puzzles should be easy to construct, easy to verify, but difficult to solve. One way to construct a puzzle might be to choose a random string s, and force the attacker to find a string r s.t. the first (or last) 20 bits of the digest $\mathcal{H}(r + s)$ are all zeros.
The attacker is forced to carried out an expected number of 2^{20} hash operations (?) to find such a choice of r.
`https://www.ndss-symposium.org/wp-content/uploads/2017/09/A-Cryptographic-Defense-Against-Connection-Depletion-Attacks-Ari-Juels.pdf`

Proof of work using (python) crypto libraries

Find (at least) *n* leading zeroes using PyCrypto libraries

```
from Crypto.Hash import SHA256
from Crypto.Cipher import AES
import struct,binascii,os,sys

s,dist,avg = os.urandom(16),0,0.0
leadingzeros = int(sys.argv[1])
while True:
   dist = dist+1
   r = os.urandom(16)
   sample = r + s
   print("          Trial:  ", binascii.hexlify(sample).upper())
   tst = bytearray(SHA256.new(sample).digest())
   int_val = int.from_bytes(tst[:4], "big")
   intbit = format(int_val, "032b")
   ni = intbit.index('1')
   if ni >= leadingzeros:
      if avg == 0.0:
         avg = float(dist)
      avg = ((63.0*avg)+float(dist))/64.0
      print("          Result:  ", binascii.hexlify(tst).upper())
      print(ni, format(avg, '7.1f'), format(dist, '7d'), " ", intbit)
      dist = 0
   else:
      print("          Result:  ", binascii.hexlify(tst).upper())
```

Proof of work using (python) crypto libraries

Find n leading zeroes test run: $\text{Result} := \mathcal{H}(r \# s)$

Display the number of leading zeroes, the current (smoothed) average, the number of trial searches since the last one found, and the first 32 bits of the bitstring found with at least n leading zeroes.

```
hugh@comp Session7Programs % python3 hash1.py 2
      Trial:   b'BD9E884F68BFDE112F6CE951C65332F3BC9B211FFDD7B29AAA66463DFDDCE304'
      Result:  b'88FDE5C709CDBD357189B5A8F09538B333030ABFD99C02A8B22F1663960FDD02'
      Trial:   b'0CA48712B62AF3218364092BB153DD1ABC9B211FFDD7B29AAA66463DFDDCE304'
      Result:  b'1E36D4B3D1A2573489F487DC45ACBC1B2346B9D9B72739E225B543ECDAEEA519'
3   2.0   2      00011110001101101101010010110011
      Trial:   b'A45096A540DE15853563E23E8BABEA75BC9B211FFDD7B29AAA66463DFDDCE304'
      Result:  b'941C51357BAD89B0884F24CC5CB0E6E0ADF31CDBFF4490C3145F7A4D5B9A3D16'
      Trial:   b'2142AF12BC8717BBB8A02B679F9C6AA3BC9B211FFDD7B29AAA66463DFDDCE304'
      Result:  b'AB181EC61B7D5A74DD0227C4DE20DBA20589692B64E8E03A4C686D80FEF4D856'
      Trial:   b'860B43CD98BC34BBB5D52EC1FE64B5EABC9B211FFDD7B29AAA66463DFDDCE304'
      Result:  b'CAA7115C78C2D71B7264F4AEA76D9EFB9366D84C48AEDFDCB02BD0A320B126A9'
      Trial:   b'4EF7DAC59FB8BD31A823919A3B446E98BC9B211FFDD7B29AAA66463DFDDCE304'
      Result:  b'31EBA90801C8F4CDBB6D971CA2766DDB3D35CAB197493DE3225A15FF5E906A77'
2   2.0   4      00110001111010111010100100001000
```

Cryptographically Generated Addresses: CGA

IPv6 address format

bits	64	64
field	*subnet prefix*	*interface identifier*

Cryptographically generated addresses in IPv6

The interface identifier is either automatically generated from the interface's MAC address, obtained from a DHCPv6 server, automatically established randomly, or assigned manually.

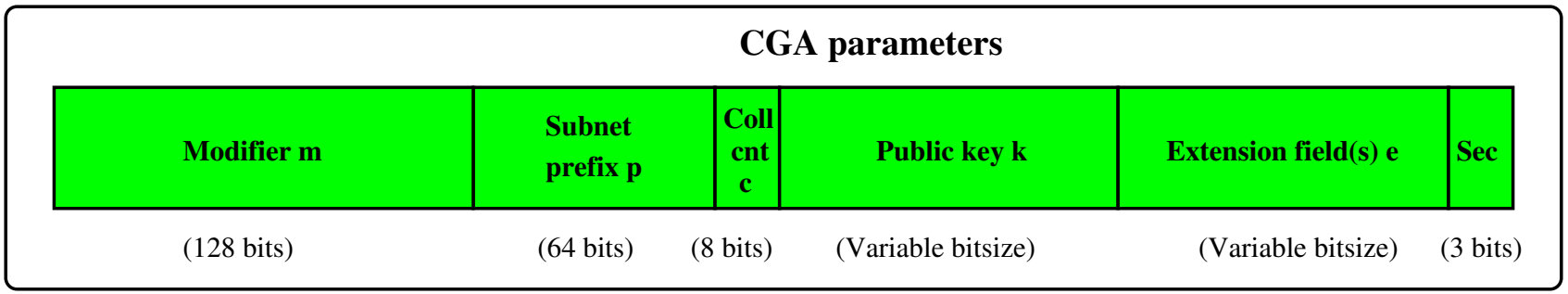
```
hugh@comp Session7Programs % ifconfig anpi1
    anpi1: flags=8863<UP,BROADCAST,SMART,RUNNING,SIMPLEX,MULTICAST>
    ether 1e:00:ea:3c:f3:3a
    inet6 fe80::1c00:eaff:fe3c:f33a
```

`fe80::1c00:eaff:fe3c:f33a` is shorthand for `fe80:0000:0000:0000:1c00:eaff:fe3c:f33a`.

RFC3972 (CGA) binds a public key to the interface identifier, with the idea that In order for an attacker to masquerade as a host with that CGA, the attacker must provide a proof-of-work: in particular, find a hash collision!

For RFC3972/CGA, see `http://www.rfc-base.org/rfc-3972.html`
or (Wikipedia) `https://en.wikipedia.org/wiki/Cryptographically_Generated_Address`

The CGA proof-of-work



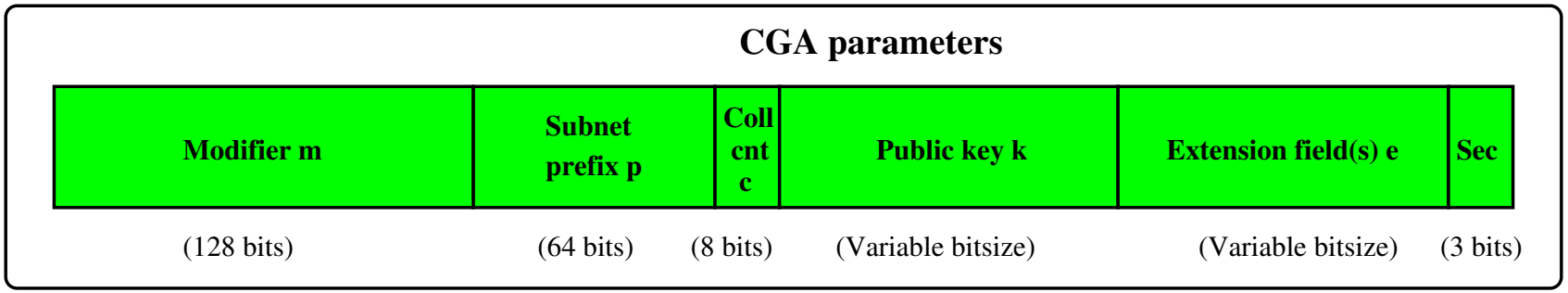
Generation of CGA

Given a public key k, and a subnet prefix p, the collision count c, an optional extension parameter e, then the CGA is determined as follows:

$m := \text{rand}(1^{128})$
do:
$m := m + 1$
$\text{dig}_2 := \mathcal{H}_{\text{SHA1}}(m + 0^{72} + k + e)$
$h_2 := \text{dig}_2[0:14]$
while ... (h_2 leftmost zeroes not 0,16,32, as set by Sec)
$\text{dig}_1 := \mathcal{H}_{\text{SHA1}}(m + p + c + k + e)$
$h_1 := \text{dig}_1[0:8]$ && 0x1c || (Sec << 5)
$\text{CGA} := p + h_1$

There are proposals to implement two different hash algorithms to reduce the likelihood of collision attacks. See RFCs for details if interested.

The CGA verification



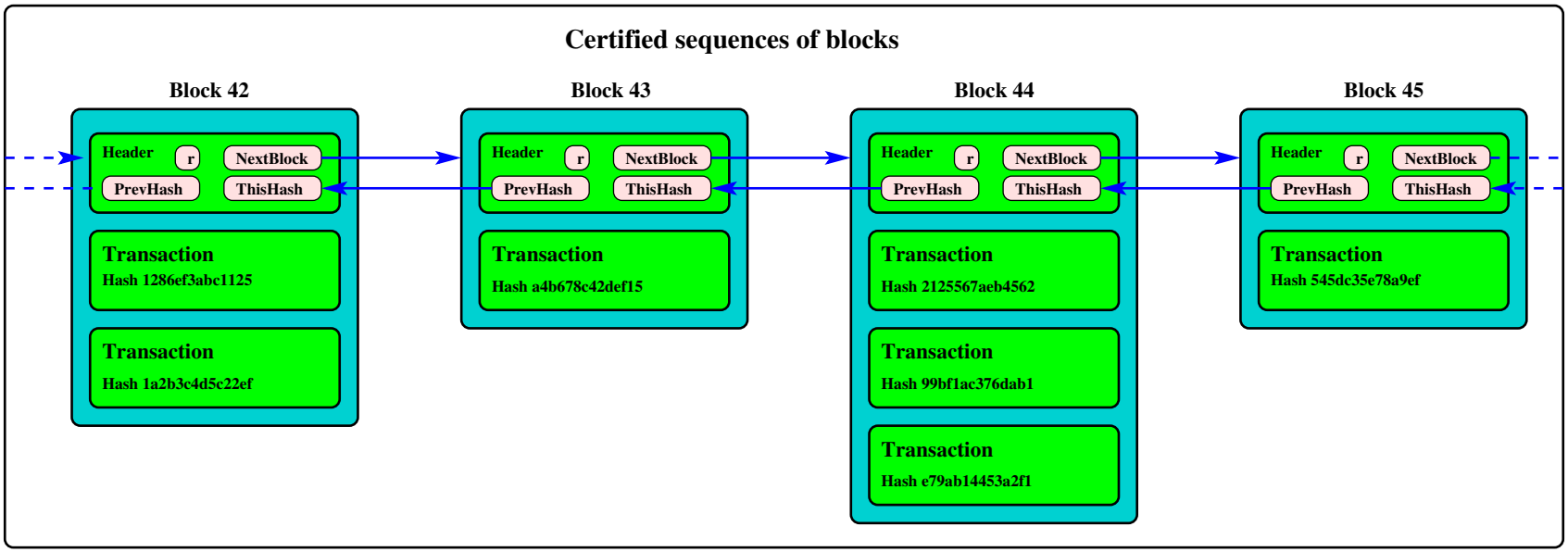
Verification of CGA

A CGA is valid if it is consistent with its parameters. It is verified as follows:

if CollCnt $>$ 2 **OR** CGA$[0:8] \neq p$ **then return false;**
$\text{dig}_1 := \mathcal{H}_{\text{SHA1}}(m \# p \# c \# k \# e)$
$h_1 := \text{dig}_1[0:8]$ && 0x1c)
IntID := CGA$[8:16]$ && 0x1c)
if $h_1 \neq$ IntID **then return false**
Sec := CGA$[8] >> 5$
$\text{dig}_2 := \mathcal{H}_{\text{SHA1}}(m \# 0^{72} \# k \# e)$
$h_2 := \text{dig}_2[0:14]$
if Sec $\neq 0$ **AND** $h_2[0:2 * \text{Sec}] \neq 0$ **then return false**
else return true

After the CGA is verified, the public key k is declared to be valid. Using the public key, one can verify the authenticity of the payload in the packet by verifying the signature (which is generated using the private key).

Proof-of-work in block-chain technology

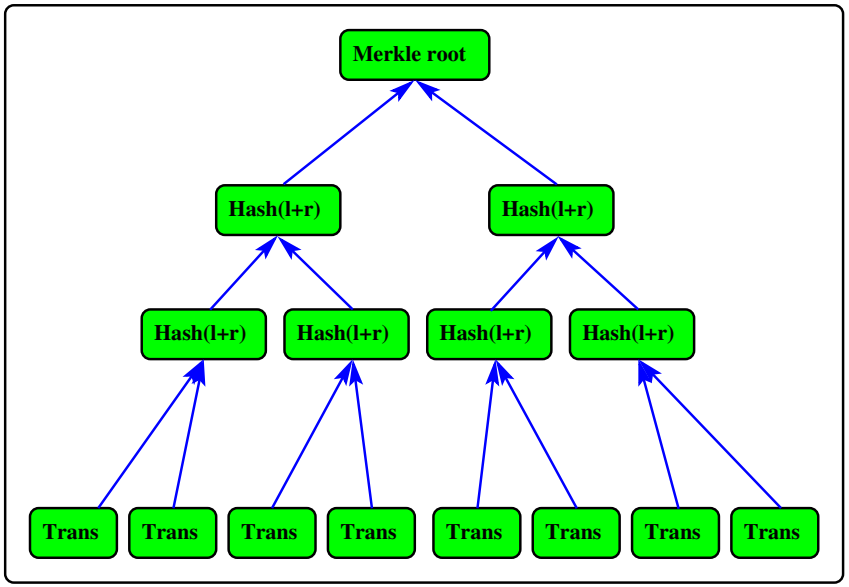


Bitcoin and so on

Bitcoin, Ethereum and many "permissionless" distributed ledgers rely on proof-of-work. Each block has a hash value composed from the preceding blocks and *this* particular block's transactions. Picture a sequence of such accepted/certified sets of transactions.

At some point someone does the work to get a new block accepted - commonly by finding a one-time-value r such that $\mathcal{H}(r + \text{ThisHash})$ has (say) 20 leading zeroes. A blockchain is made from a sequence of blocks of sets of such certified accepted transactions.

Managing a large ledger

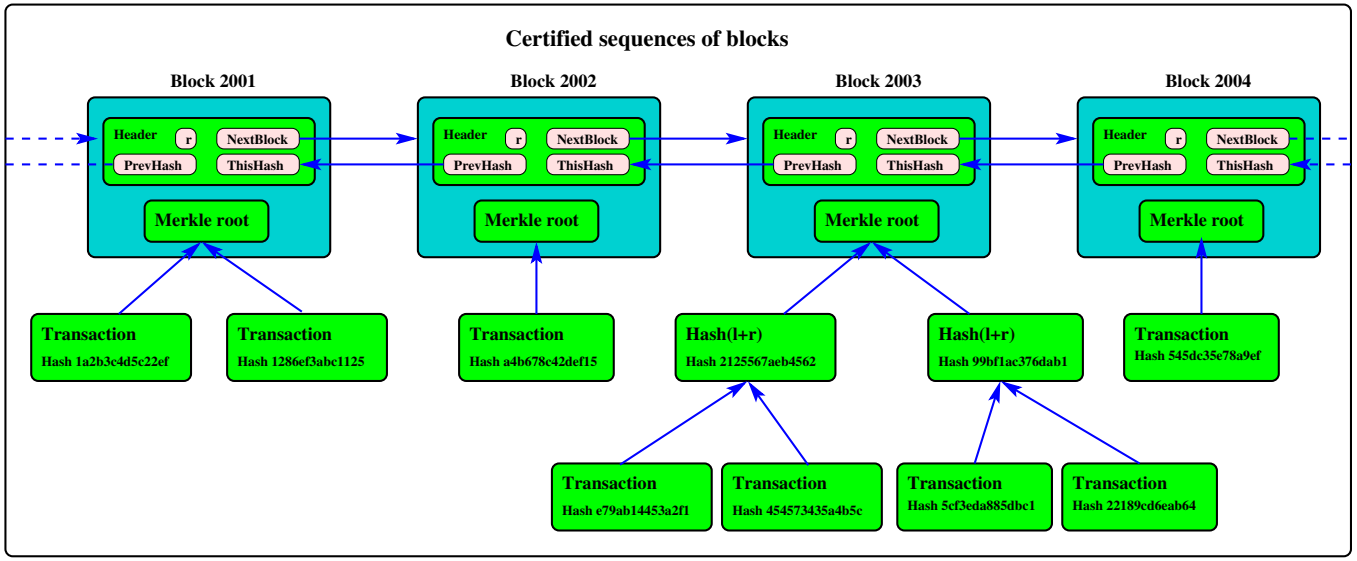


Merkle hash tree

Merkle trees are an efficient example of a cryptographic commitment scheme, in which the root of the Merkle tree is seen as a commitment and leaf nodes may be revealed and proven to be part of the original commitment.

In cryptography and computer science, a hash tree or Merkle tree is a tree in which every leaf node is labelled with the cryptographic hash of a data block, and every non-leaf node is labelled with the cryptographic hash of the labels of its child nodes. Hash trees allow efficient and secure verification of the contents of large data structures.

Merkle hash tree in bitcoin



The hash is accepted as a proof

Although it could be checked if needed.

A comment on puzzles

The examples we have seen are puzzles that consume computation time.

There are puzzles that are designed to consume storage. It is arguably more difficult for an adversary to side-step memory space compared to processing time:

(1) `https://users.soe.ucsc.edu/~abadi/Papers/memory-longer-acm.pdf`
(2) `https://eprint.iacr.org/2014/059.pdf`

Symmetric encryption schemes

$$\mathrm{ENC}_k = (\mathrm{Gen}(1^n), \mathrm{Enc}_k(m), \mathrm{Dec}_k(c)) \qquad \textbf{(ch1)}$$

Properties	Defs	Constructions	Proofs
Perfect secrecy	2.3	(One time pad)	Thm 2.9
↕			Lemma 2.6
Perfect indistinguishability	2.5		
↓			
EAV-secure	3.8	3.17 (PRG)	Thm 3.18
↑			Proof given
EAV-Multi-encryption	3.19		
↑			
CPA-Multi-encryption	3.22		
↕			Thm 3.24
CPA-secure	3.23	3.30 (PRF)	Thm 3.31
↑			
CCA-secure	3.33		
+			
Unforgeable	4.16		
↕			Thm 4.19
Authenticated	4.17	4.18 (Enc-then-Auth)	

MAC and HASH schemes

$$\mathrm{MAC}_k = (\mathrm{Gen}(1^n), \mathrm{Mac}_k(m), \mathrm{Vrfy}_k(m,t)) \quad \textbf{(4.1)}$$

Properties	Defs	Constructions	Proofs
Unforgeable	4.2		
↑			Discussed
Strong	4.3	4.5 (Using PRF)	Thm 4.5
		4.7 (Variable length)	Thm 4.8
		4.11 (CBC-MAC)	Thm 4.12
		4.11(a) (Variable CBC-MAC)	

$$\mathrm{HASH} = (\mathrm{Gen}(1^n), \mathcal{H}^s(x)) \quad \textbf{(5.1)}$$

Properties	Defs	Constructions	Proofs
CollisionResistant	5.2	5.3 (Long message)	