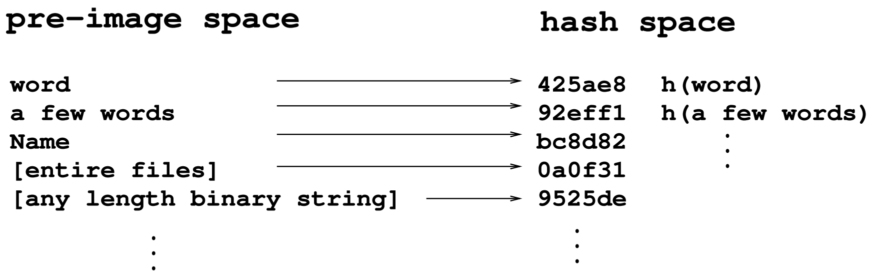
Hash Functions

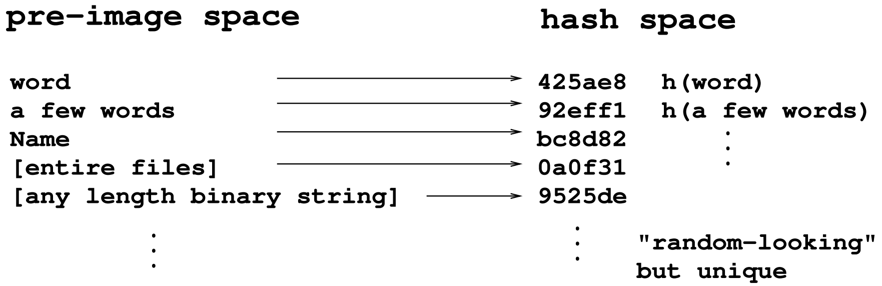
Hash Functions

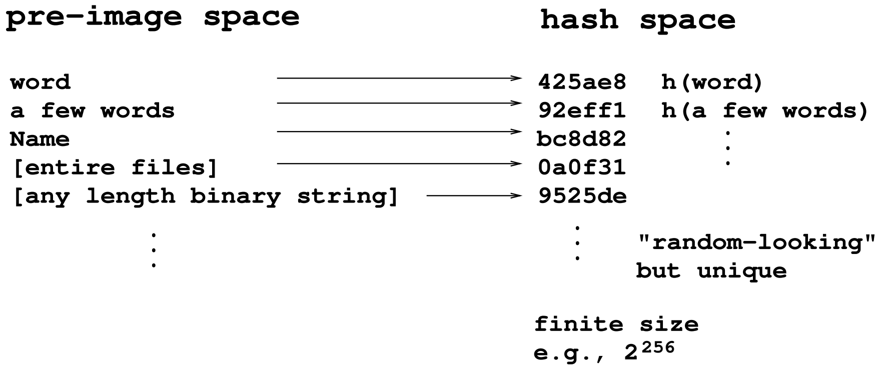
- a function that maps any objects to a concise digest
    - any particular object gives the same digest
    - the digest is fixed size
    - the digest doesn't tell you the object

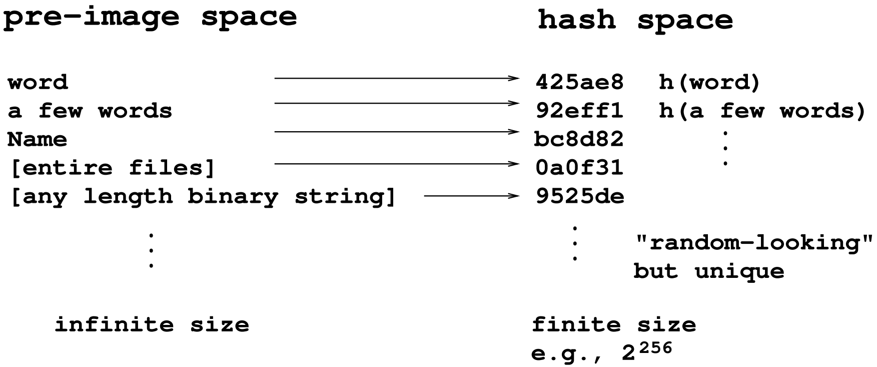
pre-image space	hash space
word	425ae8
a few words	92eff1
Name	bc8d82
[entire files]	0a0f31
[any length binary string]	9525de
⋮	⋮

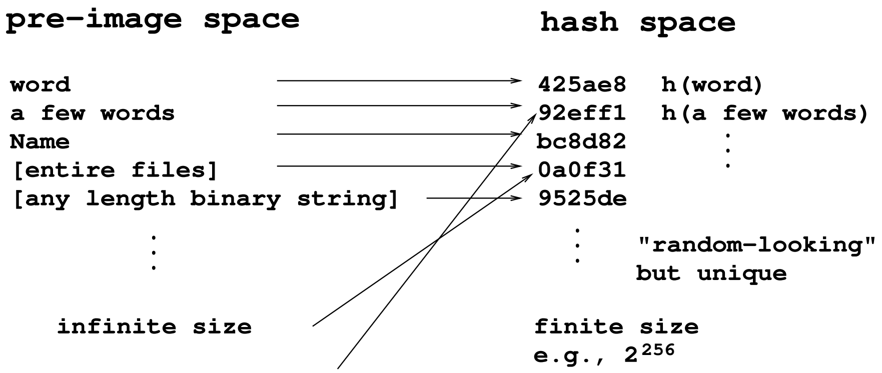
$$\begin{tikzcd}
\text{pre-image space} & \text{hash space} \\
\text{word} \arrow[r] & \text{425ae8} \\
\text{a few words} \arrow[r] & \text{92eff1} \\
\text{Name} \arrow[r] & \text{bc8d82} \\
\text{[entire files]} \arrow[r] & \text{0a0f31} \\
\text{[any length binary string]} \arrow[r] & \text{9525de} \\
\vdots & \vdots
\end{tikzcd}$$


pre-image space
hash space
word
a few words
Name
[entire files]
[any length binary string]
425ae8
92eff1
bc8d82
0a0f31
9525de
h(word)
h(a few words)


pre-image space
hash space
word
a few words
Name
[entire files]
[any length binary string]
425ae8
92eff1
bc8d82
0a0f31
9525de
h(word)
h(a few words)
"random-looking"
but unique


pre-image space
hash space
word → 425ae8 h(word)
a few words → 92eff1 h(a few words)
Name → bc8d82
[entire files] → 0a0f31
[any length binary string] → 9525de
"random-looking" but unique
finite size e.g., 2^{256}


pre-image space
hash space
word
a few words
Name
[entire files]
[any length binary string]
425ae8
92eff1
bc8d82
0a0f31
9525de
h(word)
h(a few words)
"random-looking"
but unique
infinite size
finite size
e.g., 2^{256}


pre-image space
hash space
word
a few words
Name
[entire files]
[any length binary string]
infinite size
425ae8
92eff1
bc8d82
0a0f31
9525de
h(word)
h(a few words)
"random-looking"
but unique
finite size
e.g., 2^{256}

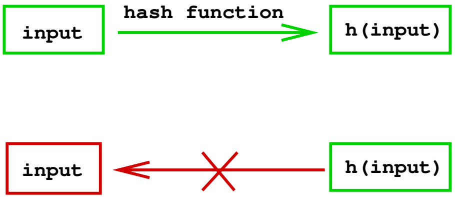
Hash Functions

- two kinds
  - cryptographically suitable and not cryptographically suitable
  - we only consider **cryptographically suitable** hash functions
  - **cryptographic hash function**
  - hash functions used for e.g., hash tables are something else
- hash functions are widespread and are used for all sorts of reasons
  - both in and out of cryptography

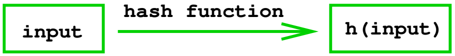
Three Goals

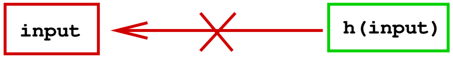
- H1: preimage resistant (one-way property)
  - given the description of a hash function H, and y such that $y = H(x)$ it is computationally infeasible to compute x
- H2: second preimage resistant
  - given the description of a hash function H, and x, it is computationally infeasible to compute $x' \neq x$ such that $H(x') = H(x)$
  - remember that the domain of hash functions is infinite while the range is finite
- H3: collision resistant
  - given the description of a hash function H, it is computationally infeasible to compute x and y such that $H(x) = H(y)$
  - this condition is surprising important
  - it also seems so unlikely: just **one** collision is needed
  - collision resistance implies second preimage resistance (why?)

$$\begin{tikzcd}\boxed{\texttt{input}} \arrow[r, "\texttt{hash function}"] & \boxed{\texttt{h(input)}}\end{tikzcd}$$

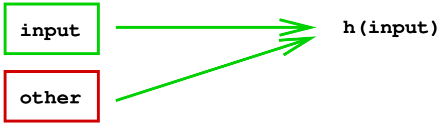


H1 pre-image resistance

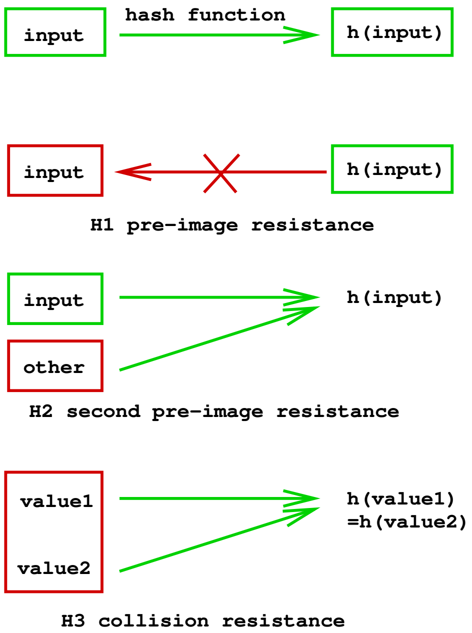




H1 pre-image resistance



H2 second pre-image resistance


input
hash function
h(input)
input
h(input)
H1 pre-image resistance
input
other
h(input)
H2 second pre-image resistance
value1
value2
h(value1)
=h(value2)
H3 collision resistance

The goal of a hash function is to implement a **random oracle**.

The goal of a hash function is to implement a **random oracle**. Random oracle gives some random value to every different possible input but crucially consistently gives the same value to same input.

The goal of a hash function is to implement a **random oracle**.
Random oracle gives some random value to every different possible input but crucially consistently gives the same value to same input.
H1–3 are trivially achieved.

Merkle–Damgård Construction

- define a secure (H1–3) **compression function**
  - compression function maps $\{0,1\}^{2 \cdot n} \rightarrow \{0,1\}^n$
  - i.e., it halves the number of bits each time in a collision resistant, one-way manner
  - takes two n-length inputs: IV and a message block
  - produces one n-length output: hash
- then to hash a message M:
  - add 0s to M until it is block aligned (i.e., length a multiple of n)
  - start with a fixed initialization vector IV
  - pass IV and first block to compression function
  - use hash output as IV input for next compression function
  - output the last result

Iterated Hashing

- divide input to hash into blocks $B_1, B_2, \ldots, B_n$, each of length 256
    - B_n zero padded if necessary
- $\textsc{cur} \leftarrow \mathsf{IV}$
- for $i = 1 \ldots n$
    - $\textsc{cur} \leftarrow \mathsf{H}(\textsc{cur}, B_i)$
- return $\textsc{cur}$

M–D construction guarantees that if the **compression function** is collision resistant then the **hash function** is collision resistant.

Standard hash functions use M–D construction: MD5, SHA1, SHA256.

Standard hash functions use M–D construction: MD5, SHA1, SHA256.
At the core they are a secure compression function.

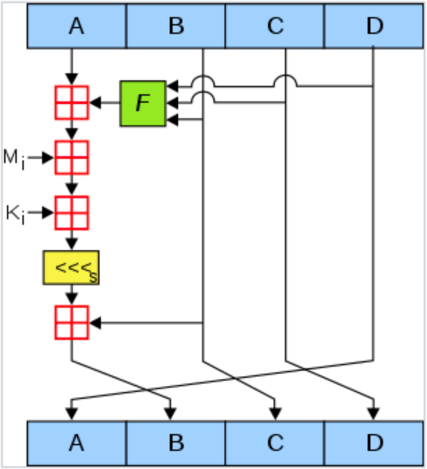


Figure 1. One MD5 operation. MD5 consists of 64 of these operations, grouped in four rounds of 16 operations. F is a nonlinear function; one function is used in each round. M_i denotes a 32-bit block of the message input, and K_i denotes a 32-bit constant, different for each operation. $<<<_s$ denotes a left bit rotation by s places; s varies for each operation. $\boxplus$ denotes addition modulo 2^{32}.

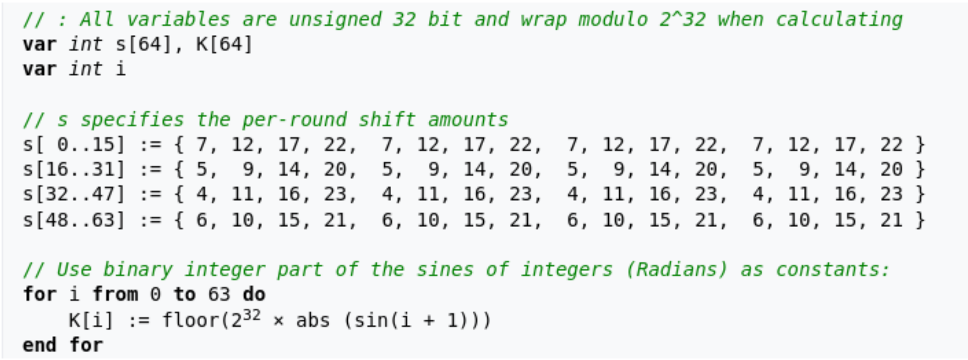
Algorithm [edit]

MD5 processes a variable-length message into a fixed-length output of 128 bits. The input message is broken up into chunks of 512-bit blocks (sixteen 32-bit words); the message is padded so that its length is divisible by 512. The padding works as follows: first a single bit, 1, is appended to the end of the message. This is followed by as many zeros as are required to bring the length of the message up to 64 bits fewer than a multiple of 512. The remaining bits are filled up with 64 bits representing the length of the original message, modulo 2^{64}.

The main MD5 algorithm operates on a 128-bit state, divided into four 32-bit words, denoted A, B, C, and D. These are initialized to certain fixed constants. The main algorithm then uses each 512-bit message block in turn to modify the state. The processing of a message block consists of four similar stages, termed *rounds*; each round is composed of 16 similar operations based on a non-linear function F, modular addition, and left rotation. Figure 1 illustrates one operation within a round. There are four possible functions; a different one is used in each round:

$$\begin{aligned} F(B,C,D) &= (B \wedge C) \vee (\neg B \wedge D) \\ G(B,C,D) &= (B \wedge D) \vee (C \wedge \neg D) \\ H(B,C,D) &= B \oplus C \oplus D \\ I(B,C,D) &= C \oplus (B \vee \neg D) \end{aligned}$$

$\oplus, \wedge, \vee, \neg$ denote the XOR, AND, OR and NOT operations respectively.



```
// : All variables are unsigned 32 bit and wrap modulo 2^32 when calculating
var int s[64], K[64]
var int i

// s specifies the per-round shift amounts
s[ 0..15] := { 7, 12, 17, 22,  7, 12, 17, 22,  7, 12, 17, 22,  7, 12, 17, 22 }
s[16..31] := { 5,  9, 14, 20,  5,  9, 14, 20,  5,  9, 14, 20,  5,  9, 14, 20 }
s[32..47] := { 4, 11, 16, 23,  4, 11, 16, 23,  4, 11, 16, 23,  4, 11, 16, 23 }
s[48..63] := { 6, 10, 15, 21,  6, 10, 15, 21,  6, 10, 15, 21,  6, 10, 15, 21 }

// Use binary integer part of the sines of integers (Radians) as constants:
for i from 0 to 63 do
    K[i] := floor($2^{32}$ × abs (sin(i + 1)))
end for
```

SHA1 is similiar but a bit more complicated.

SHA1 is similiar but a bit more complicated.
SHA256 is similiar but a bit more complicated.

SHA1 is similiar but a bit more complicated.
SHA256 is similiar but a bit more complicated.
Use SHA256 whenever you need a hash (for now).

SHA1 is similiar but a bit more complicated.
SHA256 is similiar but a bit more complicated.
Use SHA256 whenever you need a hash (for now).
SHA3 does not use the M–D construction

Uses of Hash Functions

Unique Digests

- have a unique way of representing some data or state
- unique file names
  - e.g., distributed hash tables where hash of data is the key
- data deduplication in cloud storage
  - e.g., email service where multiple users have the same attachment

Message Signing

- public-key crypto (discussed later) allows us to “digitally sign” messages
    - anyone can verify, only we can produce signature
    - this is an expensive operation
- small digest **represents** the full message
- so instead of signing a message, we only sign its hash
    - what does this mean for security?
        - what would go wrong if H1, H2, or H3 didn’t hold?
        - would it be obvious?

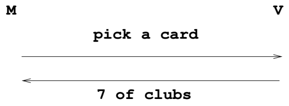
Chained Hash

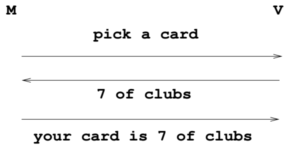
- used to prove the order of messages
  - trivial if you include the entire earlier message
  - more efficient: include the hash of it instead
- shows that everything in one message was created knowing the preimage
- what hash property prevents creating alternative history?

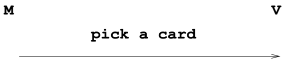
Commitment

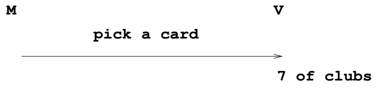
- used in lots protocols
- idea is that one party **commits** to a value
- the value can be later **revealed**
- the commitment alone gives no information about the value (under computational bound), but a different value cannot be given
- allows you to get information without being allowed to change your answers based on what you learn
    - proves that the message “existed” before the commitment is created
- how to implement this?

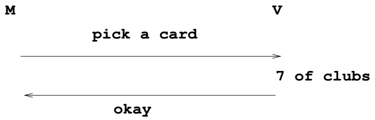
$$\begin{tikzcd}
\mathbf{M} \arrow[r, "\texttt{pick a card}"] & \mathbf{V}
\end{tikzcd}$$

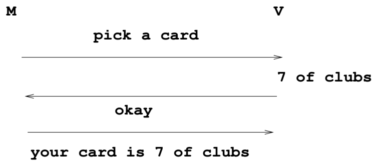

M
V
pick a card
7 of clubs

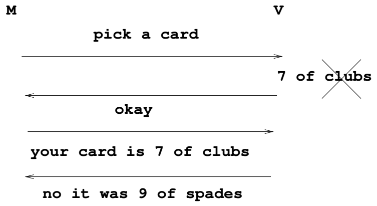

M
V
pick a card
7 of clubs
your card is 7 of clubs

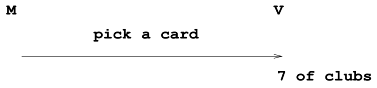

M
V
pick a card

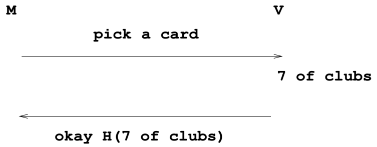

M
V
pick a card
7 of clubs

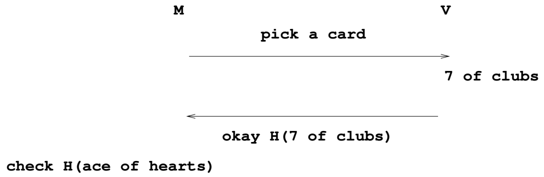

M
V
pick a card
7 of clubs
okay

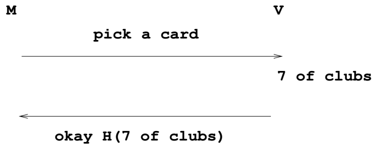

M
V
pick a card
7 of clubs
okay
your card is 7 of clubs


M
V
pick a card
7 of clubs
okay
your card is 7 of clubs
no it was 9 of spades

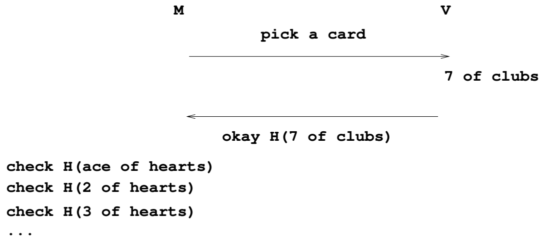

M
V
pick a card
7 of clubs

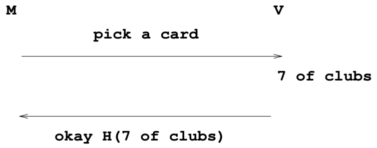

M
V
pick a card
7 of clubs
okay H(7 of clubs)


M
V
pick a card
7 of clubs
okay H(7 of clubs)
check H(ace of hearts)



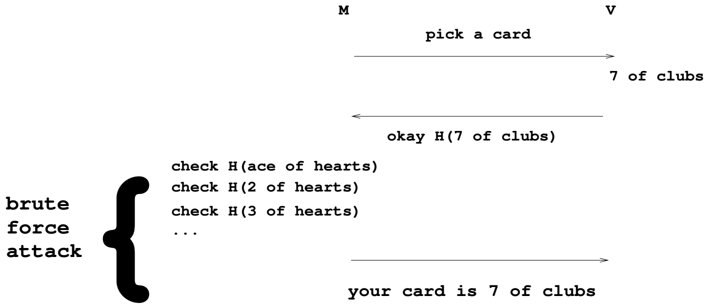
check H(ace of hearts)
check H(2 of hearts)

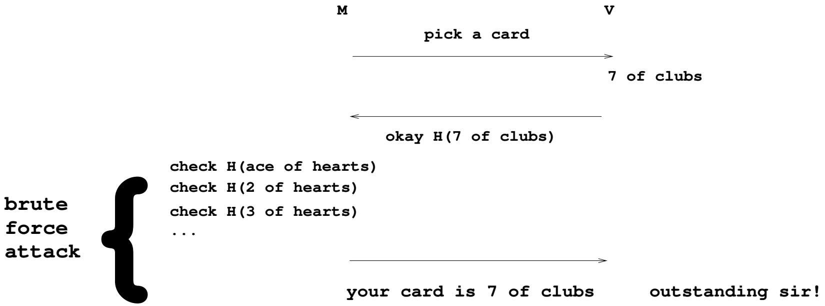

M
V
pick a card
7 of clubs
okay H(7 of clubs)
check H(ace of hearts)
check H(2 of hearts)
check H(3 of hearts)
...

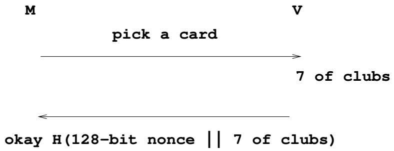


brute force attack

```
check H(ace of hearts)
check H(2 of hearts)
check H(3 of hearts)
...
```


M
V
pick a card
7 of clubs
okay H(7 of clubs)
check H(ace of hearts)
check H(2 of hearts)
check H(3 of hearts)
...
brute force attack
your card is 7 of clubs


M
V
pick a card
7 of clubs
okay H(7 of clubs)
check H(ace of hearts)
check H(2 of hearts)
check H(3 of hearts)
...
brute force attack
your card is 7 of clubs
outstanding sir!


M
V
pick a card
7 of clubs
okay H(128-bit nonce || 7 of clubs)

Checksums (Integrity)

- check that file downloaded correctly
- hash the file and check it against the “checksum”
    - the file of the message
- if any bit changes, even in checksum, it will be noticed
- you know you have the message in its entirety

Computing a checksum:

- `cat filename | sha256sum`
- `cat filename | sha256sum`
- `sha256sum file1 file2 ...`
- `echo -n hello | sha256sum`
- other hash functions operate similar
  - e.g., md5sum, sha1sum

Do checksums prevent Eve from modifying messages?

Do checksums prevent Eve from modifying messages?
Only if Eve is passive, in which case she can't alter it.

Do checksums prevent Eve from modifying messages?
Only if Eve is passive, in which case she can't alter it.
Protects against random bit errors; not **deliberate ones**.

Do checksums prevent Eve from modifying messages?
Only if Eve is passive, in which case she can't alter it.
Protects against random bit errors; not **deliberate ones**.
These are the kind of things that **adversaries** do
(low probability faults).

Do checksums prevent Eve from modifying messages?
Only if Eve is passive, in which case she can't alter it.
Protects against random bit errors; not **deliberate ones**.
These are the kind of things that **adversaries** do
(low probability faults).
i.e., the failure conditions can be all
the worst things possible for you.

Message Authentication Codes (MAC)

- used to **authenticate** the message
  - know the origin of the message
- provides message **integrity**
  - make sure it wasn't changed in transmission
- these are also called **tags**
  - message and tag
- the symmetric-key version of signatures
  - both parties can generate a MAC
  - does not not prove who sent it
  - **epistemic** evidence for two parties
  - not **non-repudiable** to outsiders

MAC Security

- MAC is a function $f_k(m) \to t$
  - f is a public function
  - k is a secret key
  - m is a message
  - t is the corresponding tag
- security assumptions
  - adversary is assumed to not know k
  - adversary is given some number of (m_i, t_i) pairs where $t_i = f_k(m_i)$
- **existential forgery**
  - adversary computes (m, t) where $t = f_k(m)$
- **selective forgery**
  - adversary is given m and computes t where $t = f_k(m)$
- **verifiable forgery**
  - adversary is given m, t and can determine if $t = f_k(m)$
- **key recovery**
  - an adversary determines k

Implementing MACs

- main idea: hash the message and a key somehow together
  - tag is this hash
  - a **keyed hash function**
- security based on the key being secret
  - if message changes, the hash won't match
  - adversary cannot compute the tag for an altered message
- how can we implement this?

Try 1: $H(k|m)$

Try 1: $H(k|m)$
any problems?

Try 1: $H(k|m)$

any problems?

for M–D hash functions if I have $H(k|m)$

then I can compute $H(k|m|m')$

(modulo block alignment)

Try 1: $H(k|m)$

any problems?

for M–D hash functions if I have $H(k|m)$

then I can compute $H(k|m|m')$

(modulo block alignment)

meaning I can append to a message

and compute Alice's MAC for it

Length Extension Example

Length Extension Example
ATTACK AT DAWN

Length Extension Example

ATTACK AT DAWN
WOULD BE A BAD IDEA
ATTACK AT DUSK INSTEAD

Another Extension Example

Another Extension Example
I will pay you \$100

Another Extension Example

I will pay you $100

,000,000

Another Extension Examples
given tag of URL
$/\textsc{q}?\textsc{field}1{=}\textsc{true}\&\textsc{field}2{=}12$ compute:

Another Extension Examples
given tag of URL
$/\textsc{q}?\textsc{field}1{=}\textsc{true}\&\textsc{field}2{=}12$ compute:
$/\textsc{q}?\textsc{field}1{=}\textsc{true}\&\textsc{field}2{=}123$

Another Extension Examples

given tag of URL

/q?field1=true&field2=12 compute:

/q?field1=true&field2=123

/q?field1=true&field2=12&field3=extra

Another Extension Examples

given tag of URL

/q?field1=true&field2=12 compute:

/q?field1=true&field2=123

/q?field1=true&field2=12&field3=extra

/q?field1=true&field2=12&field1=false

```
class some_class:
        def good(self):
                    print "good"

s = some_class()
s.good()
```

```
class some_class:
        def good(self):
                print "good"

s = some_class()
s.good()


def evil(self):
    print "bad"
from types import MethodType
s.good = MethodType(evil, s)

s.good()
```

Try 2: $H(m|k)$

Try 2: $H(m|k)$
any problems?

Try 2: $H(m|k)$

any problems?

suppose H collides for m, m',

then tag is valid for both m and m'

Try 2: $H(m|k)$

any problems?

suppose H collides for m, m',

then tag is valid for both m and m'

this shouldn't happen (H3)

Try 2: $H(m|k)$

any problems?

suppose H collides for m, m',

then tag is valid for both m and m'

this shouldn't happen (H3)

but why not defend against it anyway!

Hash-Based MAC (HMAC)

- standard for MAC
  - always use this
  - **never hash a key** if you cannot articulate why HMAC is not suitable
  - TIME-TESTED TOOLS
- $\text{HMAC}(k, m) = H((K' \oplus \text{opad}) \| H((K' \oplus \text{ipad} \| m))$
  - K' is either H(k) or k with zeros to match blocksize of H
  - opad is `0x5c5c`...
  - ipad is `0x3636`...
- HMAC defeats all known attacks
  - assuming that H is a secure hash function

It’s encrypt-then-MAC.

1. Encrypt message with key.
2. HMAC ciphertext with key.
3. Append MAC to ciphertext.

It's encrypt-then-MAC.

1. Encrypt message with key.
2. HMAC ciphertext with key.
3. Append MAC to ciphertext.

On receipt, check the MAC is
valid for the ciphertext
before trying to decrypt.

Collisions in hash functions

When a hash function has a collision,

When a hash function has a collision,
we **stop using it** and replace it.

When a hash function has a collision,
we **stop using it** and replace it.
MD5 has a collision, so we used SHA1.

When a hash function has a collision,
we **stop using it** and replace it.
MD5 has a collision, so we used SHA1.
SHA1 has a collision, so we used SHA256.

MD5 Collision

4dc968ff0ee35c209572d4777b721587d36fa7b21bd
c56b74a3dc0783e7b9518afbfa20**0**a8284bf36e8e4b
55b35f427593d849676da0d1**5**55d8360fb5f07fea2

collides with

4dc968ff0ee35c209572d4777b721587d36fa7b21bd
c56b74a3dc0783e7b9518afbfa20**2**a8284bf36e8e4b
55b35f427593d849676da0d1**d**55d8360fb5f07fea2

But why is just having access to
one single collision so bad?

First, one collision can suggest a technique to make more.

First, one collision can suggest a technique to make more.
Collisions are typically not found by brute force alone.

Second, collisions in the compression function create more in the hash function.

Second, collisions in the compression function create more in the hash function. in particular, all suffices match:

$$H(x) = H(y) \Rightarrow H(x|z) = H(y|z)$$

(modulo block alignment)

Second, collisions in the compression
function create more in the hash function.
in particular, all suffices match:

$$H(x) = H(y) \Rightarrow H(x|z) = H(y|z)$$

(modulo block alignment)
if victim is willing to sign xz,
it is a valid signature for yz too.

Document formats

- postscript (printer format) has if constructs
  - if (x == y)
    - print good
  - else
    - print bad
- suppose H("if (x") = H("if (y")
  - if (y == y)
    - print good
  - else
    - print bad
- these two documents have the same hash
  - one appears as "good"
  - the other appears as "bad"
  - two arbitrary different messages

More Examples

- Microsoft documents have “macros”
    - can change what is presented on screen
- PDF has /If and /Javascript directives
- HTML has clientside JavaScript that can run
- MKV videos can have multiple track (even video)
    - track metadata identify if it is default

MD5 known weak in 2004

MD5 known weak in 2004

Still used for **certificates** in 2008

MD5 known weak in 2004

Still used for **certificates** in 2008

Prefix collision attack used to fake a certificate

MD5 known weak in 2004

Still used for **certificates** in 2008

Prefix collision attack used to fake a certificate

Still used by **Microsoft** in 2012

MD5 known weak in 2004
Still used for **certificates** in 2008
Prefix collision attack used to fake a certificate
Still used by **Microsoft** in 2012
Flame malware used prefix
collision in MS's **CODE UPDATES**

Iran, to include potential Israeli response. *For information about NSA's posture against Iran, see attached Iran Division produced paper on Iran.*

- (TS//SI//REL TO USA, FVEY) **Director Talking Point**: Emphasize that we have successfully worked multiple high-priority surges with GCHQ that have allowed us to refine maintaining mission continuity and seamless transition, and maximize our target coverage. These jointly-worked events include the storming of the British Embassy in Tehran, Iran's discovery of FLAME, and support to policymakers during the multiple rounds of P5 plus 1 negotiations.

Finding Collisions takes about 1–2 days
on a cluster of 200 PS3s

So we switched to SHA1 but then in 2017...

SHAttered

The first concrete collision attack against SHA-1

https://shattered.io

CWI

Marc Stevens
Pierre Karpman

Google

Elie Bursztein
Ange Albertini
Yarik Markov

SHAttered

The first concrete collision attack against SHA-1

https://shattered.io

CWI

Marc Stevens
Pierre Karpman

Google

Elie Bursztein
Ange Albertini
Yarik Markov

Will my browser show me a warning?

Starting from version 56, released in January 2017, Chrome will consider any website protected with a SHA-1 certificate as insecure. Firefox ~~has this feature planned for early 2017~~ has deprecated SHA-1 as of February 24th, 2017.

Encryption makes channel secure

Encryption makes channel secure
Adding a MAC makes it authentic

Encryption makes channel secure
Adding a MAC makes it authentic
Assuming there is only other entity
with encryption key and MAC key.

Two questions remain:

Two questions remain:
how are encryption keys shared
in the first place if not over
an insecure channel (future lecture)

Two questions remain:
how are encryption keys shared
in the first place if not over
an insecure channel (future lecture)
how are encryption keys generated
(next lecture)