Hash Function

Lecture 03

Software Security Engineering

Winter 2023
Thompson Rivers University

Hash Function

- 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
- a function that maps any objects to a concise digest
  - any particular object gives the same digest
  - the digest size is fixed size
  - the digest does not tell you the object
- hash functions are widespread and are used for all sorts of reasons
  - both in and out of cryptography

Hash functions are an efficient hardware implementation of a mathematical object called a random oracle

Random Oracle:

- when you give it a binary string that it has not seen before
  - generates a novel purely-random response the moment and return it
  - that means knowing all other responses gives no information about the new one
- when you give it a binary string that it has seen before
  - reliable gives the response it gave before
- it always available, can always answer
- never exposes the mapping of outputs to inputs
  - e.g., cannot ask the oracle: what is an input that makes *x*.
  - only approach: guessing the right input

A hash function simulate a random oracle under computationally-bounded assumptions

Hash Functions as Random Oracles

- we want hash functions to look like random oracle
- this means the output looks randomly generated
- the output is deterministic given the same input
- it is impossible to predict the input for a given output
- yet a hash function is an algorithm that anyone can run as often as they want
    - BRUTE FORCE attacks to guess matching input

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' != 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?)

Hash Functions

- hash functions map all binary strings $\{0, 1\}^*$ to a fixed-length one (e.g., $\{0, 1\}^{256}$)
- in practice hash functions work by iterating over the message
  - divide the message into blocks
  - process each block in sequence
  - maintain a state between blocks
    - state has initial value
    - state is used to produce output (the hash)

Merkle-Damgard (M-D) Construction

- define a secure (H1-3) compression function
  - compression function maps $\{0, 1\}^{2.n} \longrightarrow \{0, 1\}^n$
  - i.e., it halves the number of bits each time in a collision resistant, one-way manner
  - takes two *n*-length input: Initialization Vector (IV) and a message
  - 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, … , B_n, each of length 256
    - B_n zero padded if necessary
- CUR ⟵ IV
- for i = 1 … n
    - CUR ⟵ H(CUR, B_i)
- return 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.

(i.e. any hash function that has been used in Internet or your applications uses one of hash functions)

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

At the core they are a secure compression function.

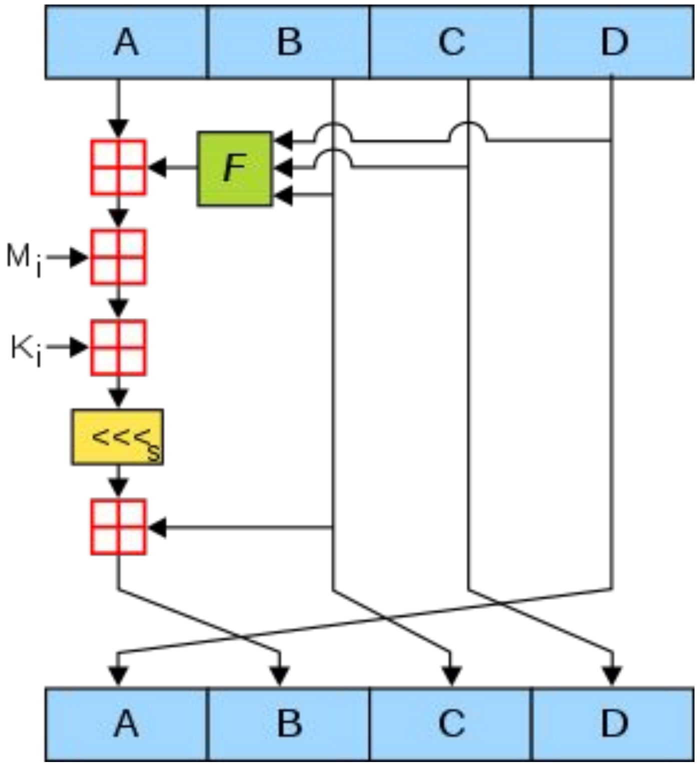
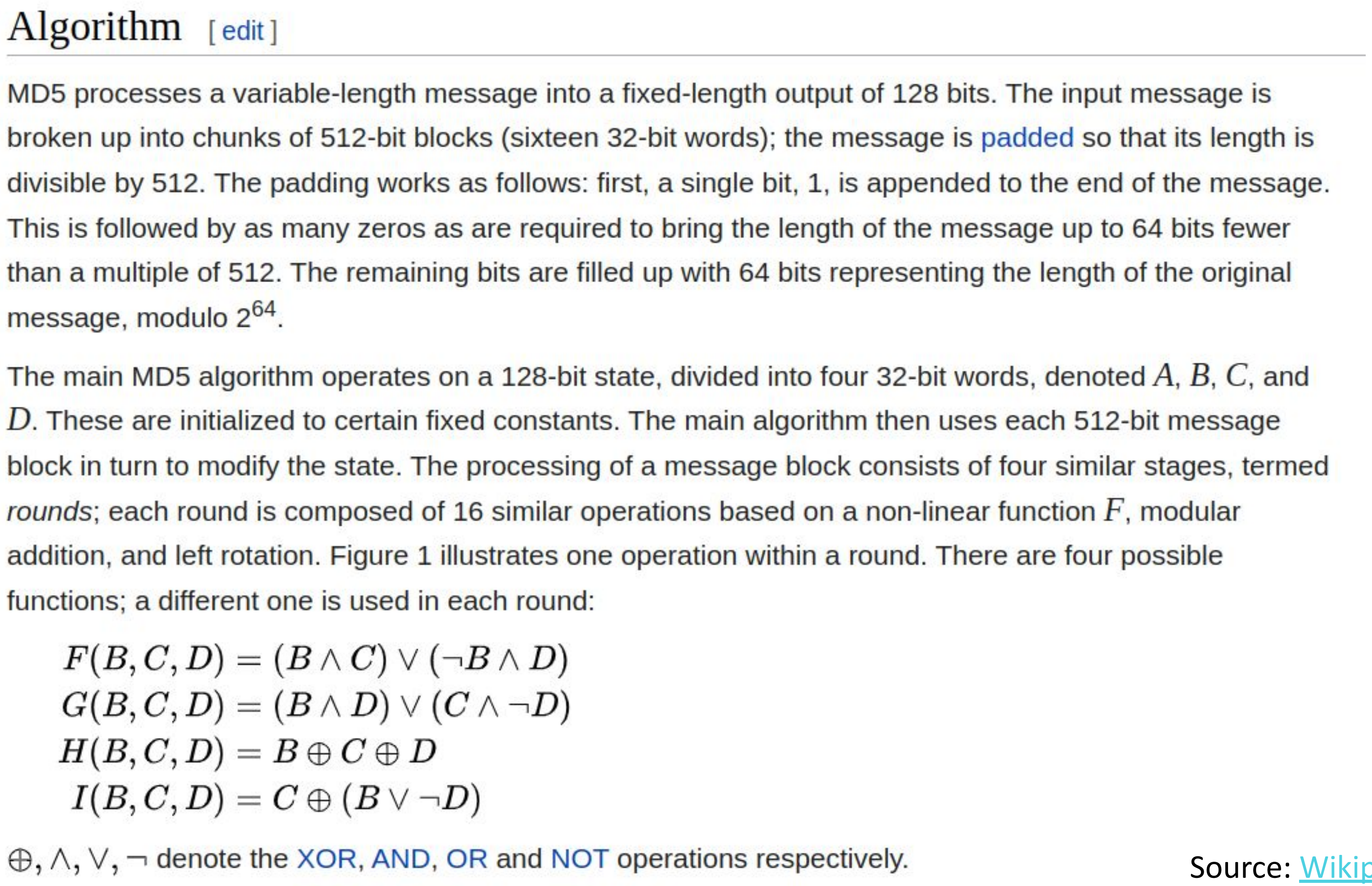


Figure 1. One MD5 operation.
MD5 consists of 64 of these operations, grouped in four rounds of 16 operations. *F* is a nonlinear function; one function is used in each round. M_i denotes a 32-bit block of the message input, and K_i denotes a 32-bit constant, different for each operation. $<<<_s$ denotes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

$\boxplus$ denotes addition modulo 2^{32}.

Source: Wikipedia



Algorithm [edit]

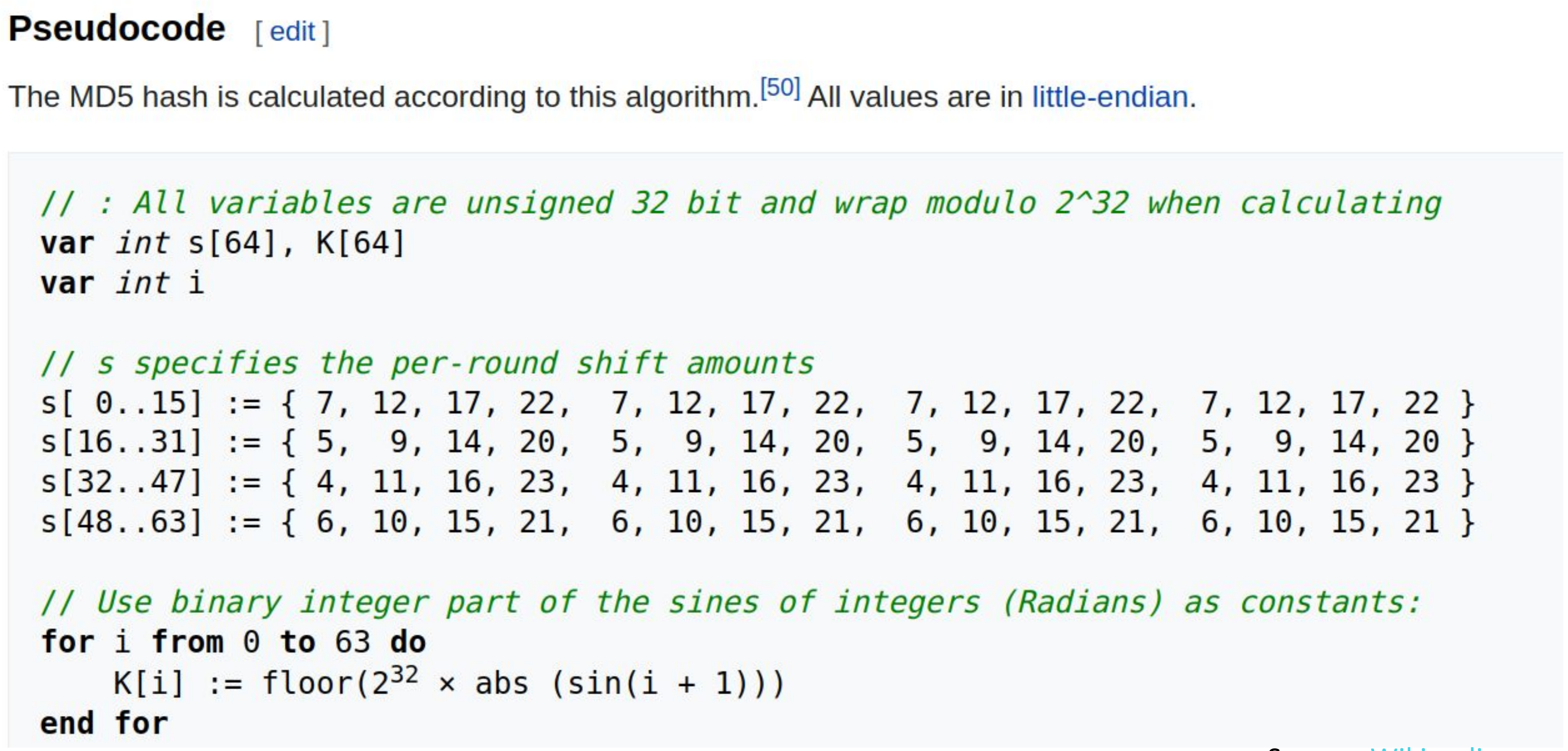
MD5 processes a variable-length message into a fixed-length output of 128 bits. The input message is broken up into chunks of 512-bit blocks (sixteen 32-bit words); the message is padded so that its length is divisible by 512. The padding works as follows: first, a single bit, 1, is appended to the end 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.

Source: Wikipedia



Pseudocode [edit]

The MD5 hash is calculated according to this algorithm.[50] All values are in little-endian.

```
// : 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
```

Source: Wikipedia

SHA1 is similar but a bit more complicated.

(SHA stands for Secure Hash Algorithm,

MD5 stands for 5th version of Message-Digest algorithm)

SHA1 is similar but a bit more complicated.

SHA256 is similar but a bit more complicated.

SHA1 is similar but a bit more complicated.

SHA256 is similar but a bit more complicated.

Use SHA256 whenever you need a hash (for now).

SHA1 is similar but a bit more complicated.

SHA256 is similar but a bit more complicated.

Use SHA256 whenever you need a hash (for now).

SHA3 uses a different construction, called a sponge construction. Why?

SHA1 is similar but a bit more complicated.

SHA256 is similar but a bit more complicated.

Use SHA256 whenever you need a hash (for now).

SHA3 uses a different construction, called a sponge construction. Why?

is developed to have a different kind of hash function available in case something go wrong with the basic design of MD construction.

Review

Random Oracle:

- when you give it a binary string that it has not seen before
  - generates a novel purely-random response the moment and return it
  - that means knowing all other responses gives no information about the new one
- when you give it a binary string that it has seen before
  - reliable gives the response it gave before
- it always available, can always answer
- never exposes the mapping of outputs to inputs
  - e.g., cannot ask the oracle: what is an input that makes *x*.
  - only approach: guessing the right input

Hash Functions as Random Oracles

- we want hash functions to look like random oracle
- this means the output looks randomly generated
- the output is deterministic given the same input
- it is impossible to predict the input for a given output
- yet a hash function is an algorithm that anyone can run as often as they want
    - BRUTE FORCE attacks to guess matching input

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' != 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?)

Hash Functions

- hash functions map all binary strings $\{0, 1\}^*$ to a fixed-length one (e.g., $\{0, 1\}^{256}$)
- in practice hash functions work by iterating over the message
  - divide the message into blocks
  - process each block in sequence
  - maintain a state between blocks
    - state has initial value
    - state is used to produce output (the hash)

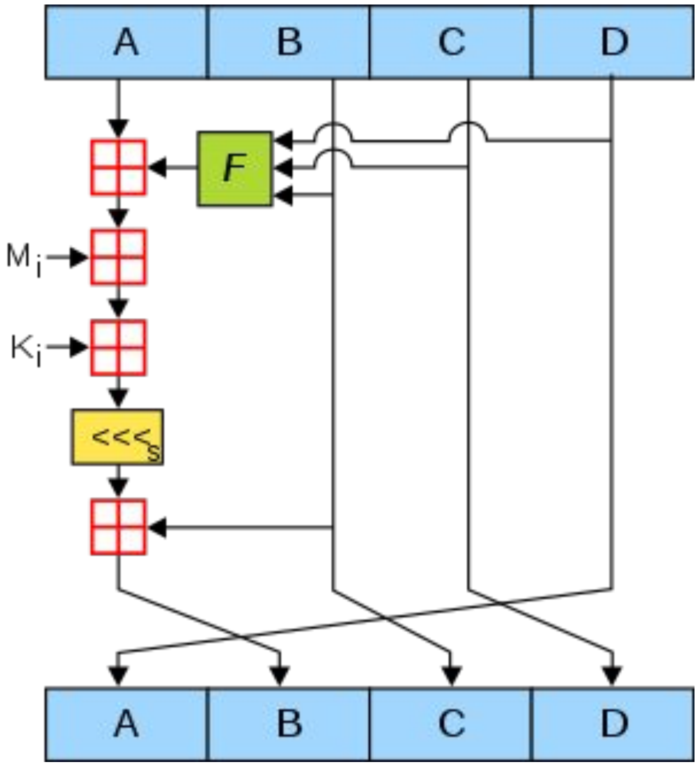


Figure 1. One MD5 operation.
MD5 consists of 64 of these operations, grouped in four rounds of 16 operations. *F* is a nonlinear function; one function is used in each round. M_i denotes a 32-bit block of the message input, and K_i denotes a 32-bit constant, different for each operation. $<<<_s$ denotes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

⊞ denotes addition modulo 2^{32}.

Source: Wikipedia

Uses of Hash Functions

Unique Digests

- have a unique way of representing some data or state
- unique file names
  - e.g., distributed hash tables
- version control software
  - e.g., git
- data deduplication in cloud storage

Public-key Signatures

- small digest represents the full message
- so instead of signing a message, we only sign its hash
  - what does this mean for security? (e.g., second preimage)
  - not necessary that collisions they are all going to look like random binary
    - e.g., blank lines, spaces, word choice

Public-key Signatures

- small digest represents the full message
- so instead of signing a message, we only sign its hash
  - what does this mean for security? (e.g., second preimage)
  - not necessary that collisions they are all going to look like random binary
    - e.g., blank lines, spaces, word choice
    - i.e., this collisions are not necessary random noises. Imagine you have a 128 bit message. if you have two space instead of one space, both messages have identical meaning. Thus, there are huge number of other 128 bit messages that have the same meaning. By using hash to sign a message, you actually signed all possible collisions to that message.
    - The security of this signature is that brute force of this possible collisions is not possible.

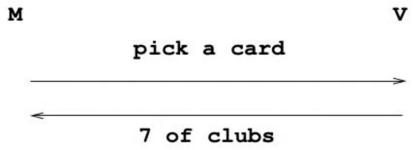
Chained Hash

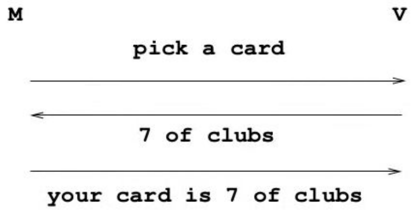
- used to prove the order of messages
  - trivial if you include the entire earlier message
  - more efficient: include the hash of it instead
  - m → m’ h(m) → m’’ h(m’ h(m)) …
- shows that everything in one message was created knowing the preimage
- not possible to create a loop (why?)

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?

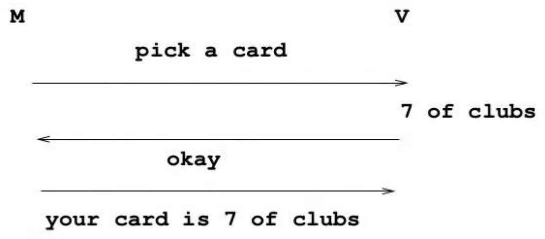
$$\begin{tikzcd}\mathbf{M} \arrow[r, "\texttt{pick a card}"] & \mathbf{V}\end{tikzcd}$$

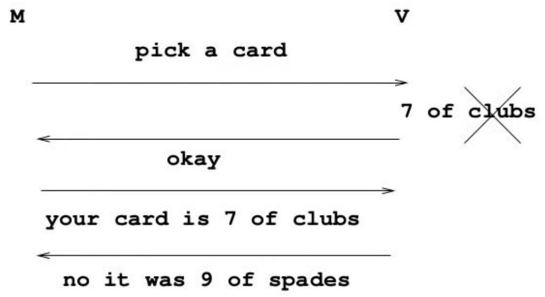

M
V
pick a card
7 of clubs

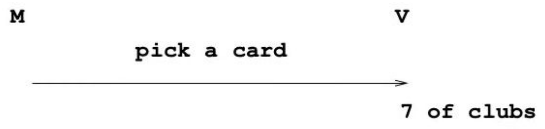

M
V
pick a card
7 of clubs
your card is 7 of clubs

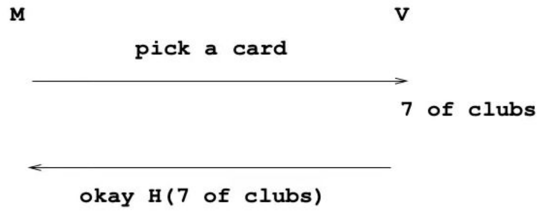
$$\begin{tikzcd}\mathbf{M} \arrow[r, "\texttt{pick a card}"] & \mathbf{V}\end{tikzcd}$$

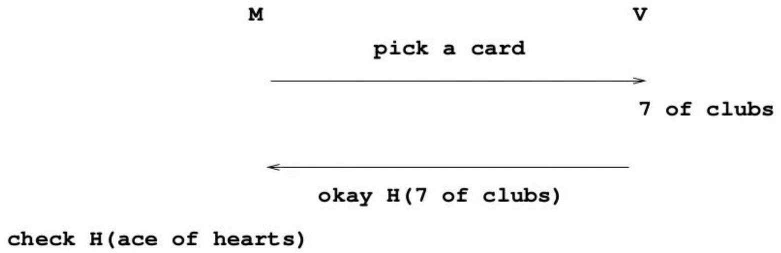
$$\begin{tikzcd} \mathbf{M} \arrow[r, "\text{pick a card}"] & \mathbf{V} \\ & \text{7 of clubs} \end{tikzcd}$$

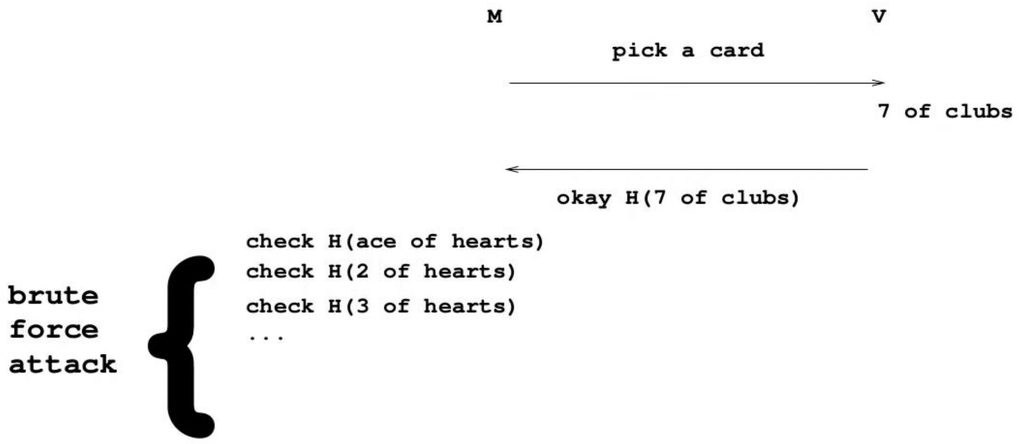

M
V
pick a card
7 of clubs
okay
your card is 7 of clubs

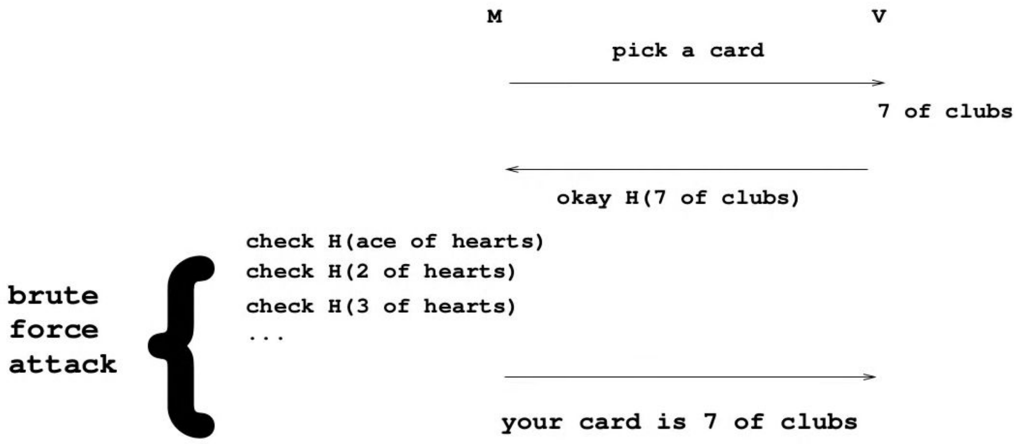

M
V
pick a card
7 of ~~clubs~~
okay
your card is 7 of clubs
no it was 9 of spades

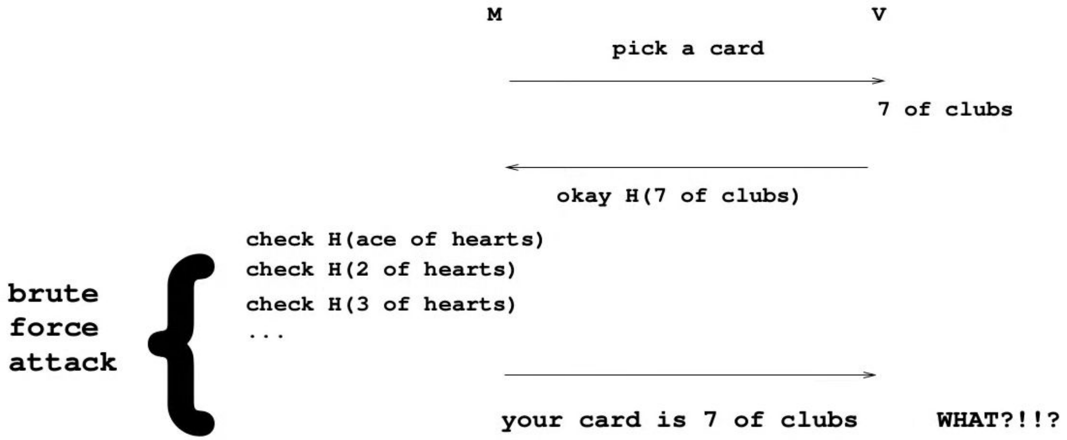

M
V
pick a card
7 of clubs


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)


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


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
WHAT?!!?

a small number of possible keys is vulnerable to brute force attack.

instead of hashing 7 of clubs, hash a random number and 7 of clubs

Checksums (Integrity)

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

Checksums (Integrity)

- integrity check to detect accidental corruption
- used to be computed with non-cryptographic hash functions
    - e.g., cyclic redundancy check (CRC)
- but now you will frequently see md5sums alongside files
- why is this only good for accidental checks?
    - e.g., consider downloading software that states the md5sum
- so why use MD5?

Computing a checksum via Unix Terminal

```
> cat filename | sha256sum

> cat filename | md5sum

> md5sum file1 file2 …

> echo -n hello | sha256sum
```

Does this protect messages from being altered by Eve?

Does this protect messages from being altered by Eve?

Only if Eve is passive, in which case she can’t alter it.

Does this protect messages from being altered by Eve?

Only if Eve is passive, in which case she can’t alter it.

Protects against random bit errors; not deliberate ones.

Does this protect messages from being altered by Eve?
Only if Eve is passive, in which case she can’t alter it.

Protects against random bit errors; not deliberate ones.

There are the kind of things that adversaries do
(low probability faults).

Does this protect messages from being altered by Eve?
Only if Eve is passive, in which case she can't alter it.

Protects against random bit errors; not deliberate ones.

There 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
- provide message integrity
  - make sure it wasn’t changed in transmission
- there are also called tags
  - message and tag
- the symmetric-key version of signatures
  - both parties can generate a MAC
  - does not prove who sent it
  - epistemic evidence for two parties
    - if Alice receives a message with its MAC, and show knows that she did not generate it, and Alice and Bob only can generate that message (have a shared key), she can conclude that Bob’ve sent the message
  - not non-repudiable to outsiders (unlike public key signature)

MAC Security (Adversarial Goals)

MAC security properties must deal with an adversary with these goals:

- existential forgery
  - an adversary without knowledge of *k* computes a valid tag *t* for any message *m*
- selective forgery
  - an adversary without knowledge of *k* computes a valid tag *t* for a particular *m*
- verifiable forgery
  - an adversary can determine if the *t* is valid for *m* with high probability
- 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)

we take key k, and hash with message m

Try 1: H(k|m)

any problem?

Try 1: $H(k|m)$

if Eve knows m and $H(k|m)$,
could she compute $H(k|m')$?

It turns out Eve can.

Try 1: H(k|m)

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)

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 add whatever to a message and
compute Alice's MAC for it

Try 1: H(k|m)

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 add whatever to a message and
compute Alice's MAC for it

This is known as Length Extension Attack

Length Extension Attack - Example

Length Extension Attack - Example

attack at dawn

Alice sends this message to Bob

Length Extension Attack - Example

attack at dawn

would be a bad idea

attack at eve instead

Eve appended a text to the message and send to Bob

Another Extension Example

given tag of `GET  /Q?FIELD1=TRUE&FIELD2=12`

Another Extension Example

given tag of `GET /Q?FIELD1=TRUE&FIELD2=12`

```
GET /Q?FIELD1=TRUE&FIELD2=123
```

Another Extension Example

given tag of `GET /Q?FIELD1=TRUE&FIELD2=12`

```
GET /Q?FIELD1=TRUE&FIELD2=123
```

```
GET /Q?FIELD1=TRUE&FIELD2=12&FIELD3=EXTRA
```

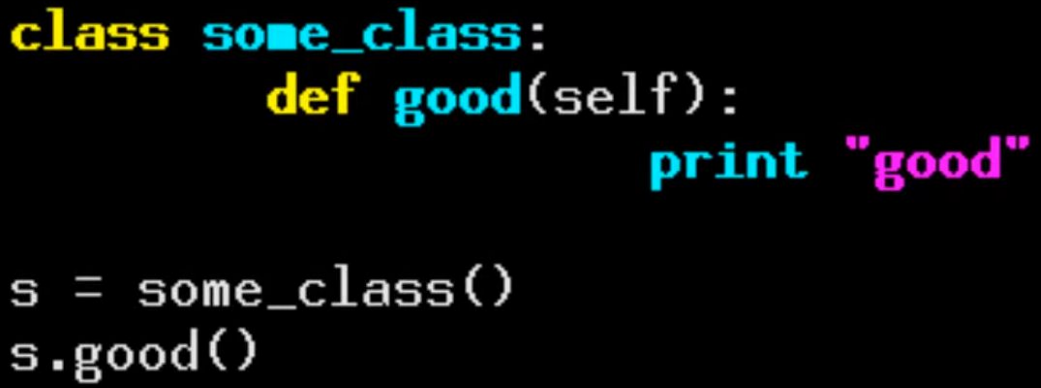
Another Extension Example

given tag of `GET /Q?FIELD1=TRUE&FIELD2=12`

```
GET /Q?FIELD1=TRUE&FIELD2=123
```

```
GET /Q?FIELD1=TRUE&FIELD2=12&FIELD3=EXTRA
```

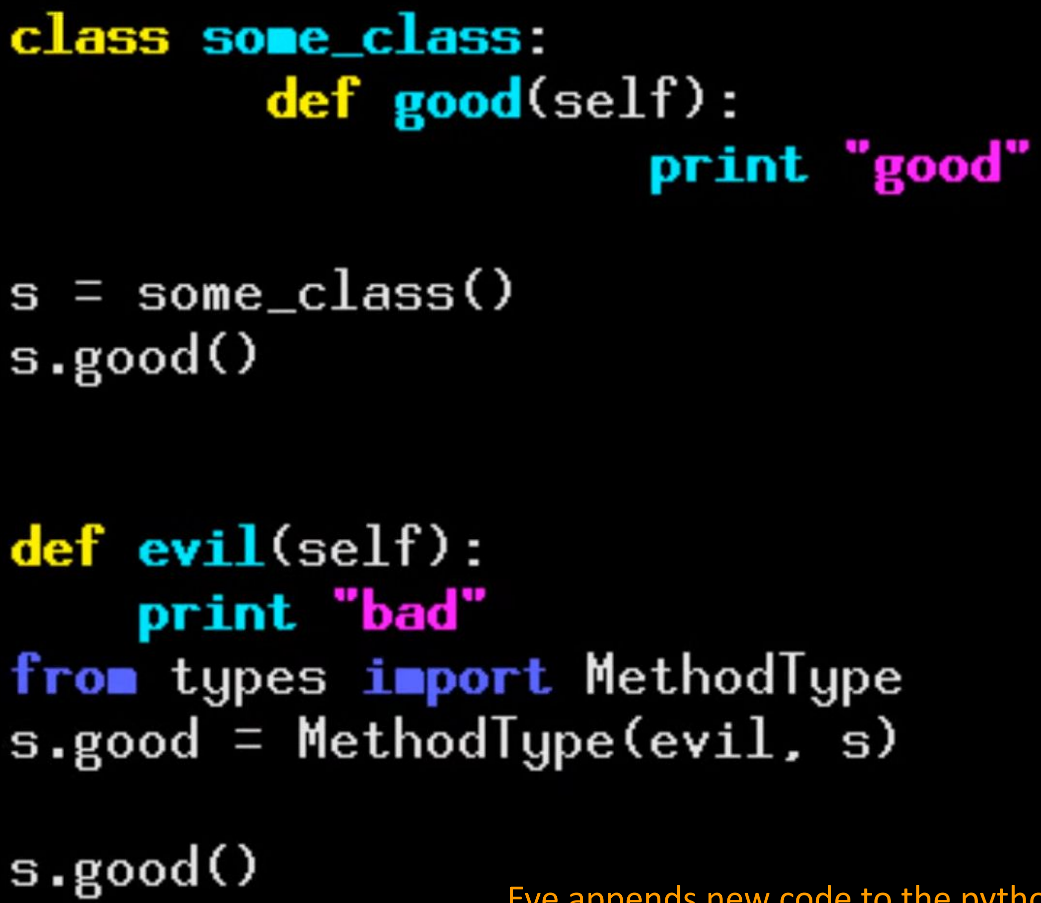
```
GET /Q?FIELD1=TRUE&FIELD2=12&FIELD1=FALSE
```



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

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

a python file



```
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()
```

Eve appends new code to the python file

Try 2: H(m|k)

this time we hash message follow by key

Try 2: $H(m|k)$

any problems?

Try 2: H(m|k)

any problems?

suppose H collides for m, m’
tag is valid for both m and m’

this means Eve can make a valid tag for a message m’ without having key k

Try 2: H(m|k)

any problems?

suppose H collides for m, m’
tag is valid for both m and m’

of course, this shouldn’t happen
since we design collision resistant Hash function
but why not defend against it anyway!

Hash-Based MAC (HMAC)

- standard for MAC
    - always use this, never hash a key without a good explanation why HMAC doesn't work
    - `TIME-TESTED TOOLS`
- HMAC(k, m) = H((k' ⊕ opad) || H((K' ⊕ 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

|| is concatenation

$$\mathrm{HMAC}(K, m) = \mathrm{H}\Big(\big(K' \oplus opad\big) \parallel \mathrm{H}\big(\big(K' \oplus ipad\big) \parallel m\big)\Big)$$

$$K' = \begin{cases} \mathrm{H}(K) & \text{if } K \text{ is larger than block size} \\ K & \text{otherwise} \end{cases}$$

where

H is a cryptographic hash function.

m is the message to be authenticated.

K is the secret key.

K' is a block-sized key derived from the secret key, *K*; either by padding to the right with 0s up to the block size, or by hashing down to less than or equal to the block size first and then padding to the right with zeros.

ǁ denotes concatenation.

⊕ denotes bitwise exclusive or (XOR).

opad is the block-sized outer padding, consisting of repeated bytes valued 0x5c.

ipad is the block-sized inner padding, consisting of repeated bytes valued 0x36.

no need to memorize this function

to provide encryption with authenticity

use encrypt-then-MAC technique

It’s encrypt-then-MAC.

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

Collisions in Hash Functions

when a hash function has a collision,

when a hash function has a collision,
we throw it out and use a new one.

when a hash function has a collision,
we throw it out and use a new one.

MD5 has a collision.

when a hash function has a collision,
we throw it out and use a new one.

MD5 has a collision.

SHA1 also has a collision.

when a hash function has a collision,
we throw it out and use a new one.

MD5 has a collision.

SHA1 also has a collision.

That’s why we must not use it for important things.

MD5 Collision

4dc968ff0ee35c209572d4777b721587d36fa7b21bd
c56b74a3dc0783e7b9518afbfa200a8284bf36e8e4b
55b35f427593d849676da0d1555d8360fb5f07fea2

collides with

4dc968ff0ee35c209572d4777b721587d36fa7b21bd
c56b74a3dc0783e7b9518afbfa202a8284bf36e8e4b
55b35f427593d849676da0d1d55d8360fb5f07fea2

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.

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

Collisions are typically not found by brute force alone.

It reveals through a clever crypto analysis of the hash function to figure out two input messages have the same output.

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:

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

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

in particular, all suffices match:

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

if victim is willing to sign xz, then it is a signature for yz too.

Prefix collision attack

Document formats

- postscript (printer format) has if constructs
  - if (x == y)
    - print bad
  - else
    - print good
- suppose H(“`if(x`”) = H(“`if(y`”) (a collision happens)
  - if (y == y)
    - print bad
  - else
    - print good

Eve can send another document

Microsoft documents have “macros” which is code that change how document look.

MD5 known weak in 2004

ideally we shouldn’t use it anymore

MD5 known weak in 2004

Still used for certificates in 2008

certificates are underpin security for day-to-day basis (we will cover it next later)

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

Flame was used for targeted cyber espionage in Middle Eastern countries.

Flame attacks computers running the Microsoft Windows operating system. The program is used for targeted cyber espionage in Middle Eastern countries.

Flame attacks computers running the Microsoft Windows operating system. The program is used for targeted cyber espionage in Middle Eastern countries.

Flame can spread to other systems over a local network (LAN). It can record audio, screenshots, keyboard activity and network traffic. The program also records Skype conversations and can turn infected computers into Bluetooth beacons which attempt to download contact information from nearby Bluetooth-enabled devices.

Flame attacks computers running the Microsoft Windows operating system. The program is used for targeted cyber espionage in Middle Eastern countries.

Flame can spread to other systems over a local network (LAN). It can record audio, screenshots, keyboard activity and network traffic. The program also records Skype conversations and can turn infected computers into Bluetooth beacons which attempt to download contact information from nearby Bluetooth-enabled devices.

Flame "is certainly the most sophisticated malware we encountered during our practice; arguably, it is the most complex malware ever found."

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.

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.