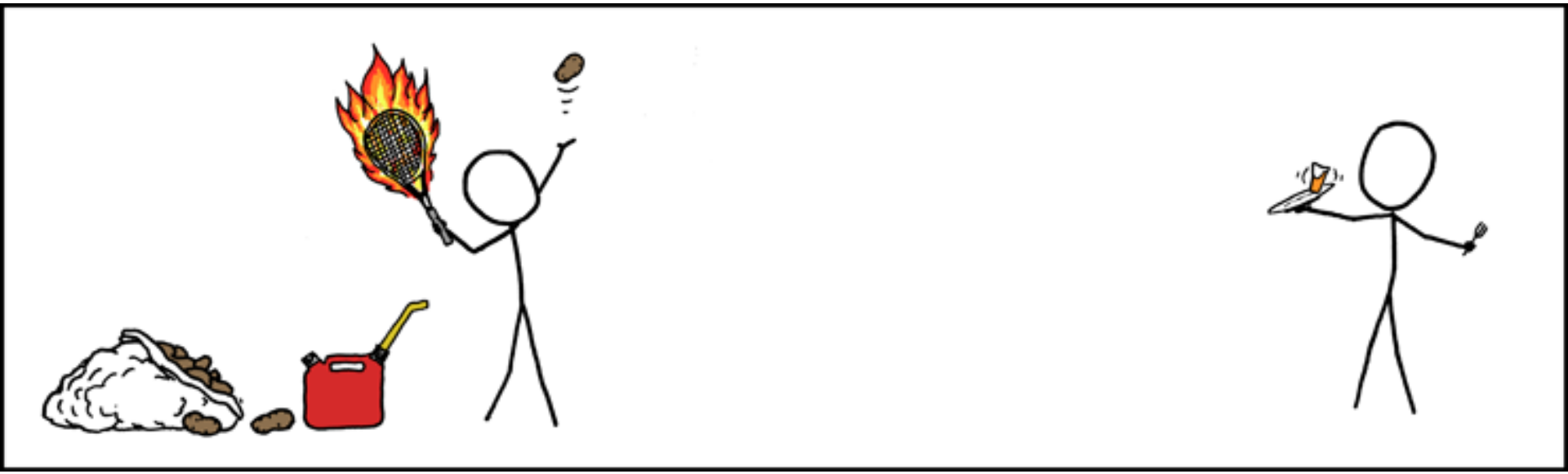
CS 55:
Security and Privacy

Hashing

Hash browns



Source: https://xkcd.com/421/

Let’s start with a game

Rules

You and I will both pick a number, if the sum is:

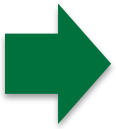
- Even – I win
- Odd – you win

Protocol

- Tell me your number!
- I will then tell you mine

Adapted from Du, Wenliang. *Computer & Internet Security: A Hands-on Approach*. 2019.

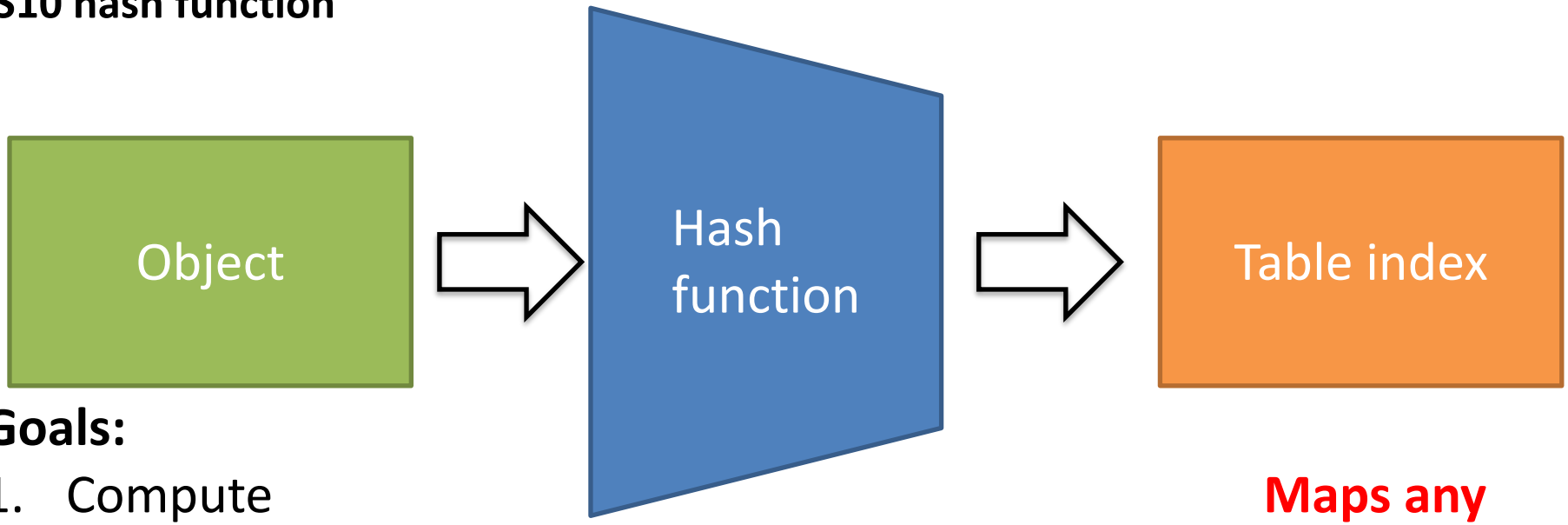
Agenda



1. Hashing intro
2. Common hash functions
3. Hashing use cases
4. MACs and attacks

Back in CS10 we looked at hashing to find an index in table based on an object

CS10 hash function



Goals:

1. Compute quickly and consistently
2. Spreads keys over the table
3. Small changes give different numbers

Step 1: convert object to integer

- Range –infinity to infinity
- Use hashCode()

Step 2: constrain to fall within hash table (hashCode %m)

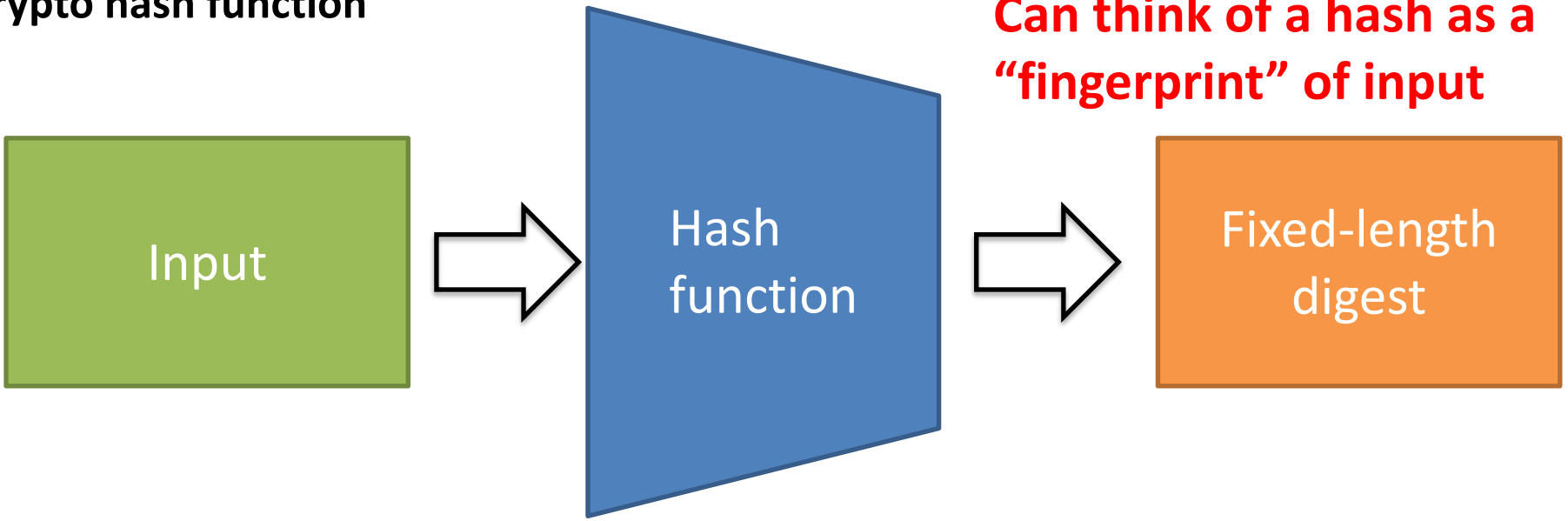
Maps any object into table index from 0..m-1

Output is fixed length

Crypto hash functions add two properties: irreversibility and collision resistance

Crypto hash function

Can think of a hash as a "fingerprint" of input



Input:
"my secret message"

Digest:
061ef7d71233886440043538e4d91de3

Hash function is a deterministic mathematical function

1. Irreversible – Cannot find plain text in "reasonable" amount of time given only the hash digest output (one-way)
2. Collision resistant – different plaintext do not produce same hash digest

Pseudorandom - change any bit and get very different result

Let's visit the game with a new protocol

Rules

You and I will both pick a number, if the sum is:

- Even – I win
- Odd – you win

Protocol

- I hash my number and give you the result (commitment)
- You then tell me your number
- I then tell you my number
- You can confirm I didn't cheat by hashing my number and comparing with commitment

Is this fair to both parties? Why or why not?

What could be a problem with this protocol?

Adapted from Du, Wenliang. *Computer & Internet Security: A Hands-on Approach*. 2019.

Agenda

1. Hashing intro
2. Common hash functions
3. Hashing use cases
4. MACs and attacks

MD5 and SHA are popular choices, but SHA-2 is currently the most popular choice

MD5 (Message Digest 5)

- Popular choice of hash function
- Designed by Ron Rivest of RSA fame in 1991
- Collision resistance broken in 2004
- Can still use, but not if collision resistance is important

SHA (Secure Hash Algorithm)

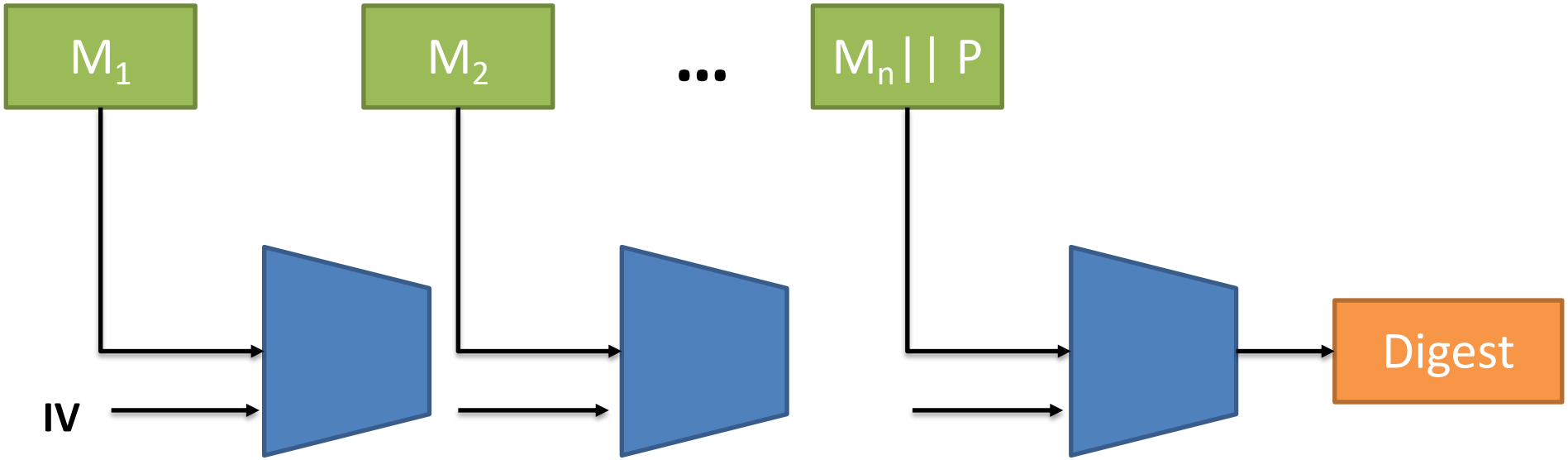
SHA-2 the most common choice today

Don't use MD5 anymore for most purposes

- SHA-1
  - 160-bit hash function
  - Not recommended after 2005
  - Collision found in 2017 (two different pdfs gave same hash)
- SHA-2
  - Two popular flavors: SHA-256 and SHA-512 (number = length of hash)
  - No known exploits… yet
- SHA-3
  - New as of 2015
  - Works differently from MD5 and SHA-2 (in case gets SHA2 broken)

MD5, SHA-1 and SHA-2 use the Merkle-Damgard construction

Merkle-Damgard



Message split into fixed sized blocks, processed one at a time
Fixed size current state initialized by IV
Each step uses the output of the prior step as input
Digest is fixed length:

- MD5: 16 bytes (128 bits)
- SHA-256: 32 bytes (256 bits)
- SHA-512: 64 bytes (512 bits)

SHA-256 returns a 256-bit (32 byte) digest despite arbitrary length input

SHA-256

Break message in 512-bit blocks, pad last block

SHA256("abc")

Input: abc (string)

bytes: [97, 98, 99] #ascii values

Message block: 00110000101100010011000 11 #abc in binary

Pad message to make length a multiple of 512 bits:

- Add one bit
- Fill with zeros until last 64 bits
- Last 64 bits are length of message

Message block: 00110000101100010011000 11 1000…11000 (length 24 bits = 3 bytes)

Message block is now 512 bits long

SHA256("abc")

= ba7816bf8f01cfea414140de5dae2223b00361a396177a9cb410ff61f20015ad

Adapted from https://www.youtube.com/watch?v=DMtFhACPnTY, see https://youtu.be/f9EbD6iY9zI for detailed explanation of how SHA-2 works

SHA-256 returns a 256-bit (32 byte) digest despite arbitrary length input

SHA-256

Simplified

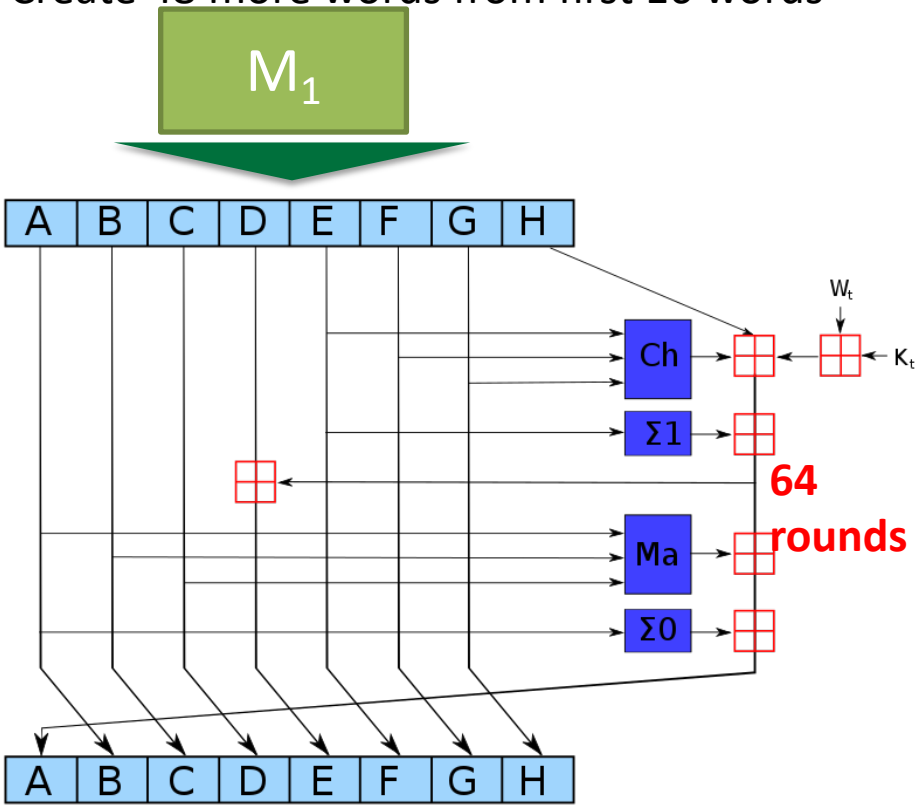
Break message in 512-bit blocks, pad last block

Calculate initial state of eight 32-bit variables based on square root of first 8 prime numbers

Calculate constants K = cube root of first 64 primes

Split input in to 16 words of length 32 bits (512 bits)

Create 48 more words from first 16 words



Compress

- **State (A—H) is 256 bits long**
- **Changes made to state as message blocks fed in**
- **Math in compress is modulo 2^{32} so variables always stay at 32 bits**
- **Modulo looses information, making it hard to reverse**

Round:

$Ch(E,F,G) = (E \wedge F) \oplus (\neg E \wedge G)$

$Ma(A,B,C) = (A \wedge B) \oplus (A \wedge C) \oplus (B \wedge C)$

$\Sigma_0(A) = (A \ggg 2) \oplus (A \ggg 13) \oplus (A \ggg 22)$

$\Sigma_1(E) = (E \ggg 6) \oplus (E \ggg 11) \oplus (E \ggg 25)$

$\boxplus$ = addition modulo 32

Output A->B, B->C, … $A = Ch + Ma + \Sigma_0 + \Sigma_1 \% 32$

Repeat 63 times

Adapted from https://www.youtube.com/watch?v=DMtFhACPnTY, see https://youtu.be/f9EbD6iY9zI for detailed explanation of how SHA-2 works

SHA-256 returns a 256-bit (32 byte) digest despite arbitrary length input

SHA-256

Simplified

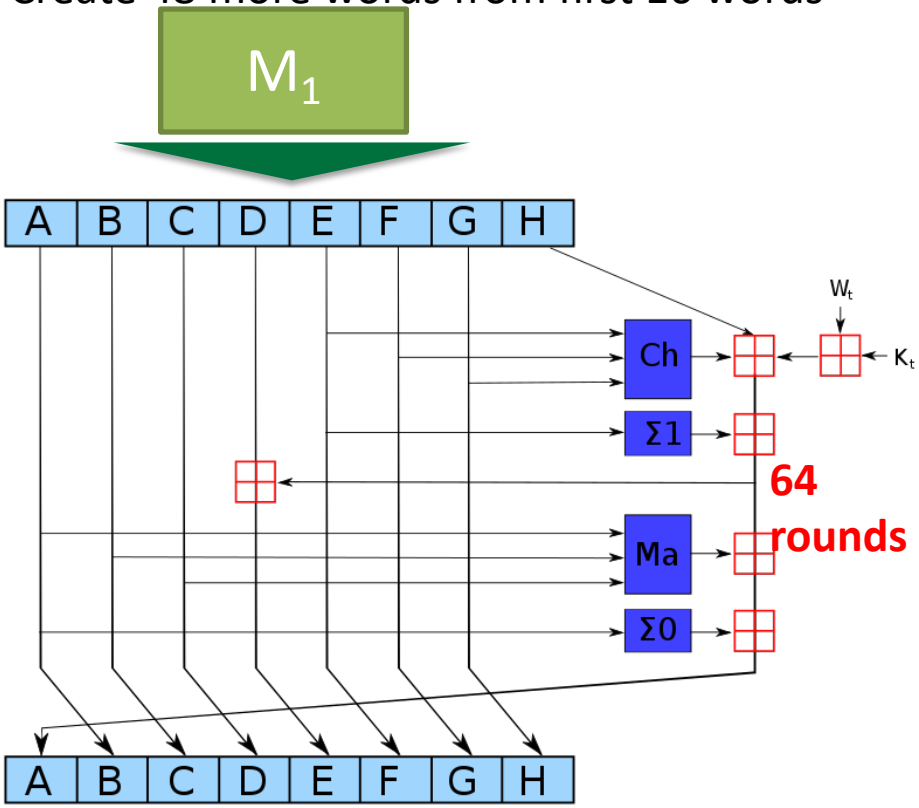
Break message in 512-bit blocks, pad last block

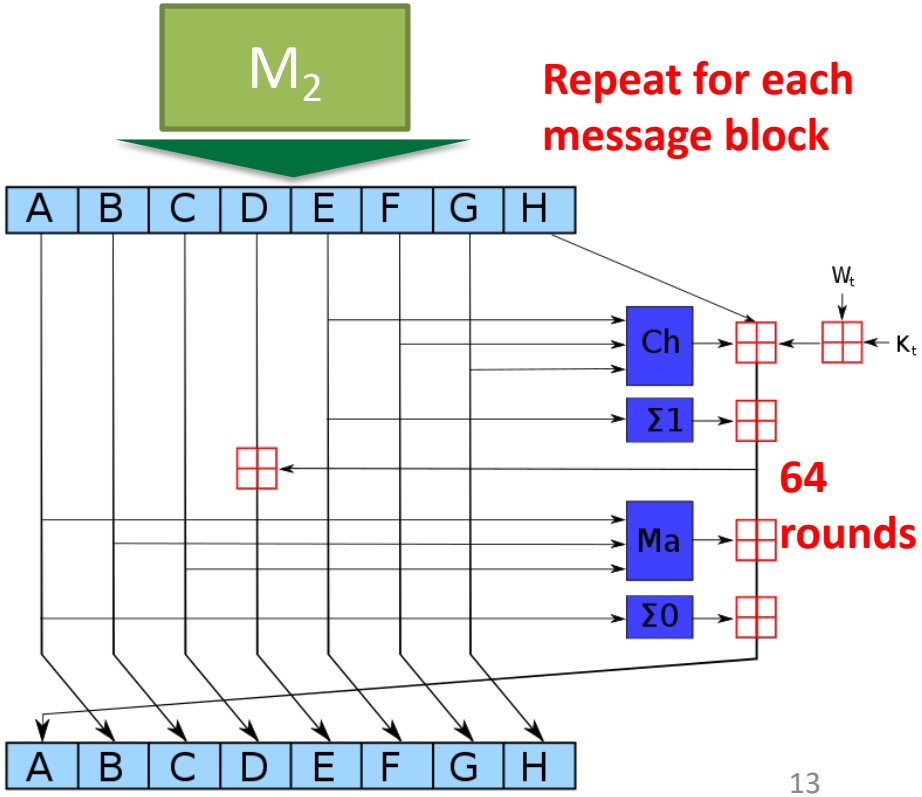
Calculate initial state of eight 32-bit variables based on square root of first 8 prime numbers

Calculate constants K = cube root of first 64 primes

Split input in to 16 words of length 32 bits (512 bits)

Create 48 more words from first 16 words





Adapted from https://www.youtube.com/watch?v=DMtFhACPnTY, see https://youtu.be/f9EbD6iY9zI for detailed explanation of how SHA-2 works

SHA-256 returns a 256-bit (32 byte) digest despite arbitrary length input

SHA-256

Simplified

Break message in 512-bit blocks, pad last block

Calculate initial state of eight 32-bit variables based on square root of first 8 prime numbers

Calculate constants K = cube root of first 64 primes

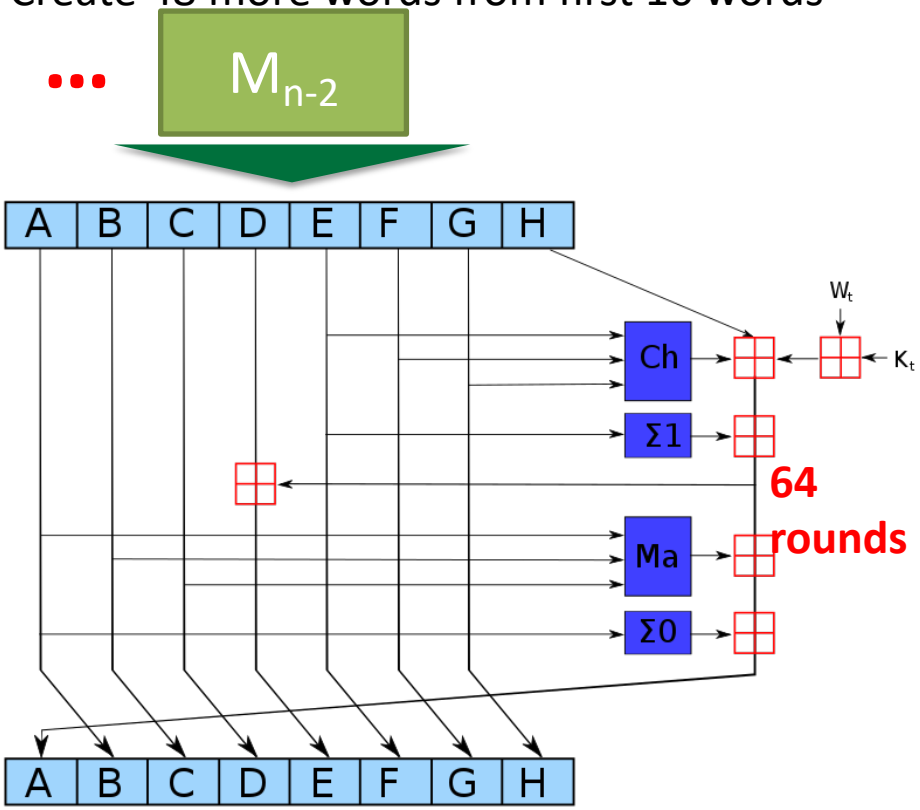
Split input in to 16 words of length 32 bits (512 bits)

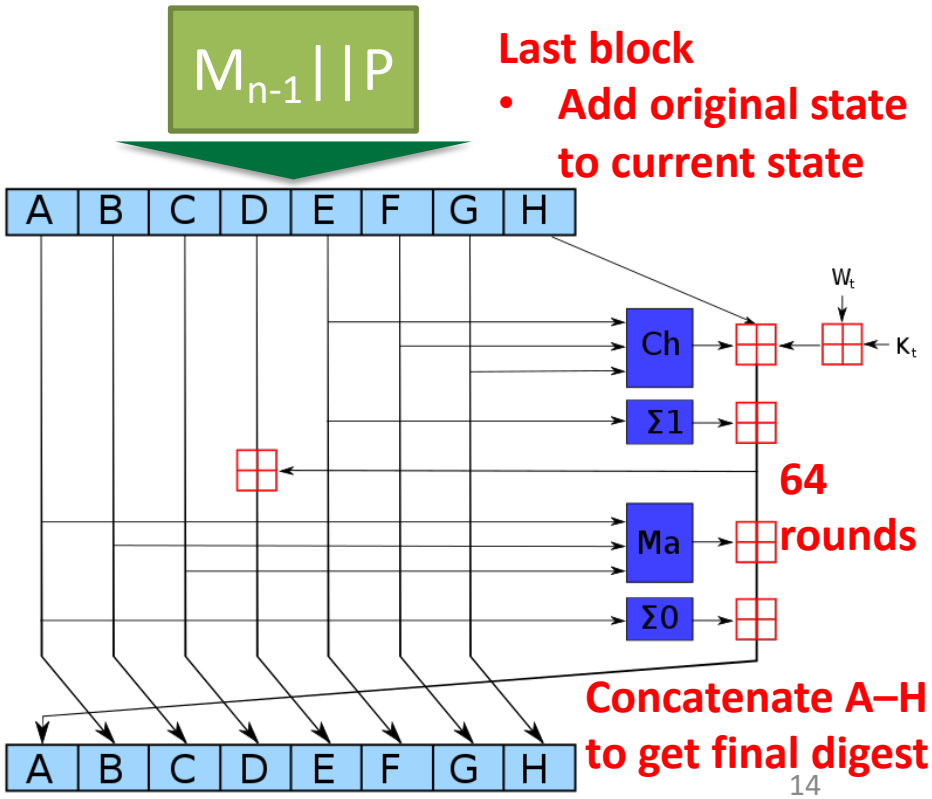
Create 48 more words from first 16 words

Key points: internal state is always 256 bits

State changes as new blocks fed in

Modulo 32 looses information





Adapted from https://www.youtube.com/watch?v=DMtFhACPnTY, see https://youtu.be/f9EbD6iY9zI for detailed explanation of how SHA-2 works

Most Linux distributions come with utilities to compute hashes from the command line

```
#get md5 hash of plain.txt file
#hashes to fixed length of 128 bits (16 bytes)
$ md5sum plain.txt
9db228551f900dc17d6a8059bedc0880  plain.txt

#get SHA-256 hash of plain.txt file
#hashes to fixed length of 256 bits (32 bytes)
$ sha256sum plain.txt
45c946e93dc0509e4546f5489c1cd1083e7088976c7743c059ef70f49b031895  plain.txt

#get SHA-512 hash of plain.txt file
#hashes to fixed length of 512 bits (64 bytes)
$ sha512sum plain.txt
1969715ca7f29edcffaabcafa6aeaf4061a0d0b1bab8352e2d0da38c61437f5093ecfadfccc280e5
599a220da31c58fce7924841a036f82c889182b9551870fd  plain.txt

#can also use openssl
$ openssl dgst -sha512 plain.txt
SHA512(plain.txt)=
1969715ca7f29edcffaabcafa6aeaf4061a0d0b1bab8352e2d0da38c61437f5093ecfadfccc280e5
599a220da31c58fce7924841a036f82c889182b9551870fd
```

You can calculate hashes in Python using hashlib

compute_hash.py

```
try:
    #set up hashlib for SHA512
    m = hashlib.sha512()

    #read data file line by line
    f = open(sys.argv[1],'r')
    for line in f:
        #update after each line
        m.update(line.encode("utf-8"))
    f.close()
    #print hash of file
    print(m.hexdigest())
except:
    traceback.print_exc()
```

To confirm type "sha512sum <filename>" on command line

Note: same result if read line by line or read the whole file and then hash

You can also calculate hashes in C using OpenSSL

compute_hash.c

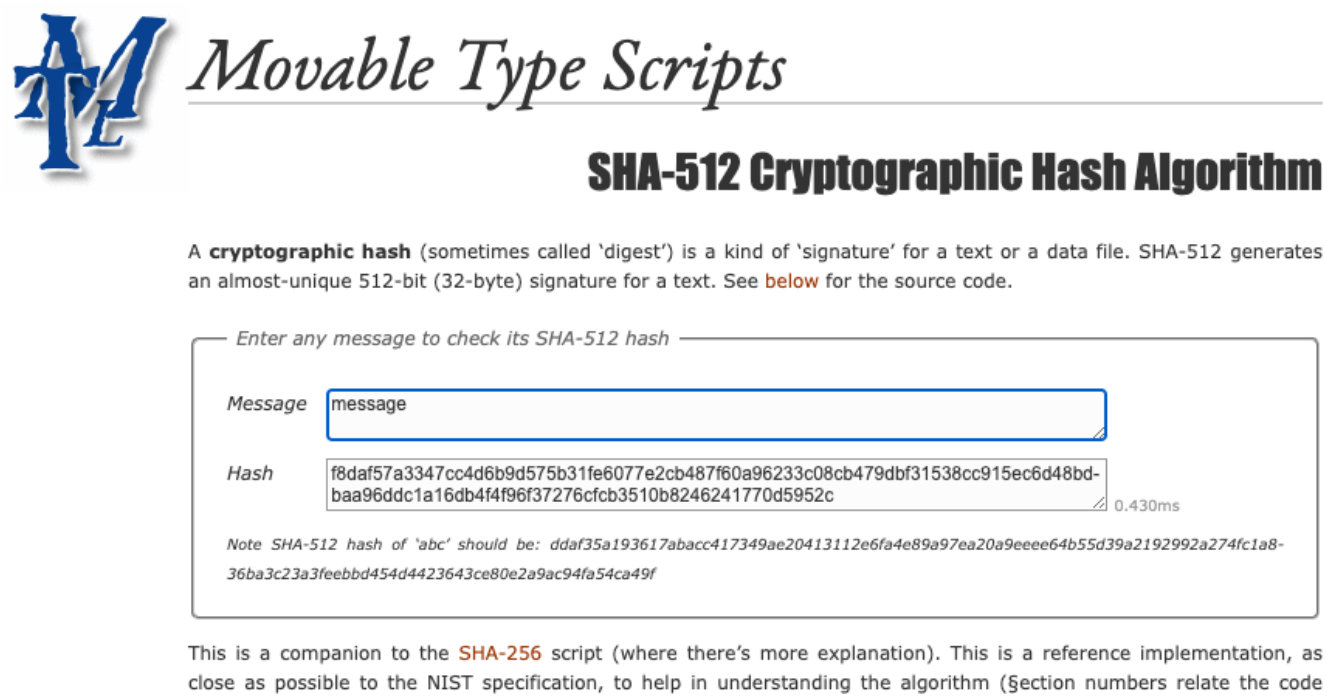
Compile with: gcc compute_hash.c -o compute_hash -lcrypto
Run with: compute_hash plain.txt
(Some code snipped for space on slide)

```
void main(int argc, char *argv[]) {
        SHA512_CTX ctx;
        u_int8_t results[SHA512_DIGEST_LENGTH];
        FILE * fp;

        SHA512_Init(&ctx); //set up for hashing

        fp = fopen(argv[1], "r"); //open file for reading
        if (fp == NULL) exit(EXIT_FAILURE);

        while ((read = getline(&line, &len, fp)) != -1) { //read lines
                SHA512_Update(&ctx, line, strlen(line)); //update digest
        }

        SHA512_Final(results,&ctx); //finalize hashing

        //print results
        for(i=0;i< SHA512_DIGEST_LENGTH;i++) {
                printf("%02x",results[i]);
        }
        printf("\n");
}
```

Returns the same digest as Python and the command line

Some websites can give a hash such as SHA-256 or SHA-2512 on the fly

Website that calculates SHA hashes on the fly
https://www.movable-type.co.uk/scripts/sha512.html



Changing even one bit in the input message results in a different digest

```
#compute SHA-256 of message
$ echo "Hello world!" | sha256sum
0ba904eae8773b70c75333db4de2f3ac45a8ad4ddba1b242f0b3cfc199391dd8

#change message and recompute hash
$ echo "Hallo world!" | sha256sum
bf1adae4567d9fb6b3bfb30cbf4dfdd2503e89a831cf3472c399b39fb9c73289
```

It is *extremely* unlikely hashes of two different inputs will collide

How unlikely you ask?

It is extremely unlikely that hashes of two different inputs will collide

https://www.youtube.com/watch?v=S9JGmA5_unY

Agenda

1. Hashing intro
2. Common hash functions
3. Hashing use cases
4. MACs and attacks

Use case 1: Committing to a secret without revealing it

Committing to a secret

- Our game
- Hash secret and publish hash
- Disclosing hash does not disclose secret – irreversible (one way)
- Once hash is published, cannot change document without being detected – collision resistance

Example

- You've figured out a way to predict the stock market
- You don't want to tell everyone beforehand (or price goes up)
- Hash your predictions (with random nonce) and post them
- Later people can check your predictions were accurate

Adapted from Du, Wenliang. *Computer & Internet Security: A Hands-on Approach*. 2019.

Use case 2: Integrity verification

Integrity verification (e.g., determine nothing has changed, or if it has!)

- Hash the original object
- To see if the object hash been changed, compare the original hash with a hash of the current version of the object

Hash is equivalent to a duplicate copy because if anything changed in OS file, hashes won't match

Example

- Want to know if OS kernel files have been changed by malware
- Could keep a spare copy of each OS file and check when going to run if it matches the spare copy – wasteful/impractical
- Instead, hash OS kernel files and store the 32-byte hash (not a second copy)
- Adversary might change kernel files, inserting malicious code
- Before running code, hash it and compare with stored version
- If no match, do not run it!

Many files on the Internet post a hash of the original
Compare hash of what you downloaded with posted hash to confirm you got an unaltered copy (is this good enough?)

Adapted from Du, Wenliang. *Computer & Internet Security: A Hands-on Approach*. 2019.

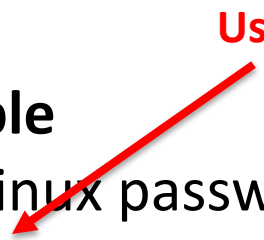
Use case 3: Password verification

Password verification

- Do not store password in plaintext
- Store hashed version of password
- Use salt to defeat dictionary/rainbow tables
- To authenticate: enter password, hash it, compare with stored

Example

- In Linux passwords stored in /etc/shadow
- seed:6wDRrWCQz$IsBXp9.9wz9SGrF.nbihpoN5w.zQx02sht4c TY8qI7YKh00wN/sfYvDeCAcEo2QYzCfpZoaEVJ8sbCT7hkxXY/:17372:0:99999:7:::



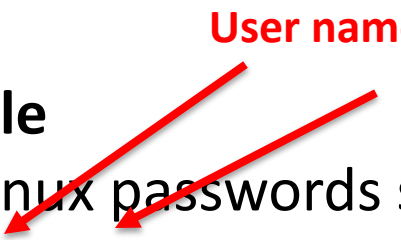
Adapted from Du, Wenliang. *Computer & Internet Security: A Hands-on Approach*. 2019.

Use case 3: Password verification

Password verification

- Do not store password in plaintext
- Store hashed version of password
- Use salt to defeat dictionary/rainbow tables
- To authenticate: enter password, hash it, compare with stored

Example



- In Linux passwords stored in /etc/shadow
- seed:6wDRrWCQz$IsBXp9.9wz9SGrF.nbihpoN5w.zQx02sht4c TY8qI7YKh00wN/sfYvDeCAcEo2QYzCfpZoaEVJ8sbCT7hkxXY/:17372:0:99999:7:::

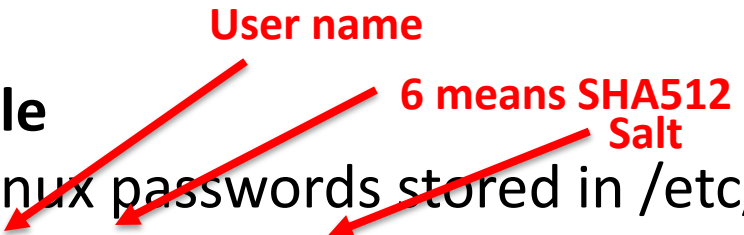
Adapted from Du, Wenliang. *Computer & Internet Security: A Hands-on Approach*. 2019.

Use case 3: Password verification

Password verification

- Do not store password in plaintext
- Store hashed version of password
- Use salt to defeat dictionary/rainbow tables
- To authenticate: enter password, hash it, compare with stored

Example



- In Linux passwords stored in /etc/shadow
- seed:6wDRrWCQz$IsBXp9.9wz9SGrF.nbihpoN5w.zQx02sht4c TY8qI7YKh00wN/sfYvDeCAcEo2QYzCfpZoaEVJ8sbCT7hkxXY/:17372:0:99999:7:::

Adapted from Du, Wenliang. *Computer & Internet Security: A Hands-on Approach*. 2019.

Use case 3: Password verification

Password verification

- Do not store password in plaintext
- Store hashed version of password
- Use salt to defeat dictionary/rainbow tables
- To authenticate: enter password, hash it, compare with stored

Example

- In Linux passwords stored in /etc/shadow
- seed:6wDRrWCQz$IsBXp9.9wz9SGrF.nbihpoN5w.zQx02sht4cTY8qI7YKh00wN/sfYvDeCAcEo2QYzCfpZoaEVJ8sbCT7hkxXY/:17372:0:99999:7:::

Adapted from Du, Wenliang. *Computer & Internet Security: A Hands-on Approach*. 2019.

Use case 3: Password verification

Password verification

- Do not store password in plaintext
- Store hashed version of password
- Use salt to defeat dictionary/rainbow tables
- To authenticate: enter password, hash it, compare with stored

Example

- In Linux passwords stored in /etc/shadow
- seed:6wDRrWCQz$IsBXp9.9wz9SGrF.nbihpoN5w.zQx02sht4cTY8qI7YKh00wN/sfYvDeCAcEo2QYzCfpZoaEVJ8sbCT7hkxXY/:17372:0:99999:7:::


```
$ sudo cat /etc/shadow #list all passwords on system (requires sudo)
…
seed:$6$wDRrWCQz$IsBXp9.9wz9SGrF.nbihpoN5w.zQx02sht4cTY8qI7YKh00wN/
…
$ python3 make_linux_password_hash.py dees '$6$wDRrWCQz'
$6$wDRrWCQz$IsBXp9.9wz9SGrF.nbihpoN5w.zQx02sht4cTY8qI7YKh00wN/
```

Does 5,000 rounds of SHA512 to make password slow to compute

Use case 4: Trusted timestamping

RFC 3161

Trusted timestamping

- Want to be able to prove a document existed at a point in time (called Long Term Validation – LTV)
- Also want non-repudiation (cannot later deny something, repudiate means to deny)

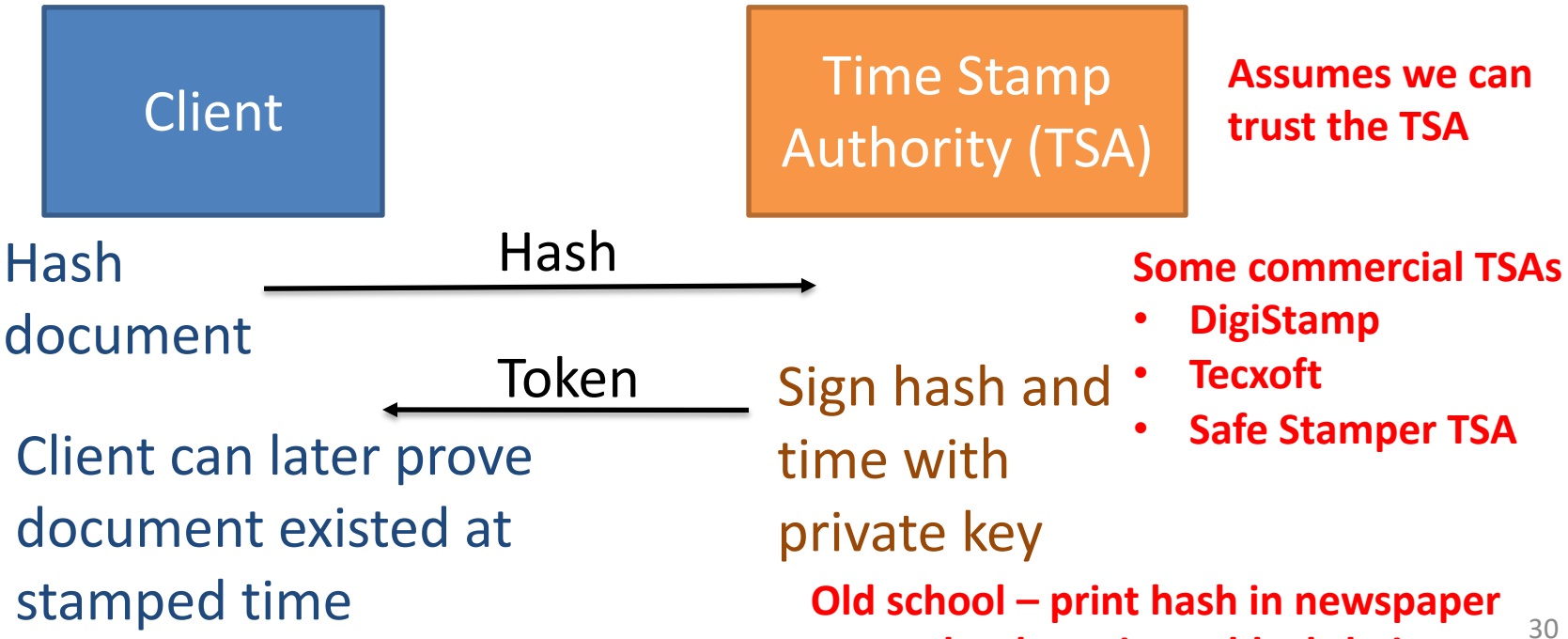
Adapted from: https://www.forensicnotes.com/timestamping-authority-tsa/

Use case 4: Trusted timestamping

RFC 3161

Trusted timestamping

- Want to be able to prove a document existed at a point in time (called Long Term Validation – LTV)
- Also want non-repudiation (cannot later deny something, repudiate means to deny)



Adapted from: https://www.forensicnotes.com/timestamping-authority-tsa/

Discussion

Scenario:

- You are pitching a start up idea to a sketchy venture capitalist
- You'd like to be able to prove you have produced your business plan *before* you meet with them
- What can you do?

Agenda

1. Hashing intro
2. Common hash functions
3. Hashing use cases
4. MACs and attacks

We need to guard against a message that has been changed or forged

Alice sends message to Bob

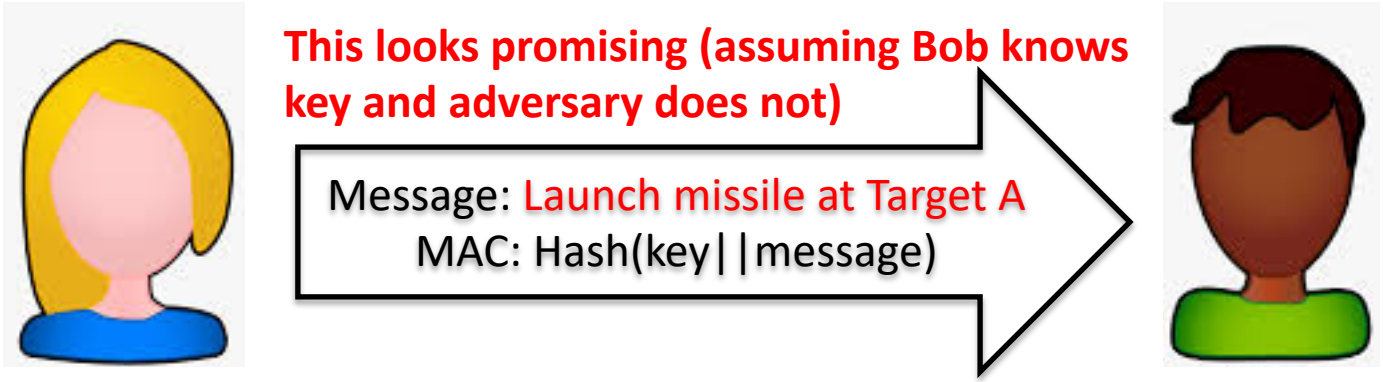
How does Bob know if the message has been altered?

MACs are used to detect if a message has been changed or forged but be careful!

Message Authentication Code (MAC)

Not secure!

Changing the message fed into a hash function by even one bit changes output



Assume Alice and Bob share a secret key

Alice could *hash(key || message)* and send hash with message (called a Message Authentication Code, MAC, or a tag)

How does Bob know if the message has been altered?

Bob knows Key and can hash key||message and compare with MAC sent by Alice

MACs are used to detect if a message has been changed or forged but be careful!

Message Authentication Code (MAC)

Not secure!

Assume Alice and Bob share a secret key

Alice could *hash(key || message)* and send hash with message (called a Message Authentication Code, MAC, or a tag)

How does Bob know if the message has been altered?

echo -n **”theKey:Launch a missile at Target A" | sha256sum**
5a97c8de6b858a3bb145b62661bb511eb7c77aeb1ef0cf86d585d8b81ecef2e7

MACs are used to detect if a message has been changed or forged but be careful!

Message Authentication Code (MAC)

Not secure!

Assume Alice and Bob share a secret key

Alice could *hash(key || message)* and send hash with message (called a Message Authentication Code, MAC, or a tag)

How does Bob know if the message has been altered?

Bob knows Key and can hash key||message and compare with MAC

echo -n **"theKey:Launch a missile at Target A" | sha256sum**
5a97c8de6b858a3bb145b62661bb511eb7c77aeb1ef0cf86d585d8b81ecef2e7

MACs are used to detect if a message has been changed or forged but be careful!

Message Authentication Code (MAC)

Not secure!

Alice **Bob**

Assume Alice and Bob share a secret key

Bob computes h(key||message)
h(theKey:Launch a missile at Target A)

MAC and hash match, so message has not been changed

How does Bob know if the message has been altered?

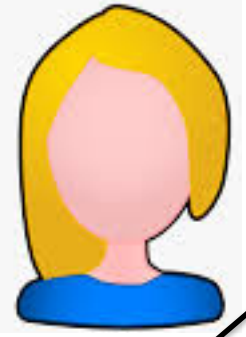
Bob knows Key and can hash key||message and compare with MAC

echo -n **"theKey:Launch a missile at Target A" | sha256sum**
5a97c8de6b858a3bb145b62661bb511eb7c77aeb1ef0cf86d585d8b81ecef2e7

MACs are used to detect if a message has been changed or forged but be careful!

Message Authentication Code (MAC)

Not secure!



Alice

Bob hashes message with the correct key
Hash does not match MAC, message alerted!

hash(theKey:message) ≠ hash(wrongKey:message)

Bob

Message: Launch missile at **Target A**
MAC: **5a97c8<snip>8581ecef2e7**

Eve changes the message and guesses the key, but gets it wrong, sends result to Bob

Eve intercepts message

Eve

Eve does not know the key

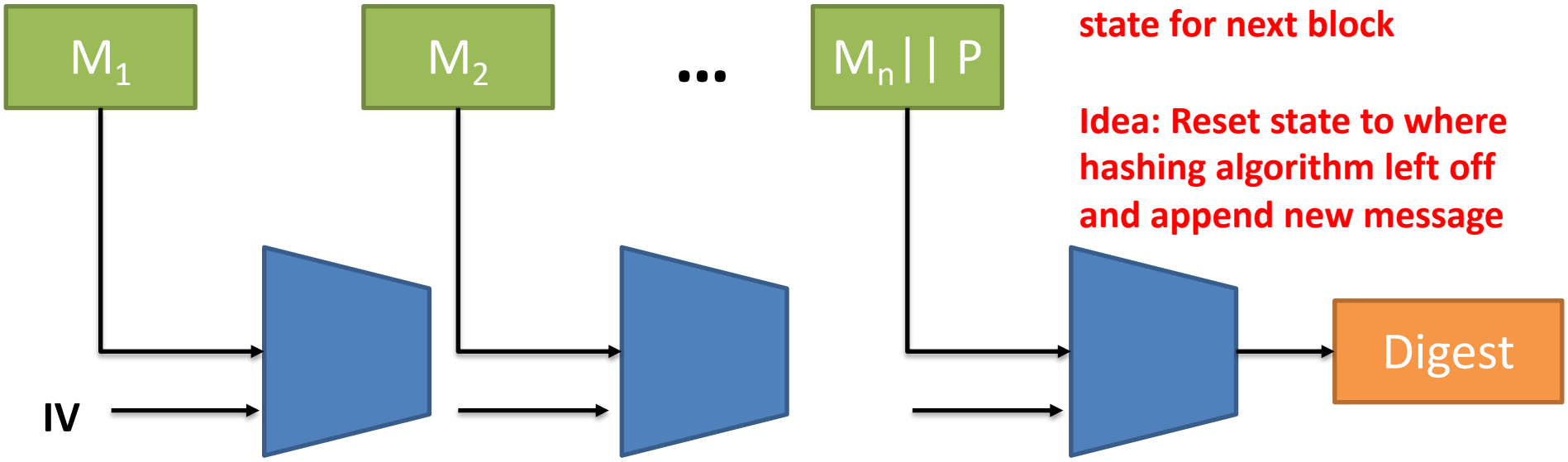
Message: Launch missile at **Target B**
MAC: **c64edd247<snip>6618ff25b81**

Bob knows message is invalid

Sounds good right?

Hash length extension attacks exploit Merkle-Damgard construction

Hash length extension attack



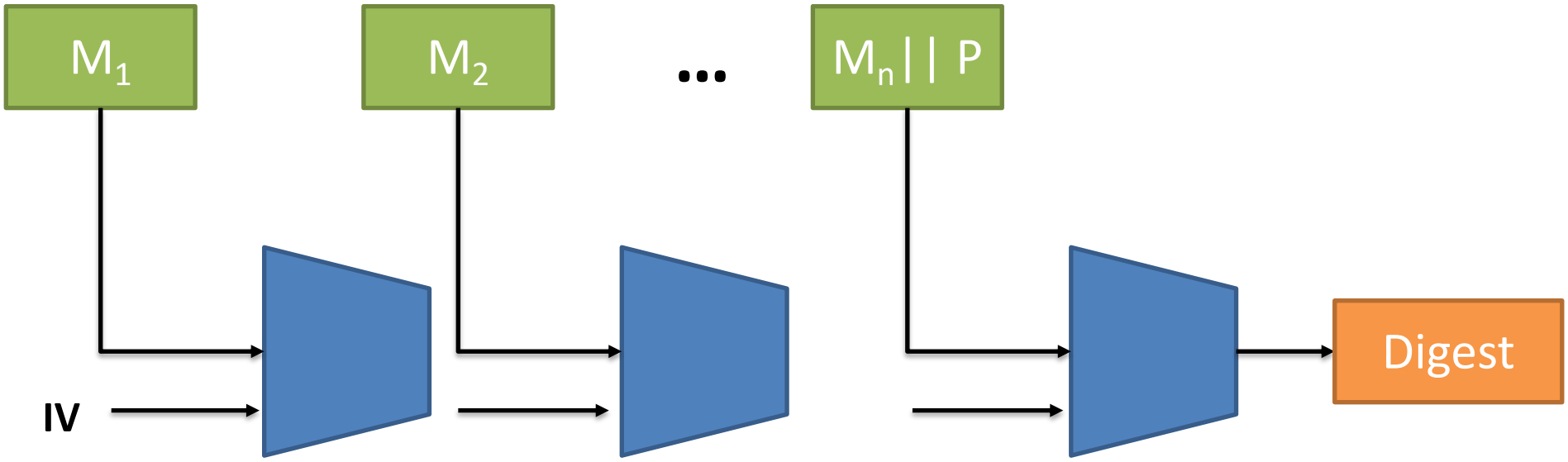
Because a block depends on the prior block

- If adversary intercepts message with MAC and can guess key length
- Adversary can add more text on to the end and still get a valid MAC, even though they do not know the key

Adapted from Du, Wenliang. *Computer & Internet Security: A Hands-on Approach*. 2019.

Hash length extension attacks exploit Merkle-Damgard construction

Hash length extension attack



theKey:Launch a missile at Target A \x80\x00 \x00 \x00 \x00 \x00 \x00 \x00 \x00 \x00 \x00 \x00 \x00 \x00 \x00 \x00 \x00 \x00 \x00 \x00 \x00 \x00 \x00 \x00 \x00 \x00 \x00 \x00 \x00 \x00 \x00 \x00 \x00 \x01 \x18

SHA2 uses 512-bit (64-byte) blocks with padding on the last block

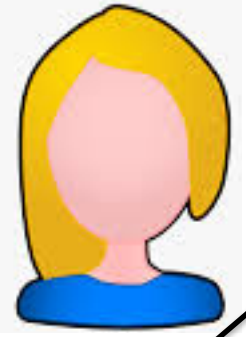
- Pad with 1 followed by zeros,
- Pad ends with size of key:message in hex
  - Len("theKey:Launch missile at Target A") = 35
  - End with 35 bytes * 8 = 280 (decimal) = 0x118 (hex)

This is the value that is actually hashed

MACs are used to detect if a message has been changed or forged but be careful!

Message Authentication Code (MAC)

Not secure!



Alice

Bob hashes theKey:Launch a missile at Target A
\x80\x00 \x00 \x00 \x00 \x00 \x00 \x00 \x00
\x00 \x00 \x00 \x00 \x00 \x00 \x00 \x00 \x00
\x00 \x00 \x00 \x00 \x00 \x00 \x00 \x00 \x00
\x01 \x18 and at Hanover

Bob

Message: Launch missile at Target A
MAC: 5a97c8<snip>8581ecef2e7

Eve resets her SHA256 context using MAC
Extends message adding " and at Hanover"
Does not know secret key, but sends altered message and MAC to Bob

Eve intercepts message

Eve

Eve does not know the key

Message: Launch ... and at Hanover
MAC: bebb350f2<snip>ecd56a0792

Hash checks out!

Fire two missiles!

Adversary resets SHA256 context to where it left off, then adds new message

sha256_length_extension.c

```
void main(int argc, const char *argv[]) {
  int i;
  unsigned char buffer[SHA256_DIGEST_LENTH];
  SHA256_CTX c;

  SHA256_Init(&c);
  for (i =0; i<64; i++)  SHA256_Update(&c, "*", 1);

  c.h[0] = htole32(0x5a97c8de);
  c.h[1] = htole32(0x6b858a3b);
  c.h[2] = htole32(0xb145b626);
  c.h[3] = htole32(0x61bb511e);
  c.h[4] = htole32(0xb7c77aeb);
  c.h[5] = htole32(0x1ef0cf86);
  c.h[6] = htole32(0xd585d8b8);
  c.h[7] = htole32(0x1ecef2e7);

  // Append the additional message
  SHA256_Update(&c, " and at Hanover", 15);
  SHA256_Final(buffer, &c);
  for (i = 0; i < 32; i++) {
     printf("%02x", buffer[i]);
  }
  printf("\n");
}
```

Legitimate message: "Launch a missile at Target A"
Legitimate MAC: 5a97c8de6b858a3bb145b626
61bb511eb7c77aeb1ef0cf86d585d8b81ecef2e7

Eve sees message and MAC

Set up SHA256 context

Reset context to where hashing left off using MAC

Add new message (this is the message length extension)

Print new MAC

Note: padding not printable characters!

Send Bob

- **"Launch a missile at Target A <padding> and at Hanover"**
- **New MAC**

Adapted from Du, Wenliang. *Computer & Internet Security: A Hands-on Approach*. 2019.

MAC checks out at the receiver, even though adversary does not know secret

sha256_padding.c

```
int main(int argc, const char *argv[]) {
  SHA256_CTX c;
  unsigned char buffer[SHA256_DIGEST_LENGTH];
  int i;

  SHA256_Init(&c);
  SHA256_Update(&c,
    "theKey:Launch a missile at Target A"
    "\x80\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00"
    "\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x01\x18"
    " and at Hanover",
    64+17);
  SHA256_Final(buffer, &c);

  printf("New MAC\n");
  for (int i = 0; i < 32; i++) {
    printf("%02x", buffer[i]);
  }
  printf("\n");
  return 0;
}
```

Message: “Launch a missile at Target A <padding> and at Hanover”
New MAC: bebb350f2613abff0520fa9754cc4 4cb58d36e3ec17fbd0092a2e9ecd56a0792

Set up SHA256 context

Hash with original message, padding, and extension

MAC matches the one sent

Two missiles launched!

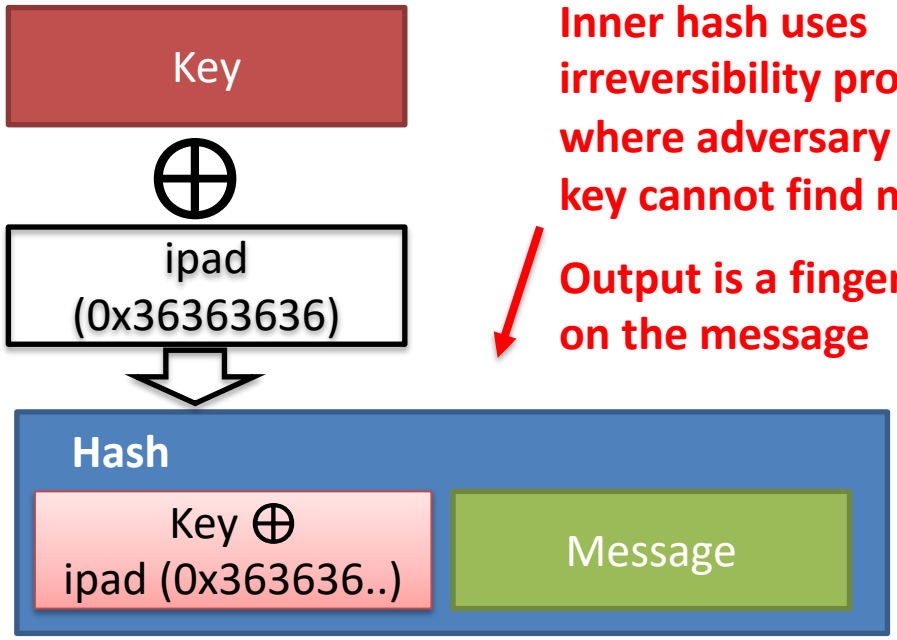
Adapted from Du, Wenliang. *Computer & Internet Security: A Hands-on Approach*. 2019.

A better solution is to use a Keyed-Hash Message Authentication Code (HMAC)

HMAC (K,m) = H((K ⊕ opad) || H((K ⊕ ipad) || m))

Requires secret key and two hashes using:

- K = key
- opad = outer pad = 0x5c5c5c5c
- ipad = inner pad = 0x36363636



Inner hash uses irreversibility property where adversary without key cannot find message

Output is a fingerprint on the message

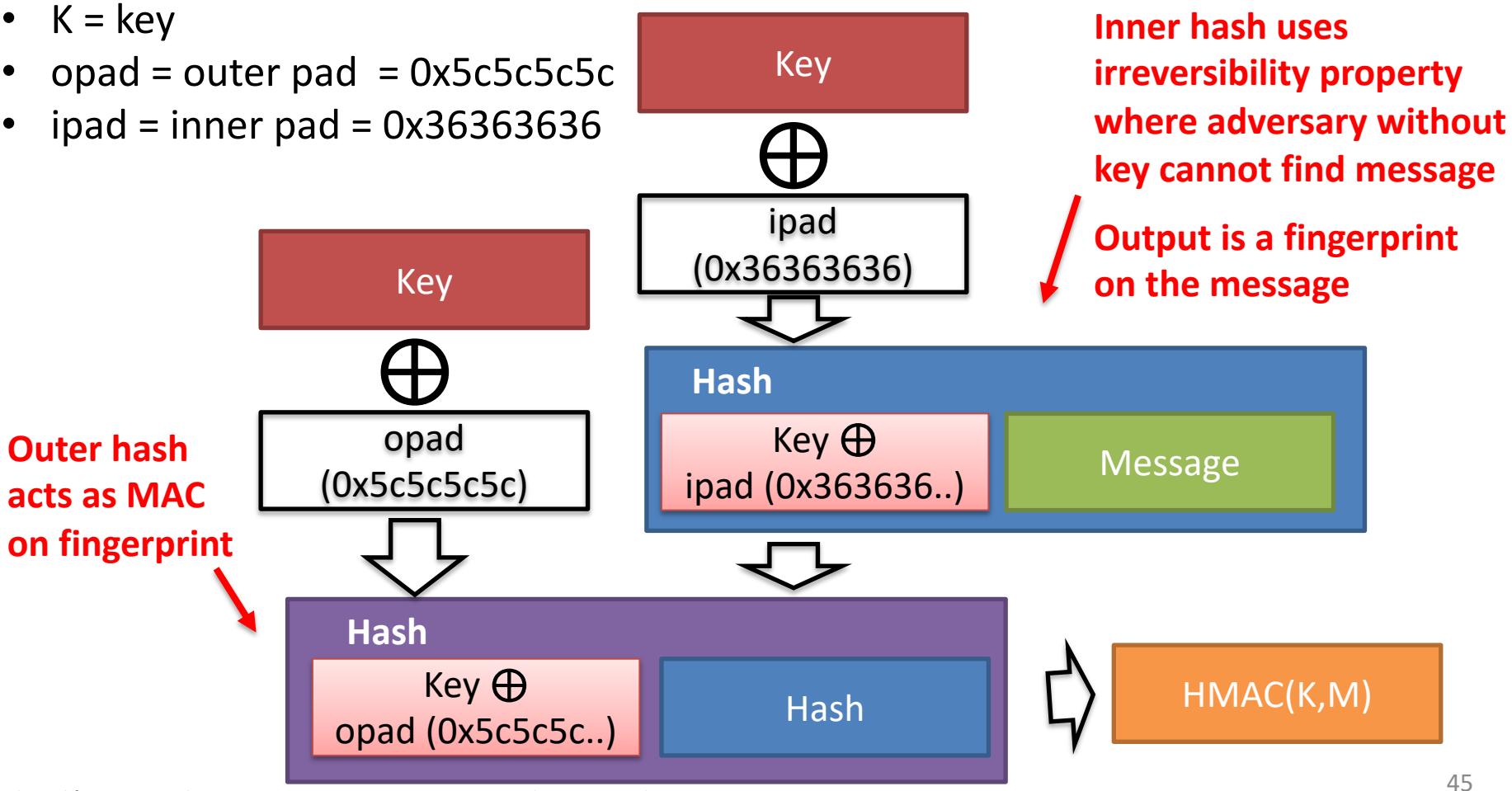
Adapted from Du, Wenliang. *Computer & Internet Security: A Hands-on Approach*. 2019.
and https://crypto.stackexchange.com/questions/12680/how-does-the-secret-key-in-an-hmac-prevent-modification-of-the-hmac

A better solution is to use a Keyed-Hash Message Authentication Code (HMAC)

HMAC (K,m) = H((K ⊕ opad) || H((K ⊕ ipad) || m))

Requires secret key and two hashes using:

- K = key
- opad = outer pad = 0x5c5c5c5c
- ipad = inner pad = 0x36363636



Adapted from Du, Wenliang. *Computer & Internet Security: A Hands-on Approach*. 2019.
and https://crypto.stackexchange.com/questions/12680/how-does-the-secret-key-in-an-hmac-prevent-modification-of-the-hmac