Crypto II

Modern Cryptography

Administrivia

- This is our *last lecture*
- Next week will be a working class
    - Come in and work on homework, because right now the majority of the class has fewer than 2 problems solved

Historical Perspective

- Around 1940, Enigma was cracked
  - Huge effort by Allied forces
  - One of the first uses of computers
- "Now what do we do?"
  - No real understanding on what makes something secure
  - How can we prevent this from happening again?

A Good Idea?

- Eventually computational complexity became a thing
  - Some problems seem *really hard* to solve on computers
  - SAT, knapsack problem, travelling salesman, etc.
- Reduce security of cryptosystem to solving a known "hard" problem!
  - Seems like an excellent idea!

Security from Complexity

- Turns out this idea was really bad
  - NP-hard problems *still* are hard, but not uniformly so
  - Most random instances of NP-hard problems turn out to be really easy!
  - If you want *hard* instances, you need to do a lot of work!
- Very interesting, but now what do we do?

Security from Magic

- Nowadays we rely on magic
    - Number theory problems that seem hard, but no one really knows about
        - Factoring numbers, discrete logarithms, non-linear systems
    - Magic systems that someone creates, and even after trying a while, no one can break
        - See: AES, SHA1-3, anything symmetric

Modern Cryptography

- Three basic areas of modern crypto:
    - Asymmetric encryption
        - Alice can encrypt messages to Bob with "public" parameters, only Bob can read messages
    - Symmetric encryption
        - Alice and Bob share the same secret, use this to encode and decode messages to each other
    - Hash functions
        - Secure way to represent long strings, used in a variety of ways
- Each area broken down into cryptographic "primitives" which are used in other areas

Asymmetric Encryption

- Based on number theory magic (most of which you can understand!)
- Important algorithms:
  - **RSA**, **Diffie-Hellman**, ECC, El-Gamal
- RSA
  - Magic algorithm used for asymmetric encryption, signatures, authentication
- Diffie-Hellman
  - Easier algorithm used *only* to establish shared secrets (to then use with symmetric cryptography)

Diffie-Hellman

- Alice and Bob establish a shared secret
- All communication public
- Based on the hardness of the *discrete logarithm* problem
  - given ($y = g^x \bmod n$), g, and n: recover x

Diffie-Hellman

Alice	Bob
choose random g, p (prime)	
	[receive g, p]
choose random, secret x	choose random, secret y
send $X = g^x \bmod p$	send $Y = g^y \bmod p$
receive Y	receive X
<u>secret = $Y^x \bmod p$</u>	<u>secret = $X^y \bmod p$</u>

Diffie-Hellman

- Eve sees:
    - g, p
    - $X = g^x \bmod p$, $Y = g^y \bmod p$
    - without solving discrete logarithm, can't break it!
- Questions so far?

RSA

- Rivest, Shamir, and Adleman
    - Invented in secret years earlier by GCHQ
- One of the most important algorithms of all time
- Used by just about everything
- Surprisingly tricky to get correct

RSA

<u>Number theoretic</u>

$n = p * q$ (both primes)

$e = 0x10001 = 65537$

$d = e^{-1}$ modulo $\phi(n) = (p-1)(q-1)$

number theory magic:

$x^{(a*b)} \bmod n = x^{(a*b \bmod \phi(n))} \bmod n$

$e*d \bmod \phi(n) = 1$

RSA

- Alice generates:
  - p and q primes, $n=p*q$
  - $d = e^{-1} \mod \phi(n)$
  - Secret: p,q,d
  - Public: e,n
- To encrypt a message:
  - Send Alice $m^e \mod n$
  - Alice decrypts by raising to the d
  - No one else can read the messages!

RSA

- RSA gets tricky for a few reasons
  - First: how to generate p and q
    - while p is not prime, p = rand(0,2**1024) ?
      - no
    - while 2**1024+p not prime, p = rand(0,2**1023) ?
      - not quite
    - very tough to get right, source of a lot of problems!
    - also need to make sure your random number generator is *good*

RSA

- let n be $\sim 2^{2048}$
- let e be 3
- m = "hello" = 448378203247
- m^3 mod n = 90143305010218464651239068244550223
  - This is less than n! We can just take the cube root!
  - How is this RSA thing secure at all?

RSA

- Padding becomes *very important* for RSA
  - ensure that when raised to a power, message gets "sufficiently garbled"
  - common scheme: append N bytes, each with value N (PKCS#5/7)
    - \x07\x07\x07\x07\x07\x07\x07
  - better scheme: OAEP "optimal asymmetric encryption padding"
    - Complicated, but sort of proven to be really secure!

RSA

- Message signing
  - Alice wants to verify that she wrote a message
  - Raises message to *d* mod n and publishes it as S
  - Anyone can verify that S^e mod n is the original message, as e and n are public!
- How do you verify that the public key belongs to Alice?
  - "Web of trust" you sign public keys of people you can verify in person
  - If you can find a path of people you trust which verified keys, you can be reasonably sure a key belongs to someone!

RSA

- Some common attacks against RSA signatures
    - Again, rely heavily on improper RSA padding!
    - Bleichenbacher attack when e=3, by simply creating a message whose result will be a perfect cube
- Overall secure and widely used for lots of applications

RSA

- Any questions up to this point?

Symmetric Encryption

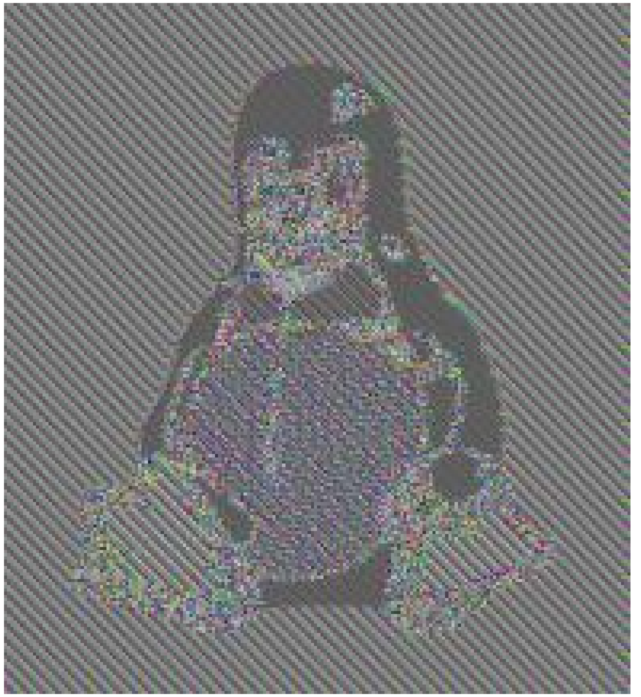
- Alice and Bob get to share a secret (eg by DH) so now things are easier!
- Use "block ciphers"
  - Take in fixed "blocks" of data, output "blocks" of output
- Input and output sizes the same? Model encryption as a *pseudo random permutation*
  - E(key,message) randomly selects one $2^{|\text{block size}|}$ output
  - D(key,message) just inverts the permutation!

Symmetric Encryption

- What do you do with a PRP?
  - I have a bunch of data to encrypt, not a single 128 bit block!
  - Pad data to a multiple of block size (eg with PKCS#5)
- Block cipher "modes of operation"
  - Take not so useful block ciphers, turn them into something better!

Modes of Operation

- Electronic codebook mode
  - Break up message into block sized chunks
  - Encrypt each one!
  - Why does this suck?



Modes of Operation

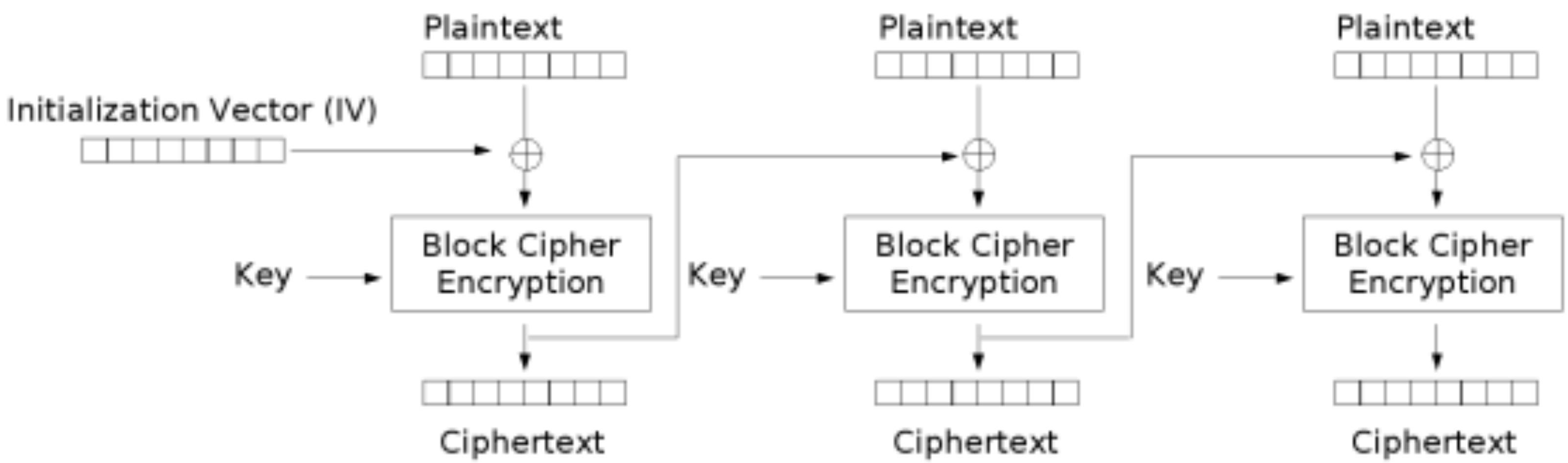
- Counter mode
  - Break up message into block sized chunks
  - Start a counter at 0
  - For each chunk, encrypt counter value, xor result with chunk
  - Increment counter and repeat!
  - We've made a secure one time pad!
  - Problems?

Modes of Operation

- Counter mode
    - Counter mode is actually *very* secure as described!
    - Problem is *everyone* messes up the counter, and resets it at some point
    - Despite simplicity and security, rarely used because everyone will mess it up eventually

Modes of Operation

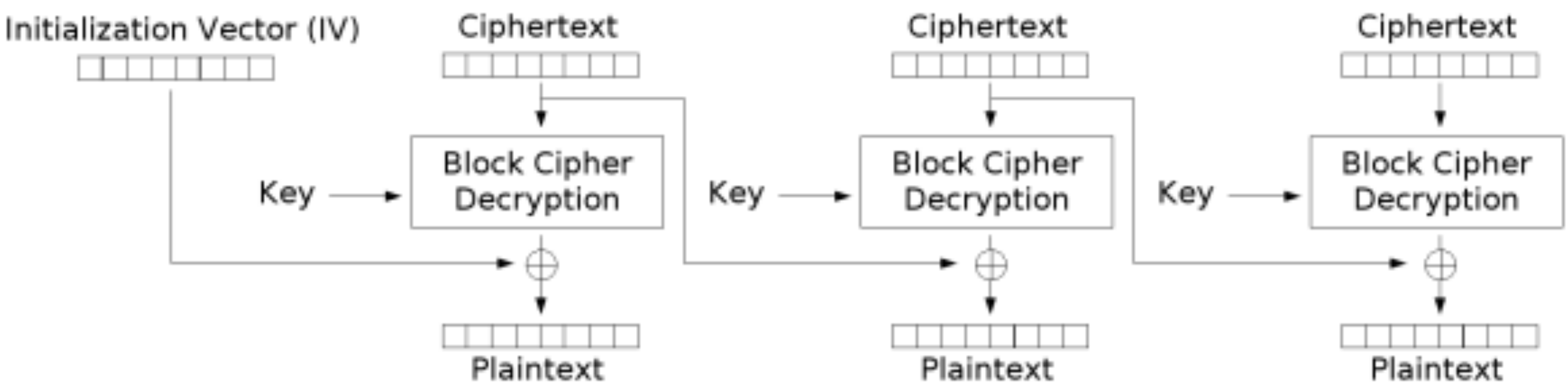
- Cipher block chaining



Cipher Block Chaining (CBC) mode encryption

Modes of Operation

- Cipher block chaining



Cipher Block Chaining (CBC) mode decryption

Modes of Operation

- CBC is used very widely in practice
- It can be parallelized, it encrypts multiple blocks to different things (unlike ECB)
- It is relatively simple to implement
- Some interesting (non CBC-specific) attacks...

Padding Oracle

- Imagine code which functions as follows:
  - Attempt to decrypt message
  - If able to decrypt, and padding (PKCS7) is wrong return BAD_PADDING
  - If able to decrypt, padding right, and message doesn't make sense return BAD_MESSAGE

Padding Oracle

- We call this a "padding oracle" because it can answer "yes" or "no" as to whether or not our padding is correct
- This can be used to encrypt and decrypt some messages for CBC!

Padding Oracle

`aaaabbbbccccdddd` `eeeeffffgggghhhh`

[block 1] [block 2]

gives BAD_PADDING

Padding Oracle

```
aaaabbbbccccdddd  eeeeffffgggghhhh
```

[block 1] [block 2]

change last byte of first block until we get BAD_MESSAGE

Padding Oracle

`aaaabbbbccccddd\x05 eeeeffffgggghhhh`

[block 1] [block 2]

change last byte of first block until we get BAD_MESSAGE

Padding Oracle

`aaaabbbbccccddd\x05 eeeeffffgggghhhh`

[block 1] [block 2]

change last byte of first block until we get BAD_MESSAGE

That means our last byte encodes to \x01 with PKCS padding!

Padding Oracle

`aaaabbbbccccddd\x05 eeeeffffgggghhhh`

[block 1] [block 2]

change last byte of first block until we get BAD_MESSAGE

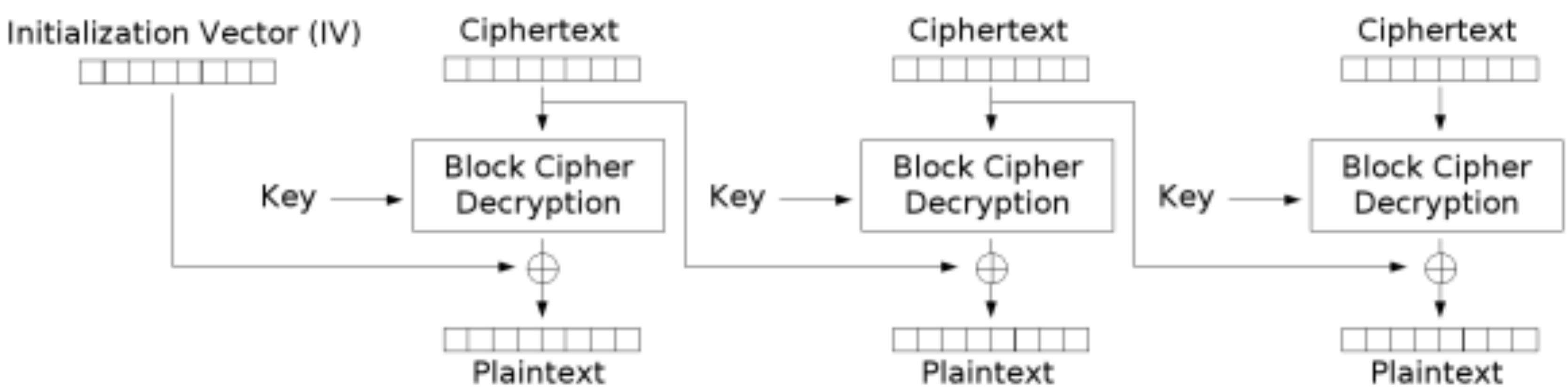
That means our last byte encodes to \x01 with PKCS padding!

xor with \x01 xor \x02, and now modify second to last byte

Padding Oracle

`\x48\x7a\x57\x3b\x91\x0f\xf5\xb5\xa5\x9e\x69\x18\x05\x56\xb3\x05`

- Eventually get some 16 random looking bytes
- These act as an xor key for the following block!



Cipher Block Chaining (CBC) mode decryption

Modes of Operation

- What happened? Was it CBC's fault?
  - Not really, this is just what happens when crypto is done poorly
  - Assumptions were made when making CBC that were violated here
  - It turns out making crypto correct is very difficult

Other crypto subtlety

- When encrypting, should you compress *before* or *after* the encryption step?
  - Encrypting should randomize input, making compression impossible, so compress first
  - What if an untrusted user can add data to your datastream?

CRIME

- Attack on SSL cookies
  - malicious javascript injects data into the SSL session
  - observes how the new data affects the size of the *compressed* transmitted stream
  - "secret secret" compresses better than "random secret"

Symmetric Encryption

- Any questions?

Hash Functions

- Very similar to functions you know for hash tables, but with certain properties:
  1. **Efficient to calculate**
  2. **Preimage resistance:** given a hash, it is difficult to find m such that HASH(m) = h
  3. **Second preimage resistance:** given m1, it is difficult to find a different m2 such that HASH(m1) = HASH(m2)
  4. **Collision resistance:** it is difficult to find any distinct m1, m2 such that HASH(m1) = HASH(m2)

Hash Functions

- Modeled as Pseudo Random Functions
  - Random number generator takes in a "message" and outputs a "hash"
  - Input unbounded, but output is fixed size
- Let's look at collision resistance!

Collision Resistance

- For hash with size N, the best security against collisions is $\sqrt{N}$
  - Birthday paradox: probability ANY two people in a room share a birthday goes like $1-e^{(-n^2)}$
  - Just generate a ton of hashes, and store them!
  - MD5 outputs 128 bit hashes so $2^{64}*128$ bits of hash
    - 256 *million* terabytes worth of hashes

Preimage Resistance

- Even harder to defeat preimage resistance
- Brute force takes N-1 bits on average...
- Even most "broken" hash functions secure against preimage resistance

Hash uses

- Integrity check:
  - Alice sends Bob a 10GB file (maybe encrypted)
  - Bob wants to verify he received the file without error
  - Bob could send the whole file back again, but that is slow and annoying
  - Bob can send HASH(file), and Alice can quickly verify!

Hash uses

- Password verification
  - Alice wants to authenticate to Bob, but doesn't want Bob to store her password
  - Bob stores HASH(*password*)
  - Alice (securely!) sends Bob *password*, and he can quickly verify, without ever storing *password*
  - What is wrong with this?

Hash uses

- Password verification
  - An attacker can hash and store all sorts of passwords into a *rainbow table*
  - Finding Alice's password as easy as a DB lookup!
- Salted passwords
  - Don't store HASH(password), store `RANDOM_STRING||HASH(password||RANDOM_STRING)`
  - This isn't much better at all!
  - Hash functions first property is speed of calculation!
  - An attacker can easily calculate 10s of billions of hashes a second, and brute force passwords!

Hash uses

- Lamport password hash
  - Very cool authentication technique
  - Alice sends $HASH^{(n)}$(password) to Bob
  - Next time she wants to log in, Alice sends $HASH^{(n-1)}$(password) to Bob
  - Bob hashes this result, and gets previously stored result, verifying Alice's identity!

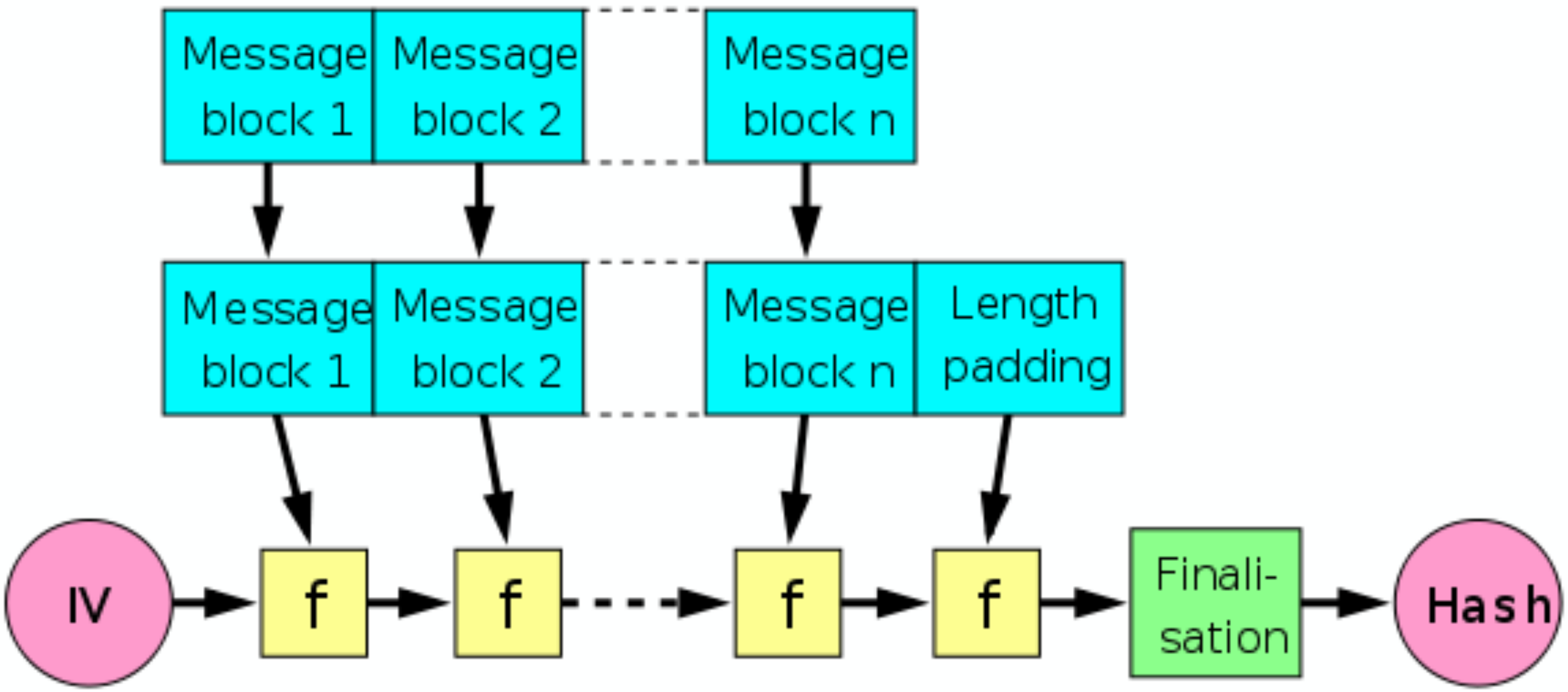
Hash uses

- Message authentication code (MAC)
  - Alice wants Eve to hold onto a large file for a while, but Eve can't be trusted
  - Alice generates a short SECRET, which she stores securely
  - Gives Eve `file||HASH(SECRET||file)`
  - After retrieving the file from Eve, verify the `HASH(SECRET||file)` matches the expected value!

Length Extension

- This works great for Pseudo Random Functions, and is provably secure!
  - Turns out all hash functions aren't so great :(
  - For example: MD5, SHA1, SHA2.....
- How do you build a hash function?
  - "Merkle Darmgard"

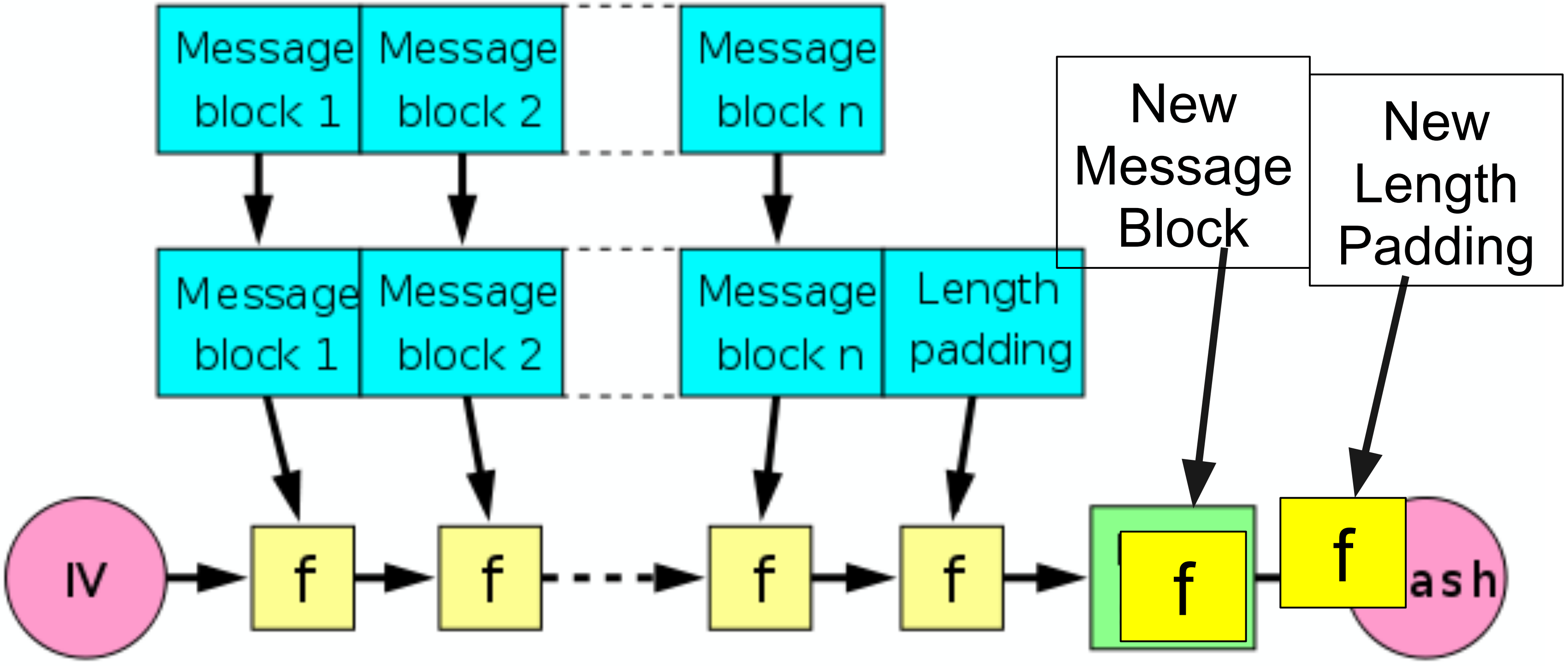
Length Extension



Length Extension

- "Finalization" step is a no-op for most hash functions
  - That means the output of the hash function is the last internal Merkle Darmgard state!
  - "Pick up" where the hash function "left off"

Length Extension



Length Extension

- Given `HASH(secret||message)`, able to construct
  `HASH(secret||message||PADDING||new message)`
- This is not expected, and not a good property to have!
- Led to a number of attacks with authentication cookies (eg Flickr)
- Solution is to use HMAC (hash based MAC) formulation, or to use SHA-3!

Hash uses

- Why can't we use a hash as an encryption function in counter mode?
  - HASH(secret||n) xor BLOCK_n
  - Thoughts?
  - You can!!! It just turns out to be slower, so people don't use it!

Hash uses

- Proof of work
  - Somewhat like a CAPTCHA, with a purpose of rate limiting something
  - eg. sending emails to prevent spam, submitting answers to a website
  - rather than enforce a strict time delay, simply require users to "do some work" first
  - Server sends "hash prefix" and "message prefix"
  - user *brute forces* until finding a message suffix such that HASH(prefix||suffix) has the desired hash prefix

Hash uses

- Proof of work
    - Can make work arbitrarily hard by increasing the prefix length!
    - See: bitcoins

Hashes

- Questions up to this point?

Cryptography

- Takeaway lessons:
  - Variety of primitives from asymmetric crypto, symmetric crypto, and hash functions
  - Can combine primitives to more powerful and secure constructions
  - Incredibly hard to write code that implements crypto securely! Please don't try to write crypto code yourself!