When evaluating a crypto system:
1. Does it work? (can bob decrypt?)
2. Is it secure?

Counter-mode encryption.
- “Key stream”: a key is run through AES with a bunch of different values that are then xor'd with the individual message chunks. The keystream values appear random to an adversary that doesn’t know k. Therefore, the ciphertexts appear totally random because the messages are xor'd with randomness.
- Fully parallelizable, unlike CBC.
- Don't have to pad messages if the message is not a multiple of block len, because the COUNTERs are used in AES instead of the messages.

Birthday attacks
- 366 possible birthdays, with ~23 people in a room, good chance (~50%) that 2 people have the same birthday
- For counter mode, if we have a counter collision (i.e., 2 counters are the same), that’s bad. If the adversary sees different messages, they can start xor'ing things together. If you xor 2 cipher texts that use the same 2 counters, you get back the xor of the 2 original messages.
- Vulnerable under a chosen plaintext attack.
  - 2^128 different 128-bit keys (or other random values)
  - Pick one key at random. To exhaustively search for this key requires trying on average 2^127 keys.
  - Expect a “collision” after selecting approximately 2^64 random keys.
  - 64 bits of security against collision attacks, not 128 bits.
  1. Don’t encrypt too many messages with the same key.
  2. Partition into random part and counter part.

Hash functions.
- A function \( h \) that takes some value from a domain and maps it to a range, with the property that the domain is much bigger than the range.
- Map arbitrary-bit lengths to n-bit range.
- Used for message authentication, digital signatures, etc.
- Examples: (MD4, MD5, SHA1 - all bad). (SHA256, SHA512, SHA3 - ok for now).
- Hash functions underpin computer use today.
  - Downloading and verifying software
  - Password verification
  - SSL/SSH
  - “fun” use cases: secure bit flipping
Properties:

One-wayness -- given range point should be hard to find domain point.
- Select domain element.
- Hash it.
- Give hash to adversary. Adversary should not be able to find a preimage (except for exhaustive search).

Second preimage resistance -- given x and h(x), find x' such that h(x') = h(x)
- Select point in domain (x).
- Hash it (call result y).
- Give x, y to adversary. Adversary should not be able to find new x'!=x s.t.
  hash(x') = y

Collision-resistance -- Should be hard to find two different elements x, x' that hash to the same value.

Birthday paradox. Pick x, x1, x2, ... What is prob that any two collide? $2^{^n}$
How many pairs do we want: $O(2^{^n})$
Choose t pairs such that t choose 2 is $2^{^n}$
t around $2^{^n}/2$

Example: linkedin.com password storage. Suppose passwords are stored in plaintext.
- alice: abc123
- bob: xyz123
- Alice needs to send username/password to server so the server can check.
- What can go wrong in the login process? Well… attackers can dump the linkedin database and now have everyone’s passwords. (Linkedin actually did this and got breached…)
- So instead, linkedin can store hashed passwords, not plaintext.
- Now when alice enters password, server can just hash it and compare the hashes.
- Now adversary needs to check all possible passwords until he/she finds one that hashes to alice’s value. If the hash function is secure, this is the ONLY way the adversary can get a preimage. But what if it’s not secure?

Example: Alice and Bob want to flip a coin over the phone (bit commitment).
- What won’t work:
  - A picks as random bit ba. Bob picks a random bit bb.
  - Heads if ba^bb = 1, tails if ba^bb = 0 (^ is xor)
  - But how do alice and bob share their bits?
  - Can’t rely on sending the bits “at the same time”.
- Instead:
Alice picks a long random string sa, bob picks sb.

Alice sends h(ba, sa) to bob, and Bob sends h(bb, sb) to bob.

- Oneway: can't figure out ba from h(ba, sa) and bb from h(bb, sb)

After these values are received, alice and bob both send ba,sa and bb,sb in cleartext.

- Collision resistance means altering ba or bb after the fact will cause the hashes to not match up.

Now they both compute ba^bb, and they verify the hashes match up too.

What properties do we need?
(*-discuss) UNIX passwords stored as hash(password)

One-wayness: hard to recover the/a valid password

Integrity of software distribution

Weak collision resistance (second-preimage resistance)

But software images are not really random...

Collision resistance if considering malicious developers

Auction bidding

Alice wants to bid B, sends H(B), later reveals B

One-wayness: rival bidders should not recover B (this may mean that she needs to hash some randomness with B too)

Collision resistance: Alice should not be able to change her mind to bid B’ such that H(B)=H(B’)

Old hash functions: MD5 (128-bit), SHA1 (160-bit), SHA256,SHA512, SHA3

MACs

HMAC - hash function based mac

\[ T = H(K \text{ xor } \text{ipad, } M) \rightarrow \text{ipad=internal padding value} \]

\[ T = H(K \text{ xor } \text{opad, } T) \rightarrow \text{opad=outer padding value} \]

- HMAC standard defines ipad and opad, so they are basically global variables that all parties know about.

- To verify, re-run the above procedure using the message M sent over the wire and generate a candidate tag X. Check if X=T (the tag sent over the wire)

- Why do we run twice?
  - The inputs to the first hash function are variable length messages.
  - The inputs to the second hash are all the same fixed length, since they are outputs from the first hash. This makes it easier to reason about security properties than only dealing with variable length inputs.

Encrypt-and-MAC
- Message M
- Encrypt it with key k1.
- Mac it with key k2.
- Cipher text we send is the concatenation of the encryption and mac.

Mac-then-encrypt → what you would expect. Encrypt the output of the Mac.
Encrypt-then-mac → also what you would expect, use the output of encrypt as input to the mac.
  - In general, encrypt-then-mac is best.