These notes are intended to remind me of what I think is important. It is not
a substitute for reading the book and coming to class.

Cryptographic Hashing

A hash function maps a large domain into a small range.

A cryptographic hash function has no key and has function signature

  {0,1}^* --> {0,1}^n

(ie, takes strings of any length and outputs n-bit string)

The most important properties for crypto hash h are:

- preimage resistance: Given z, it is hard to find x where h(x)=z
- 2nd preimage resistance: Given x it is hard to find y!=x where h(x)=h(y)
- Collision resistant: It is hard to find any x!=y where h(x)=h(y)

Other nice properties

- Efficiency: h is efficient to compute
- Random: h behaves like a public randomly chosen function

Most famous cryptographic hashes
- MD5 (1992): 128-bit. Broken 2004. Now collisions can be found within seconds.
- SHA-1 (1995): 160-bit. Weakness found 2005. Collisions in 2^51 operations.
- SHA-2 (2001): 256 or 512 bit. Similar to SHA-1 but not same weakness.
- SHA-3 (2012): 256 or 512 bit. Competition similar to AES competition.

All of these except SHA-3 are based on the Merkle-Damgard construction

- Pad message to multiple of block length
- Process block at a time, feeding chaining value and block to
  "compression function"
- Compression function is a pseudo-random confusion/difusion function
- Output is final chaining value.

[Draw picture]

Problem: Extension attack

SHA-1 pads M as M || 10* || bitlength(M) where 10* are as needed to get to a
multiple of 512 bits.

Let

  y = SHA-1(M)

we know what comes out of the chaining value of M || 10* || bitlength(M)

Let

  M* = M || 10* || bitlength(M) || M'

If we know y we can now figure out SHA-1(M*) without ever knowing M.

Reductions

We say that Problem A reduces to Problem B if a Problem B Solver can be used as
a subroutine to create a Problem A Solver:

```
  ASolver(w)           \\ w is an instance of Problem A
    x = preprocess(w)  \\ Convert w into an instance of Problem B
    y = BSolver(x)     \\ y is the solution to x
    z = postprocess(y) \\ Convert the solution to x into a solution for w
    return z           \\ Return correct answer for w
```

If we could convince ourselves that there is a way to solve Problem A instances in this way, then we will have shown that

```
   The existence of BSolver implies the existence of ASolver
```

This is purely an existential statement. It is saying *if* a BSolver existed, then so would an ASolver exist. It does *not* say that an ASolver actually exists.

The contrapositive of this statement is of prime importance in security.

```
  If there is no ASolver, then there is no BSolver
```

This important, because we can define Problems A and B as that of breaking a security scheme.

```
  ABreaker(w)          \\ w is an instance of Problem A
    x = preprocess(w)  \\ Convert w into an instance of Problem B
    y = BBreaker(x)    \\ y is the solution to x
    z = postprocess(y) \\ Convert the solution to x into a solution for w
    return z           \\ Return correct answer for w
```

If we can show how to use a BBreaker to construct an ABreaker, then

```
  no ABreaker implies no BBreaker
```

or, in other words

```
  A secure implies B secure
```

This means that, for example

- If we make an encryption scheme using a block cipher (eg, CTR), and
- We show how to break the block cipher using an encryption scheme breaker,

then

```
  block cipher security implies encryption scheme security.
```

And so, if we trust our block cipher, then we can trust encryption scheme too.

--

Some simple general reductions:

Median finding reduces to sorting.
isPrime reduces to factoring.
hasClique reduces to subgraphIsomorphism

If A reduces to B:
  B solver --> A Solver   (A is not harder than B)
  no A Solver --> no B Solver (B is not easier than A)
--

CollisionFinder reduces to PreimageFinder

```
CollisionFinder()  // Return x!=y where h(x)==h(y)
  repeat
    let x be randomly chosen
    y = PreimageFinder(h(x))      // Returns "error" if fails
```

```
  until ((x!=y) && h(x)==h(y))
  return (x,y)

Why a loop?
- PreimageFinder might find x given h(x). Unlikely but possible.

If PreimageFinder works well, CollisionFinder works well
  (PreimageFinder exists --> CollisionFinder exists)
  (not CollisionFinder exists --> not PreimageFinder exists)
  (Collision resistant --> Preimage resistant)

-----

CollisionFinder reduces to SecondPreimageFinder

CollisionFinder()  // Return x!=y where h(x)==h(y)
  repeat
    let x be randomly chosen
    y = SecondPreimageFinder(x)      // Returns "error" if fails
  until (h(x)==h(y))
  return (x,y)

Why a loop?
- x might be the only input mapping to h(x). Unlikely but possible.

If SecondPreimageFinder works well, CollisionFinder works well
  (SecondPreimageFinder exists --> CollisionFinder exists)
  (not CollisionFinder exists --> not SecondPreimageFinder exists)
  (Collision resistant --> Second Preimage resistant)

-----

SecondPreimageFinder(x)   // Finds y!=x where h(x)==h(y)
   repeat
      y = PreimageFinder(h(x))
    until (x!=y)
    return y

Why a loop?
- PreimageFinder might find x given h(x). Unlikely but possible.
- If PreimageFinder is randomized, will return different preimage each time.

If PreimageFinder works well, SecondPreimageFinder works well
  (PreimageFinder exists --> SecondPreimageFinder exists)
  (not SecondPreimageFinder exists --> not PreimageFinder exists)
  (Second Preimage resistant --> Preimage resistant)

-----

PreimageFinder(y)   // Finds x where h(x) = y
   ...
   (a,b) = CollisionFinder()
   ...
   ???

(a,b) is unrelated to y, so doesn't help

-----

SecondPreimageFinder(x)   // Finds y!=x where h(x)==h(y)
   ...
   (a,b) = CollisionFinder()
   ...
   ???
```

(a,b) is unrelated to x, so doesn't help

```
PreimageFinder(y)   // Finds x where h(x) = y
   ...
   y' = SecondPreimageFinder(x)
   ...
   ???
```

Don't know a first preimage of y, so cannot ask for a second

CR --> SPR --> PR

CR gives us all three goals, and is therefore the main goal in a crypto hash.

--

Security reduction.

Show that breaking AES reduces to breaking CTR.

Here is an algorithm to break AES:

1. Let f be randomly chosen as either AES(K,x) for random K, or a random 128-bit function.

2. Let A be an adversary that is good at distinguishing CTR[AES] from an oracle that returns appropriate-length random strings.

3. Run A. For each of A's queries, compute the result using CTR[f] (ie, encrypt A's message using f instead of a block cipher).

4. If A guesses CTR[AES], we guess AES for f. If A guesses "random", we guess "random" for f.

Note that if f is AES, then our use of CTR[f] produces the exact same distribution of responses as CTR[AES], and if f is a random function, then CTR[f] produces the exact same distribution as random encryption.

So, if A does a good job breaking CTR[AES], then this algorithm does equally well breaking AES.

And the contrapositive:

  if nobody does well breaking AES, then nobody does well breaking CTR[AES].

Also,

  if we trust AES to be good, then we should trust CTR[AES] to be good.