11 Hash Functions

Suppose you share a huge file with a friend, but you are not sure whether you both have the same version of the file. You could send your version of the file to your friend and they could compare to their version. Is there any way to check that involves less communication than this?

Let's call your version of the file x (a string) and your friend's version y. The goal is to determine whether $x = y$. A natural approach is to agree on some deterministic function H, compute $H(x)$, and send it to your friend. Your friend can compute $H(y)$ and, since H is deterministic, compare the result to your $H(x)$. In order for this method to be fool-proof, we need H to have the property that different inputs always map to different outputs — in other words, H must be **injective** (1-to-1). Unfortunately, if H is injective and $H : \{0,1\}^{in} \to \{0,1\}^{out}$ is injective, then $out \geqslant in$. This means that sending $H(x)$ is no better/shorter than sending x itself!

Let us call a pair (x, y) a **collision** in H if $x \neq y$ and $H(x) = H(y)$. An injective function has no collisions. One common theme in cryptography is that you don't always need something to be *impossible*; it's often enough for that thing to be just highly unlikely. Instead of saying that H should have *no* collisions, what if we just say that collisions should be hard (for polynomial-time algorithms) to find? An H with this property will probably be good enough for anything we care about. It might also be possible to construct such an H with outputs that are shorter than its inputs!

What we have been describing is exactly a **cryptographic hash function.** A hash function has long inputs and short outputs — typically $H : \{0,1\}^* \to \{0,1\}^n$. Such an H must necessarily have many collisions. The security property of a hash function is that it is hard to find any such collision. Another good name for a hash function (which I just made up, and no one else uses) would be a "pseudo-injective" function. Although it is not injective, it behaves like one for our purposes.

11.1 Security Properties for Hash Functions

There are two common security properties of hash functions:

Collision resistance. It should be hard to compute any collision $x \neq x'$ such that $H(x) = H(x')$.

Second-preimage resistance. Given x, it should be hard to compute any collision involving x. In other words, it should be hard to compute $x' \neq x$ such that $H(x) = H(x')$.

© Copyright Mike Rosulek. Creative Commons BY-NC-SA 4.0. Latest version at joyofcryptography.com.

Brute Force Attacks on Hash Functions

There is an important difference between collision resistance and second-preimage resistance, which is reflected in the difficulty of their respective brute force attacks. Suppose H is a hash function whose outputs are n bits long. Let's make a simplifying assumption that for any $m > n$, the following distribution is roughly uniform over $\{0,1\}^n$:

```
x ← {0,1}^m
return H(x)
```

This is quite a realistic assumption for practical hash functions. If this were not true, then H would introduce some bias towards some outputs and away from other outputs, which would be perceived as suspicious. Also, as the output of H deviates farther from a uniform distribution, it only makes finding collisions easier.

Below are straight-forward brute-force attacks for collision resistance (left) and second-preimage resistance (right):

Collision brute force:

```
A_cr():
  for i = 1,...:
    x_i ← {0,1}^m
    y_i := H(x_i)
    if there is some j < i with x_i ≠ x_j
                        but y_i = y_j:
      return (x_i, x_j)
```

Second preimage brute force:

```
A_2pi(x):
  while true:
    x' ← {0,1}^m
    y' := H(x')
    if y' = H(x): return x'
```

Under the simplifying assumption on H, the collision-resistance brute force attack $\mathcal{A}_{\text{cr}}$ is essentially choosing each y_i uniformly at random. Since each $y_i \in \{0,1\}^n$, the probability of finding a repeated value after q times through the main loop is roughly $\Theta(q^2/2^n)$ by the birthday bound. While in the **worst case** it could take 2^n steps to find a collision in H, the birthday bound implies that it takes only $2^{n/2}$ attempts to find a collision with 99% probability (or any constant probability).

On the other hand, the second-preimage brute force attack $\mathcal{A}_{\text{2pi}}$ is given y as input and (under our simplifying assumption on H) essentially samples y' uniformly at random until y is the result. It will therefore take $\Theta(2^n)$ attempts in expectation to terminate successfully.[1]

There is a fundamental difference in how hard it is to break collision resistance and second-preimage resistance. Breaking collision-resistance is like inviting more people into the room until the room contains 2 people with the same birthday. Breaking second-preimage resistance is like inviting more people into the room until the room contains another person with *your* birthday. One of these fundamentally takes longer than the other.

[1] A well-known and useful fact from probability theory is that if an event happens with probability p, then the expected number of times to repeat before seeing the event is $1/p$. For example, the probability of rolling a 1 on a D6 die is 1/6, so it takes 6 rolls in expectation before seeing a 1. The probability of sampling a particular y from $\{0,1\}^n$ in one try is $1/2^n$, so the expected number of trials before seeing y is 2^n.

This difference explains why you will typically see cryptographic hash functions in practice that have 256- to 512-bit output length (but not 128-bit output length), while you only typically see block ciphers with 128-bit or 256-bit keys. In order to make brute force attacks cost 2^n, a block cipher needs only an n-bit key while a collision-resistant hash function needs a $2n$-bit output.

to-do *Discussion of these attacks in terms of graphs, where # of edges is the "number of chances" to get a collision. Collision-resistance brute force is a complete graph (need $\sqrt{N}$ vertices to have N edges / chances for a collision) . Second-preimage brute force is a star graph (need N vertices to N edges). Can generalize to consider complete bipartite graph between $\sqrt{N} + \sqrt{N}$ vertices.*

Hash Function Security In Practice

We will focus on developing a formal definition for collision resistance. We can take some inspiration from the security definition for MACs. Security for a MAC means that it should be hard to produce a forgery. The MAC security definition formalized that idea with one library that checks for a forgery and another library that assumes a forgery is impossible. If the two libraries are indistinguishable, then it must be hard to find a forgery.

We can take a similar approach to say that it should be hard to produce a collision. Here is an attempt:

```
TEST(x, x'):
  if x ≠ x' and H(x) = H(x'): return true
  else: return false
```

$\approx$

```
TEST(x, x'):
  return false
```

This corresponds to what I would call the "folk definition" of collision resistance. It makes intuitive sense (as long as you're comfortable with our style of security definition), but unfortunately the definition suffers from a very subtle technical problem.

Because of Kerckhoffs' principle, we allow calling programs to depend arbitrarily on the source code of the two libraries. This is a way of formalizing the idea that "the attacker knows everything about the algorithms." Our security definitions restrict calling programs to be polynomial-time algorithms, but they never consider *the effort that goes into finding the source code of the calling program!*

This strange loophole leads to the following valid attack. When we consider the security of some function H, we know that there exists many collisions (x, x') in H. These collisions may be hard to find, but they certainly exist. With exponential time, we could find such an (x, x') pair and write down the code of an attacker:

```
𝒜:
return TEST(x, x')
```

Here, the values x and x' are hard-coded into $\mathcal{A}$. The algorithm $\mathcal{A}$ is clearly polynomial-time (in fact, constant time). The "loophoole" is that the definition considers only the cost of *running* the algorithm $\mathcal{A}$, and not the cost of finding the source code of $\mathcal{A}$.

The way this kind of situation is avoided in other security definitions is that the libraries have some secret randomness. While the attacker is allowed to depend arbitrarily on the *source code* of the libraries, it is not allowed to depend on the *choice of outcomes* for random events in the libraries, like sampling a secret key. Since the calling program can't "prepare" for the random choice that it will be faced with, we don't have such trivial attacks. On the other hand, these two libraries for collision resistance are totally deterministic. There are no "surprises" about which function H the calling program will be asked to compute a collision for, so there is nothing to prevent a calling program from being "prepared" with a pre-computed collision in H.

Hash Function Security In Theory

The way around this technical issue is to introduce some randomness into the libraries and into the inputs of H. We define hash functions to take two arguments: a randomly chosen, public value s called a **salt**, and an adversarially chosen input x.

Definition 11.1 *A hash function H is **collision-resistant** if $\mathcal{L}^{\mathcal{H}}_{\text{cr-real}} \approx \mathcal{L}^{\mathcal{H}}_{\text{cr-fake}}$, where:*

$\mathcal{L}^{\mathcal{H}}_{\text{cr-real}}$

```
s ← {0,1}^λ

GETSALT():
  return s

TEST(x, x' ∈ {0,1}*):
  if x ≠ x' and H(s,x) = H(s,x'): return true
  return false
```

$\mathcal{L}^{\mathcal{H}}_{\text{cr-fake}}$

```
s ← {0,1}^λ

GETSALT():
  return s

TEST(x, x' ∈ {0,1}*):
  return false
```

The library initially samples the salt s. Unlike in other libraries, this value s is meant to be provided to the calling program, and so the library provides a way (GETSALT) for the calling program to learn it. The calling program then attempts to find a collision $x \neq x'$ where $H(s, x) = H(s, x')$.

I don't know why the term "salt" is used with hash functions. The reason appears to be a mystery to the Internet.[2] Think of salt as an extra value that **"personalizes" the hash function** for a given application. Here is a good analogy: an encryption scheme can be thought of as a different encryption algorithm $\text{Enc}(k, \cdot)$ for each choice of key k. When I choose a random k, I get a personalized encryption algorithm $\text{Enc}(k, \cdot)$ that is unrelated to the algorithm $\text{Enc}(k', \cdot)$ that someone else would get when they choose their own k. When I choose a salt s, I get a personalized hash function $H(s, \cdot)$ that is unrelated to other $H(s', \cdot)$ functions. Because the salt is chosen uniformly from $\{0,1\}^\lambda$, a calling program cannot predict what salt (which personalized hash function) it will be challenged with.

Definition 11.1 is a valid definition for collision resistance, free of strange loopholes like the "folklore" definition. However, it is not a particularly *useful* definition to use in security proofs, when a hash function is used as a building block in a bigger system.

[2]If you have an additional random argument to a hash function, but you keep it secret, it is called a "pepper." I'm serious, this is a real thing.

It becomes cumbersome to use in those cases, because when you use a hash function, you typically don't *explicitly check* whether you've seen a collision. Instead, you simply proceed as if collisions are not going to happen.

In this chapter, we won't see provable statements of security referring to this definition.

Salts in Practice

When we define hash functions in theory, we require that the hash function accept two inputs, the first of which is interpreted as a salt. The hash functions that you see in practice have only one input, a string of arbitrary length. You can simulate the effect of a salt for such a hash function by simply concatenating the two inputs — *e.g.*, $H(s\|x)$ instead of $H(s,x)$.

The concept of a **salted hash** is not just useful to make a coherent security definition, it is also just good practice. Hash functions are commonly used to store passwords. A server may store user records of the form $(\text{username}, h = H(\text{password}))$. When a user attempts to login with password p', the server computes $H(p')$ and compares it to h. Storing hashed passwords means that, in the event that the password file is stolen, an attacker would need to find a preimage of h in order to impersonate the user.

Best practice is to use a separate salt for each user. Instead of storing $(\text{username}, H(\text{password}))$, choose a random salt s for each user and store $(\text{username}, s, H(s, \text{password}))$. The security properties of a hash function do not require s to be secret, although there is also no good reason to broadcast a user's salt publicly. The salt is only needed by the server, when it verifies a password during a login attempt.

A user-specific salt means that each user gets their own "personalized" hash function to store their password. Salts offer the following benefits:

- Without salts, it would be evident when two users have the same password — they would have the same password hashes. The same password hashed with different salts will result in unrelated hash outputs.

- An attacker can compute a dictionary of $(p, H(p))$ for common passwords. Without salts, this dictionary makes it easy to attack *all users at once,* since all users are using the same hash function. With salts, each user has a personalized hash function, each of which would require its own dictionary. Salt makes an attacker's effort scale with the number of victims.

11.2 Merkle-Damgård Construction

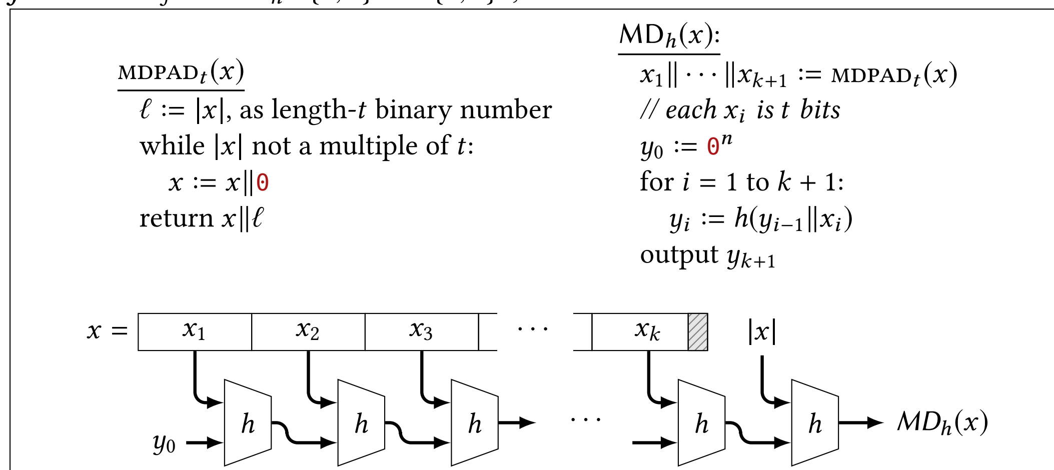
Building a hash function, especially one that accepts inputs of arbitrary length, seems like a challenging task. In this section, we'll see one approach for constructing hash functions, called the Merkle-Damgård construction.

Instead of a full-fledged hash function, imagine that we had a collision-resistant function whose inputs were of a single fixed length, but longer than its outputs. In other words, $h : \{0,1\}^{n+t} \to \{0,1\}^n$, where $t > 0$. We call such an h a **compression function**. This is not compression in the usual sense of the word — we are not concerned about recovering

the input from the output. We call it a compression function because it "compresses" its input by t bits (analogous to how a pseudorandom generator "stretches" its input by some amount).

The following construction is one way to build a full-fledged hash function (supporting inputs of arbitrary length) out of such a compression function:

Construction 11.2 (Merkle-Damgård) *Let $h : \{0,1\}^{n+t} \to \{0,1\}^n$ be a compression function. Then the* ***Merkle-Damgård transformation*** *of h is $MD_h : \{0,1\}^* \to \{0,1\}^n$, where:*



The idea of the Merkle-Damgård construction is to split the input x into blocks of size t. The end of the string is filled out with 0s if necessary. A final block called the "padding block" is added, which encodes the (original) length of x in binary.

Example *Suppose we have a compression function $h : \{0,1\}^{48} \to \{0,1\}^{32}$, so that $t = 16$. We build a Merkle-Damgård hash function out of this compression function and wish to compute the hash of the following 5-byte (40-bit) string:*

$$x = \texttt{01100011 11001101 01000011 10010111 01010000}$$

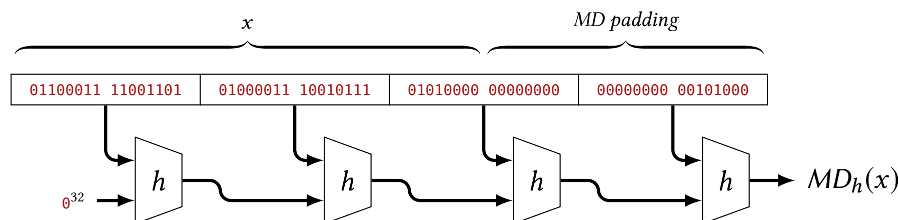
We must first pad x appropriately (MDPAD(x)):

- *Since x is not a multiple of $t = 16$ bits, we need to add 8 bits to make it so.*
- *Since $|x| = 40$, we need to add an extra 16-bit block that encodes the number 40 in binary (101000).*

After this padding, and splitting the result into blocks of length 16, we have the following:

$$\underbrace{\texttt{01100011 11001101}}_{x_1}\ \underbrace{\texttt{01000011 10010111}}_{x_2}\ \underbrace{\texttt{01010000 00000000}}_{x_3}\ \underbrace{\texttt{00000000 00101000}}_{x_4}$$

The final hash of x is computed as follows:



We are presenting a simplified version, in which MD_h accepts inputs whose maximum length is $2^t - 1$ bits (the length of the input must fit into t bits). By using multiple padding blocks (when necessary) and a suitable encoding of the original string length, the construction can be made to accommodate inputs of arbitrary length (see the exercises).

The value y_0 is called the **initialization vector** (IV), and it is a hard-coded part of the algorithm.

As discussed above, we will not be making provable security claims using the library-style definitions. However, we can justify the Merkle-Damgård construction with the following claim:

Claim 11.3 *Suppose h is a compression function and MD_h is the Merkle-Damgård construction applied to h. Given a collision x, x' in MD_h, it is easy to find a collision in h. In other words, if it is hard to find a collision in h, then it must also be hard to find a collision in MD_h.*

Proof Suppose that x, x' are a collision under MD_h. Define the values $x_1, \ldots, x_{k+1}$ and $y_1, \ldots, y_{k+1}$ as in the computation of $\text{MD}_h(x)$. Similarly, define $x'_1, \ldots, x'_{k'+1}$ and $y'_1, \ldots, y'_{k'+1}$ as in the computation of $\text{MD}_h(x')$. Note that, in general, k may not equal k'.

Recall that:

$$\text{MD}_h(x) = y_{k+1} = h(y_k \| x_{k+1})$$
$$\text{MD}_h(x') = y'_{k'+1} = h(y'_{k'} \| x'_{k'+1})$$

Since we are assuming $\text{MD}_h(x) = \text{MD}_h(x')$, we have $y_{k+1} = y'_{k'+1}$. We consider two cases:

Case 1: If $|x| \neq |x'|$, then the padding blocks x_{k+1} and $x'_{k'+1}$ which encode $|x|$ and $|x'|$ are not equal. Hence we have $y_k \| x_{k+1} \neq y'_{k'} \| x'_{k'+1}$, so $y_k \| x_{k+1}$ and $y'_{k'} \| x'_{k'+1}$ are a collision under h and we are done.

Case 2: If $|x| = |x'|$, then x and x' are broken into the same number of blocks, so $k = k'$. Let us work backwards from the final step in the computations of $\text{MD}_h(x)$ and $\text{MD}_h(x')$. We know that:

$$\begin{array}{ccc} y_{k+1} & = & h(y_k \| x_{k+1}) \\ = & & \\ y'_{k+1} & = & h(y'_k \| x'_{k+1}) \end{array}$$

If $y_k \| x_{k+1}$ and $y'_k \| x'_{k+1}$ are not equal, then they are a collision under h and we are done. Otherwise, we can apply the same logic again to y_k and y'_k, which are equal by our assumption.

More generally, if $y_i = y'_i$, then either $y_{i-1} \| x_i$ and $y'_{i-1} \| x'_i$ are a collision under h (and we say we are "lucky"), or else $y_{i-1} = y'_{i-1}$ (and we say we are "unlucky"). We start with the

premise that $y_k = y'_k$. Can we ever get "unlucky" every time, and not encounter a collision when propagating this logic back through the computations of $\text{MD}_h(x)$ and $\text{MD}_h(x')$? The answer is no, because encountering the unlucky case every time would imply that $x_i = x'_i$ for *all* i. That is, $x = x'$. But this contradicts our original assumption that $x \neq x'$. Hence we must encounter some "lucky" case and therefore a collision in h. ■

11.3 Hash Functions vs. MACs: Length-Extension Attacks

When we discuss hash functions, we generally consider the salt s to be public. A natural question is, **what happens when we make the salt private?** Of all the cryptographic primitives we have discussed so far, a hash function with secret salt most closely resembles a MAC. So, **do we get a secure MAC** by using a hash function with private salt?

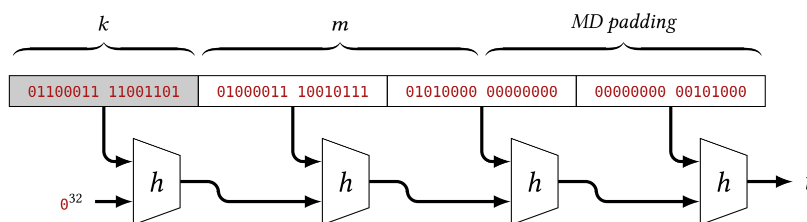
Unfortunately, the answer is no in general (although it can be yes in some cases, depending on the hash function). In particular, the method is insecure when H is constructed using the Merkle-Damgård approach. The key observation is that:

> *knowing $H(x)$ allows you to predict the hash of any string that begins with* MDPAD(x).

This concept is best illustrated by example.

Example *Let's return to our previous example, with a compression function $h : \{0,1\}^{48} \to \{0,1\}^{32}$. Suppose we construct a Merkle-Damgård hash out of this compression function, and use the construction $\text{MAC}(k,m) = H(k\|m)$ as a MAC.*

Suppose the MAC key is chosen as $k =$ `01100011 11001101`, and an attacker sees the MAC tag t of the message $m =$ `01000011 10010111 01010000`. Then $t = H(k\|m)$ corresponds exactly to the example from before:

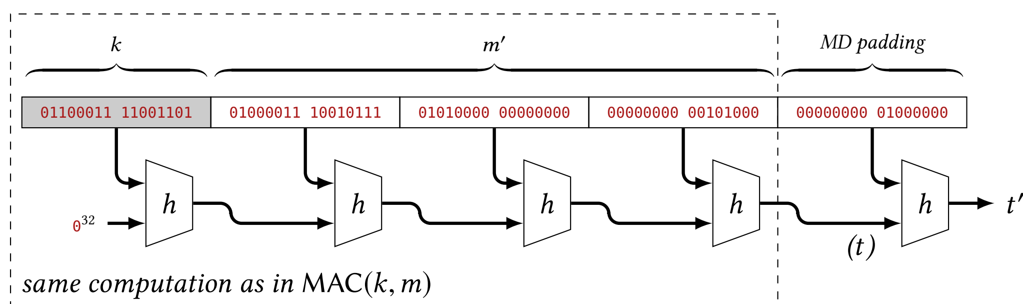


The only difference from before is that the first block contains the MAC key, so its value is not known to the attacker. We have shaded it in gray here. The attacker knows all other inputs as well as the output tag t.

I claim that the attacker can now exactly predict the tag of:

$$m' = \texttt{01000011 10010111 01010000 00000000 00000000 00101000}$$

The correct MAC tag t' of this value would be computed by someone with the key as:



The attacker can compute the output t' in a different way, without knowing the key. In particular, the attacker knows all inputs to the last instance of h. Since the h function itself is public, the attacker can compute this value herself as $t' = h(t\|$`00000000 01000000`$)$. Since she can predict the tag of m', having seen only the tag of m, she has broken the MAC scheme.

Discussion

- In our example, the attacker sees the MAC tag for m (computed as $H(k\|m)$) and then forges the tag for $m' = m\|p$, where p is the padding you must add when hashing $k\|m$. Note that the padding depends only on the *length* of k, which we assume is public.

- The same attack works to forge the tag of any m' that *begins with* $m\|p$. The attacker would simply have to compute the last several rounds (not just one round) of Merkle-Damgård herself.

- **This is not an attack on collision resistance!** Length-extension does not result in collisions! We are not saying that $k\|m$ and $k\|m\|p$ have the *same* hash under H, only that knowing the hash of $k\|m$ allows you to also compute the hash of $k\|m\|p$.

Knowing how $H(k\|m)$ fails to be a MAC helps us understand better ways to build a secure MAC from a hash function:

- The Merkle-Damgård approach suffers from length-extension attacks because it outputs its **entire internal state**. In the example picture above, the value t is both the output of $H(k\|m)$ as well as the only information about $k\|m$ needed to compute the last call to h in the computation $H(k\|m\|p)$.

  One way to avoid this problem is to only output part of the internal state. In Merkle-Damgård, we compute $y_i := h(y_{i-1}\|x_i)$ until reaching the final output y_{k+1}. Suppose instead that we only output half of y_{k+1} (the y_i values may need to be made longer in order for this to make sense). Then just knowing half of y_{k+1} is not enough to predict what the hash output will be in a length-extension scenario.

  The hash function **SHA-3** was designed in this way (often called a "wide pipe" construction). One of the explicit design criteria of SHA-3 was that $H(k\|m)$ would be a secure MAC.

- Length extension with Merkle-Damgård is possible because the computation of $H(k\|m)$ exactly appears during the computation of $H(k\|m\|p)$. Similar problems

appear in plain CBC-MAC when used with messages of mixed lengths. To avoid this, we can "do something different" to mark the end of the input. In a "wide pipe" construction, we throw away half of the internal state at the end. In ECBC-MAC, we use a different key for the last block of CBC chaining.

We can do something similar to the $H(k\|m)$ construction, by doing $H(k_2\|H(k_1\|m))$, with independent keys. This change is enough to mark the end of the input. This construction is known as **NMAC**, and it can be proven secure for Merkle-Damgård hash functions, under certain assumptions about their underlying compression function. A closely related (and popular) construction called **HMAC** allows k_1 and k_2 to even be related in some way.

Exercises

11.1. Sometimes when I verify an MD5 hash visually, I just check the first few and the last few hex digits, and don't really look at the middle of the hash.

Generate two files with opposite meanings, whose MD5 hashes agree in their first 16 bits (4 hex digits) and in their last 16 bits (4 hex digits). It could be two text files that say opposite things. It could be an image of Mario and an image of Bowser. I don't know, be creative.

As an example, the strings "`subtitle illusive planes`" and "`wantings premises forego`" actually agree in the first 20 and last 20 bits (first and last 5 hex digits) of their MD5 hashes, but it's not clear that they're very meaningful.

```
$ echo -n "subtitle illusive planes" | md5sum
4188d4cdcf2be92a112bdb8ce4500243 -
$ echo -n "wantings premises forego" | md5sum
4188d209a75e1a9b90c6fe3efe300243 -
```

Describe how you generated the files, and how many MD5 evaluations you had to make.

11.2. Let $h : \{0,1\}^{n+t} \to \{0,1\}^n$ be a fixed-length compression function. Suppose we forgot a few of the important features of the Merkle-Damgård transformation, and construct a hash function H from h as follows:

- Let x be the input.
- Split x into pieces $y_0, x_1, x_2, \ldots, x_k$, where y_0 is n bits, and each x_i is t bits. The last piece x_k should be padded with zeroes if necessary.
- For $i = 1$ to k, set $y_i = h(y_{i-1}\|x_i)$.
- Output y_k.

Basically, it is similar to the Merkle-Damgård except we lost the IV and we lost the final padding block.

1. Describe an easy way to find two messages that are broken up into the same number of pieces, which have the same hash value under H.

2. Describe an easy way to find two messages that are broken up into different number of pieces, which have the same hash value under H.

Hint: Pick any string of length $n + 2t$, then find a shorter string that collides with it.

Neither of your collisions above should involve finding a collision in h.

11.3. I've designed a hash function $H : \{0,1\}^* \to \{0,1\}^n$. One of my ideas is to make $H(x) = x$ if x is an n-bit string (assume the behavior of H is much more complicated on inputs of other lengths). That way, we know with certainty that there are no collisions among n-bit strings. Have I made a good design decision?

11.4. Same as above, but now if x is n bits long, then $H(x) = x \oplus m$, where m is a fixed, public string. Can this be a good hash function?

11.5. Let H be a hash function and let t be a fixed constant. Define $H^{(t)}$ as:

$$H^{(t)}(x) = \underbrace{H(\cdots H(H}_{t \text{ times}}(x))\cdots).$$

Show that if you are given a collision under $H^{(t)}$ then you can efficiently find a collision under H.

11.6. In this problem, if x and y are strings of the same length, then we write $x \sqsubseteq y$ if $x = y$ or x comes before y in standard dictionary ordering.

Suppose a function $H : \{0,1\}^* \to \{0,1\}^n$ has the following property. For all strings x and y of the same length, if $x \sqsubseteq y$ then $H(x) \sqsubseteq H(y)$. Show that H is **not** collision resistant (describe how to efficiently find a collision in such a function).

Hint: Binary search, always recursing on a range that is *guaranteed* to contain a collision.

★ 11.7. Suppose a function $H : \{0,1\}^* \to \{0,1\}^n$ has the following property. For all strings x and y of the same length, $H(x \oplus y) = H(x) \oplus H(y)$. Show that H is **not** collision resistant (describe how to efficiently find a collision in such a function).

★ 11.8. Let H be a salted hash function with n bits of output, and define the following function:

$$\underline{H^*(x_1\|x_2\|x_3\|\cdots\|x_k):}$$
$$\text{return } H(1,x_1) \oplus H(2,x_2) \oplus \cdots \oplus H(k,x_k)$$

Note that H^* can take inputs of any length (k). Show how to find collisions in H^* when $k > n$.

11.9. Generalize the Merkle-Damgård construction so that it works for arbitrary input lengths (and arbitrary values of t in the compression function). Extend the proof of Claim 11.3 to your new construction.

★ 11.10. Let F be a secure PRF with n-bit inputs, and let H be a collision-resistant (salted) hash function with n-bit outputs. Define the new function $F'((k,s),x) = F(k,H(s,x))$, where we interpret (k,s) to be its key. Prove that F' is a secure PRF with arbitrary-length inputs.

★ 11.11. Let MAC be a secure MAC algorithm with n-bit inputs, and let H be a collision-resistant (salted) hash function with n-bit outputs. Define the new function $\text{MAC}'((k,s),x) = \text{MAC}(k, H(s,x))$, where we interpret (k,s) to be its key. Prove that MAC′ is a secure MAC with arbitrary-length inputs.

11.12. More exotic issues with the Merkle-Damgård construction:

(a) Let H be a hash function with n-bit output, based on the Merkle-Damgård construction. Show how to compute (with high probability) 4 messages that all hash to the same value under H, using only $\sim 2 \cdot 2^{n/2}$ calls to H.

Hint: The 4 messages that collide will have the form $x\|y$, $x\|y'$, $x'\|y$ and $x'\|y'$. Use a length-extension idea and perform 2 birthday attacks.

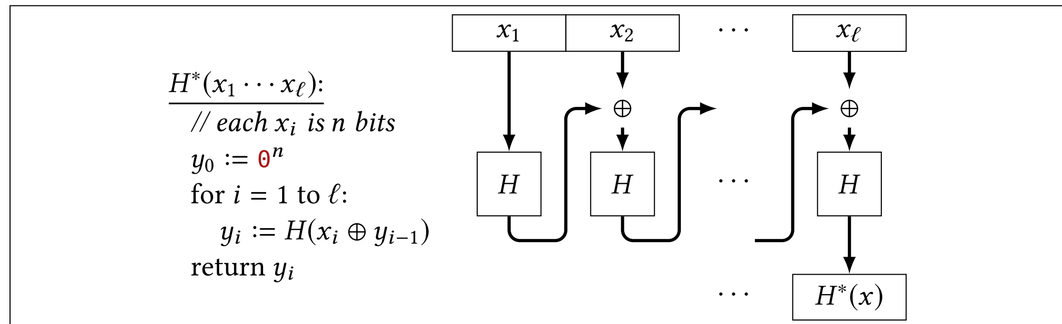
(b) Show how to construct 2^d messages that all hash to the same value under H, using only $O(d \cdot 2^{n/2})$ evaluations of H.

(c) Suppose H_1 and H_2 are (different) hash functions, both with n-bit output. Consider the function $H^*(x) = H_1(x)\|H_2(x)$. Since H^* has $2n$-bit output, it is tempting to think that finding a collision in H^* will take $2^{(2n)/2} = 2^n$ effort.

However, this intuition is not true when H_1 is a Merkle-Damgård hash. Show that when H_1 is Merkle-Damgård, then it is possible to find collisions in H^* with only $O(n2^{n/2})$ effort. The attack should assume nothing about H_2 (i.e., H_2 need not be Merkle-Damgård).

Hint: Applying part (b), first find a set of $2^{n/2}$ messages that all have the same hash under H_1. Among them, find 2 that also collide under H_2.

11.13. Let H be a collision-resistant hash function with output length n. Let H^* denote iterating H in a manner similar to CBC-MAC:



Show that H^* is **not** collision-resistant. Describe a successful attack.

11.14. Show that a bare PRP is not collision resistant. In other words, if F is a secure PRP, then show how to efficiently find collisions in $H(x\|y) = F(x,y)$.

11.15. Show that the CBC-MAC construction applied to a PRP is not collision-resistant. More precisely, let F be a secure PRP. Show how to efficiently find collisions in the following

salted hash function H:

```
H(k, m_1‖m_2‖m_3):
  c_1 := F(k, m_1)
  c_2 := F(k, m_2 ⊕ c_1)
  c_3 := F(k, m_3 ⊕ c_2)
  return c_3
```

Here we are interpreting k as the salt. This is yet another example of how collision-resistance is different than authenticity (MAC).

11.16. Let $H : \{0,1\}^\lambda \to \{0,1\}^\lambda$ be any function, and define the following function $H^* : \{0,1\}^{2\lambda} \to \{0,1\}^\lambda$:

```
H*(x‖y):
  z := H(x) ⊕ y
  return H(z) ⊕ x
```

Show how to succeed in an efficient second-preimage attack on H^*.

11.17. Adding a plain hash to a plaintext does not result in CCA security. Consider the following approach for encryption, that uses a plain (unsalted) hash function H. To encrypt plaintext m, simply encrypt $m\|H(m)$ under CTR mode. To decrypt, use normal CTR mode decryption but return err if the plaintext does not have the form $m\|H(m)$ (*i.e.*, if the last n bits are not a hash of the rest of the CTR-plaintext).

Show that the scheme does **not** have CCA security.

11.18. In the discussion of length-extension attacks, we noted that a natural way to stop them is to "do something different" for the last block of Merkle-Damgård. Suppose after performing the final call to h in Merkle-Damgård, we complement the value (y_{k+1}). Does this modified scheme still have length-extension properties?