12 Hash Functions

A **hash function** is any function that takes arbitrary-length input and has fixed-length output, so $H : \{0,1\}^* \to \{0,1\}^n$. Think of $H(m)$ as a "fingerprint" of m. Calling $H(m)$ a fingerprint suggests that different messages always have different fingerprints. But we know that can't be true — there are infinitely many messages but only 2^n possible outputs of H.[1]

Let's look a little closer. A true "unique fingerprinting function" would have to be *injective* (one-to-one). No hash function can be injective, since there must exist many x and x' with $x \neq x'$ but $H(x) = H(x')$. Let us call (x, x') a **collision** under H if it has this property. We know that collisions must exist, but what if *the problem of finding a collision* was hard for polynomial-time programs? Recall that in this course we often don't care whether something is impossible in principle — it is enough for things to be merely computationally difficult in this way. A hash function for which collision-finding is hard would effectively serve as an injective function for our purposes.

Roughly speaking, a hash function H is **collision-resistant** if no polynomial-time program can find a collision in H. Another good name for such a hash function might be "pseudo-injective." In this chapter we discuss definitions and applications of collision resistance.

12.1 Defining Security

Superficially, it seems like we have already given the formal definition of security: A hash function H is collision-resistant if no polynomial-time algorithm can output a collision under H. Unfortunately, this definition is impossible to achieve!

Fix your favorite hash function H. We argued that collisions in H definitely exist in a mathematical sense. Let x, x' be one such collision, and consider the adversary $\mathcal{A}$ that has x, x' hard-coded and simply outputs them. This adversary runs in constant time and finds a collision in H. Of course, even though $\mathcal{A}$ exists in a mathematical sense, it might be hard to *write down* the code of such an $\mathcal{A}$ given H. But (and this is a subtle technical point!) security definitions consider only the running time of $\mathcal{A}$, and not the effort that goes into *finding the source code* of $\mathcal{A}$.[2]

The way around this problem is to introduce some random choice made by the user of the hash function, who wants collisions to be hard to find. A **hash function family** is a set $\mathcal{H}$ of functions, where each function $H \in \mathcal{H}$ is a hash function with the same output length. We will require that collisions are hard to find, in a hash function *chosen randomly from the family.* This is enough to foil the hard-coded-collision distinguisher mentioned

[1]Somewhere out there is a pigeonhole with infinitely many pigeons in it.

[2]The reason we don't define security this way is that as soon as someone *does* find the code of such an $\mathcal{A}$, the hash function H is "broken" forever. Nothing anyone does (like choosing a new key) can salvage it.

above. Think of a hash function family as having exponentially many functions in it — then no polynomial-time program can have a hard-coded collision for *all* of them.

Now the difficulty of finding collisions rests in the random choice of functions. An adversary can know every fact about $\mathcal{H}$, it just doesn't know which $H \in \mathcal{H}$ it is going to be challenged on to find a collision. It's similar to how the security of other cryptographic schemes rests in the random choice of key. But in this case there is no secrecy involved, only unpredictability. The choice of H is made public to the adversary.

Note also that this definition is a mismatch to the way hash functions are typically used in practice. There, we usually do have a *single* hash function that we rely on and standardize. While it is possible to adapt the definitions and results in this lecture to the setting of fixed hash functions, it's simpler to consider hash function *families.* If you're having trouble connecting the idea of a hash function *family* to reality, imagine taking a standardized hash function like MD5 or SHA3 and considering the family of functions you get by varying the initialization parameters in those standards.

Towards the Formal Definition

The straight-forward way to define collision resistance of a hash family $\mathcal{H}$ is to say that the following two libraries should be indistinguishable:

```
$H \leftarrow \mathcal{H}$

GETH():
  return $H$

CHALLENGE($x, x' \in \{0,1\}^*$):
  return $H(x) \overset{?}{=} H(x')$
```

```
$H \leftarrow \mathcal{H}$

GETH():
  return $H$

CHALLENGE($x, x' \in \{0,1\}^*$):
  return $x \overset{?}{=} x'$
```

Indeed, this is a fine definition of collision resistance, and it follows in the style of previous security definitions. The two libraries give different outputs only when called on (x, x') where $x \neq x'$ but $H(x) = H(x')$ — *i.e.*, an H-collision.

However, it turns out to not be a particularly convenient definition to *use* when proving security of a construction that involves a hash function as a building block. To see why, think back to the security of MACs. The difference between the two $\mathcal{L}_{\text{mac-}\star}$ libraries is in the verification subroutine VER. And indeed, constructions that use MACs as a building block often perform MAC verification. The libraries — in particular, their VER subroutine — are a good fit for how MACs are used.

On the other hand, cryptographic constructions don't usually *explicitly test for collisions* when using a hash function. Rather, they typically compute the hash of some value and move on with their business *under the assumption that a collision has never been encountered.* To model this in a security definition, we use the approach below:

Definition 12.1 *Let $\mathcal{H}$ be a family of hash functions. Then $\mathcal{H}$ is* ***collision-resistant*** *if $\mathcal{L}^{\mathcal{H}}_{\text{cr-real}} \approxeq \mathcal{L}^{\mathcal{H}}_{\text{cr-fake}}$, where:*

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

GETH():
  return $H$

HASH($x \in \{0,1\}^*$):
  $y := H(x)$
  return $y$
```

```
$\mathcal{L}^{\mathcal{H}}_{\text{cr-fake}}$
$H \leftarrow \mathcal{H}$
$H^{-1}$ := empty assoc. array

GETH():
  return $H$

HASH($x \in \{0,1\}^*$):
  $y := H(x)$
  if $H^{-1}[y]$ defined and $H^{-1}[y] \neq x$:
    **self destruct**
  $H^{-1}[y] := x$
  return $y$
```

Discussion:

- The two libraries have identical behavior, except in the event that $\mathcal{L}_{\text{cr-fake}}$ triggers a "**self destruct**" statement. Think of this statement as an exception that kills the entire program, including the distinguisher. In the case that the distinguisher is killed in this way, we take its output to be 0. Suppose a distinguisher A always outputs 1 under normal, explosion-free execution. Since an explosion happens only in $\mathcal{L}_{\text{cr-fake}}$, A's advantage is simply the probability of an explosion. So if the two libraries are supposed to be indistinguishable, then it must be that **self destruct** happens with only negligible probability.

- In $\mathcal{L}_{\text{cr-fake}}$ we have given the associative array "H^{-1}" a somewhat suggestive name. During normal operation, $H^{-1}[y]$ contains the *unique* value seen by the library whose hash is y. A collision happens when the library has seen two distinct values x and x' that hash to the same y. When the library sees the second of these values x', it computes $H(x') = y$ and discovers that $H^{-1}[y]$ already exists but is not equal to x'. This is the situation in which the library **self destruct**s.

- Why do we make the library **self destruct** when it sees a collision, rather than just returning some error indicator? The reason is simply that it's easier to just *assume* there is no collision than to check the return value of HASH after each call.

  Think of $\mathcal{L}_{\text{cr-real}}$ as a world in which you take hashes of things but you might see a collision and never notice. Then $\mathcal{L}_{\text{cr-fake}}$ is a kind of thought-experiment in which some all-seeing judge ends the game immediately if there was ever a collision among the values that you hashed. The judge's presence simplifies things a bit. There's no real need to explicitly check for collisions yourself; you can just go about your business knowing that as long as the game is still going, the hash function is injective among all the values you've seen.

- The libraries have no secrets! H is public (the adversary can freely obtain it through GETH). However, note that the library — not the adversary — chooses H. This random choice of H is the sole source of security.

Since H is public, the adversary doesn't really need to call the subroutine $\textsc{hash}(x)$ to compute $H(x)$ — he could compute $H(x)$ locally. Intuitively, the library is used in a security proof to model the actions of the "good guys" who are operating on the assumption that H is collision-resistant. If an adversary finds a collision but never causes the "good guys" to evaluate H on it, then the adversary never violates their security assumption.

Other variants of collision-resistance.

There are some other variants of collision-resistance that are often discussed in practice. We don't define them formally, but give the rough idea:

Target collision resistance. Given H and $H(x)$, where $H \leftarrow \mathcal{H}$ and $x \leftarrow \{0,1\}^\ell$ are chosen randomly, it should be infeasible to compute a value x' (possibly equal to x) such that $H(x) = H(x')$.

Second-preimage resistance. Given H and x, where $H \leftarrow \mathcal{H}$ and $x \leftarrow \{0,1\}^\ell$ are chosen randomly, it should be infeasible to compute a value $x' \neq x$ such that $H(x) = H(x')$.

These conditions are weaker than the condition of (plain) collision-resistance, in the sense that if $\mathcal{H}$ is collision-resistant, then $\mathcal{H}$ is also target collision-resistant and second-preimage resistant. Hence, we focus on plain collision resistance in this course.

12.2 Hash-Then-MAC

In this section we'll see a simple application of collision resistance. It is a common theme in cryptography, that instead of dealing with large data it is often sufficient to deal with only a hash of that data. This theme is true in particular in the context of MACs.

One particularly simple way to construct a secure MAC is to use a PRF directly as a MAC. However, PRFs (and in particular, block ciphers) often have a short fixed input length, making them suitable only for MACs of such short messages. To extend such a MAC to longer inputs, it suffices to compute a MAC of the hash of the data. This idea is formalized in the following construction:

Construction 12.2 (Hash-then-MAC) *Let M be a MAC scheme with message space $\mathcal{M} = \{0,1\}^n$ and let $\mathcal{H}$ be a hash family with output length n. Then **hash-then-MAC** (HtM) refers to the following MAC scheme:*

```
$\mathcal{K} = M.\mathcal{K} \times \mathcal{H}$      $HtM$.KeyGen:           $HtM$.MAC($(k,H), m$):
$\mathcal{M} = \{0,1\}^*$               $k \leftarrow M$.KeyGen      $y := H(m)$
                              $H \leftarrow \mathcal{H}$               $t := M$.MAC$(k,y)$
                              return $(k,H)$        return $t$
```

Claim 12.3 *Construction 12.2 is a secure MAC, if $\mathcal{H}$ is collision-resistant and M is a secure MAC.*

Proof We prove the security of HtM using a standard hybrid approach.

```
$\mathcal{L}^{HtM}_{\text{mac-real}}$

$k \leftarrow \text{KeyGen}$
$H \leftarrow \mathcal{H}$

$\textsc{getmac}(m)$:
    $y := H(m)$
    $t := \text{MAC}(k, y)$
    return $t$

$\textsc{ver}(m, t)$:
    $y := H(m)$
    return $t \overset{?}{=} \text{MAC}(k, y)$
```

The starting point is $\mathcal{L}^{HtM}_{\text{mac-real}}$, shown here with the details of HtM filled in. Our goal is to eventually reach $\mathcal{L}_{\text{mac-fake}}$, where the VER subroutine returns false unless (m, t) was generated by the GETMAC subroutine.

```
$k \leftarrow \text{KeyGen}$
$H \leftarrow \mathcal{H}$
$\mathcal{T} := \emptyset$

$\textsc{getmac}(m)$:
    $y := H(m)$
    $t := \text{MAC}(k, y)$
    $\mathcal{T} := \mathcal{T} \cup \{(y, t)\}$
    return $t$

$\textsc{ver}(m, t)$:
    $y := H(m)$
    return $(y, t) \overset{?}{\in} \mathcal{T}$
```

We have applied the MAC security of M, omitting the usual details (factor out, swap libraries, inline). Now VER returns false unless $(H(m), t) \in \mathcal{T}$.

```
$k \leftarrow$ KeyGen
$H \leftarrow \mathcal{H}$
$\mathcal{T} := \emptyset$
$H^{-1} :=$ empty

GETMAC$(m)$:
    $y := H(m)$
    if $H^{-1}[y] \notin \{\text{undef}, m\}$:
        self destruct
    $H^{-1}[y] := m$
    $t := \text{MAC}(k, h)$
    $\mathcal{T} := \mathcal{T} \cup \{(y,t)\}$
    return $t$

VER$(m,t)$:
    $y := H(m)$
    if $H^{-1}[y] \notin \{\text{undef}, m\}$:
        self destruct
    $H^{-1}[y] := y$
    return $(y,t) \overset{?}{\in} \mathcal{T}$
```

Here we have applied the security of the hash family $\mathcal{H}$. We have factored out all calls to H in terms of $\mathcal{L}^{\mathcal{H}}_{\text{cr-real}}$, replaced $\mathcal{L}_{\text{cr-real}}$ with $\mathcal{L}_{\text{cr-fake}}$, and then inlined.

```
$k \leftarrow$ KeyGen
$H \leftarrow \mathcal{H}$
$\mathcal{T} := \emptyset$; $\mathcal{T}' := \emptyset$
$H^{-1} :=$ empty

GETMAC$(m)$:
    $y := H(m)$
    if $H^{-1}[y] \notin \{\text{undef}, m\}$:
        self destruct
    $H^{-1}[y] := m$
    $t := \text{MAC}(k, y)$
    $\mathcal{T} := \mathcal{T} \cup \{(y,t)\}$
    $\mathcal{T}' := \mathcal{T}' \cup \{(m,t)\}$
    return $t$

VER$(m,t)$:
    $y := H(m)$
    if $H^{-1}[y] \notin \{\text{undef}, m\}$:
        self destruct
    $H^{-1}[y] := m$
    return $(y,t) \overset{?}{\in} \mathcal{T}$
```

Now we have simply added another set $\mathcal{T}'$ which is never actually used. We point out two things: First, in GETMAC, (y,t) added to $\mathcal{T}$ if and only if $(H^{-1}[y],t)$ is added to $\mathcal{T}'$. Second, if the last line of VER is reached, then the library has not self destructed. So $H^{-1}[y] = m$, and in fact $H^{-1}[y]$ has never been defined to be anything else. This means the last line of VER is equivalent to:

$$(y,t) \in \mathcal{T} \Leftrightarrow (H^{-1}[y],t) \in \mathcal{T}' \Leftrightarrow (m,t) \in \mathcal{T}'.$$

```
$k \leftarrow \text{KeyGen}$
$H \leftarrow \mathcal{H}$
$\mathcal{T} := \emptyset; \mathcal{T}' := \emptyset$
$H^{-1} := \text{empty}$

GETMAC($m$):
  $y := H(m)$
  if $H^{-1}[y] \notin \{\text{undef}, m\}$:
    self destruct
  $H^{-1}[y] := m$
  $t := \text{MAC}(k, y)$
  $\mathcal{T} := \mathcal{T} \cup \{(y,t)\}$
  $\mathcal{T}' := \mathcal{T}' \cup \{(m,t)\}$
  return $t$

VER($m,t$):
  $y := H(m)$
  if $H^{-1}[y] \notin \{\text{undef}, m\}$:
    self destruct
  $H^{-1}[y] := m$
  return $(m,t) \overset{?}{\in} \mathcal{T}'$
```

We have replaced the condition $(H(m),t) \in \mathcal{T}$ with $(m,t) \in \mathcal{T}'$. But we just argued that these statements are logically equivalent within the library.

```
$k \leftarrow \text{KeyGen}$
$H \leftarrow \mathcal{H}$
$\mathcal{T}' := \emptyset$

GETMAC($m$):
  $y := H(m)$
  $t := \text{MAC}(k, y)$
  $\mathcal{T}' := \mathcal{T}' \cup \{(m,t)\}$
  return $t$

VER($m,t$):
  return $(m,t) \overset{?}{\in} \mathcal{T}'$
```

We remove the variable H^{-1} and **self destruct** statements via a standard sequence of changes (*i.e.*, factor out in terms of $\mathcal{L}_{\text{cr-fake}}$, replace with $\mathcal{L}_{\text{cr-real}}$, inline). We also remove the now-unused variable $\mathcal{T}$. The result is $\mathcal{L}^{HtM}_{\text{mac-fake}}$, as desired.

The next-to-last hybrid is the key step where collision-resistance comes into our reasoning. Indeed, a collision would break down the argument. If the adversary manages to find a collision $y = H(m) = H(m')$, then it could be that $(y,t) \in \mathcal{T}$ and $(m,t) \in \mathcal{T}'$ but $(m',t) \notin \mathcal{T}'$. This corresponds to a forgery of HtM in a natural way: Ask for a MAC of m, which is $t = \text{MAC}(k, H(m))$; then t is also a valid MAC of m'. ■

12.3 Merkle-Damgård Construction

Constructing a hash function seems like a challenging task, especially given that it must accept strings of arbitrary length as input. 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 (family) whose inputs were of a single fixed length, but longer than its outputs. In other words, suppose we had a family $\mathcal{H}$ of functions $h : \{0,1\}^{n+t} \to \{0,1\}^n$, where $t > 0$. We call such an h a **compression function**. This is not compression in the usual sense of data compression — 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).

We can apply the standard definition of collision-resistance to a family of *compression* functions, by restricting our interest to inputs of length exactly $n + t$. The functions in the family are not defined for any other input length.

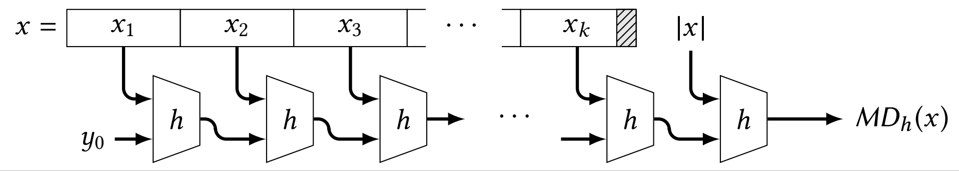
The following construction is one way to extend a compression function into a full-fledged hash function accepting arbitrary-length inputs:

Construction 12.4 (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:*

```
MDPAD_t(x)
  $\ell := |x|$, as length-$t$ binary number
  while $|x|$ not a multiple of $t$:
    $x := x \| 0$
  return $x \| \ell$
```

```
MD_h(x):
  $x_1 \| \cdots \| x_{k+1} := \textsc{mdpad}_t(x)$
  // each $x_i$ is $t$ bits
  $y_0 := 0^n$
  for $i = 1$ to $k + 1$:
    $y_i := h(y_{i-1} \| x_i)$
  output $y_{k+1}$
```



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.

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 accomodate 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. In practice, a more "random-looking" value is used as the initialization vector. Or one can think of the Merkle-Damgård construction as defining a **family** of hash functions, corresponding to the different choices of IV.

Claim 12.5 *Let $\mathcal{H}$ be a family of compression functions, and define $MD_{\mathcal{H}} = \{MD_h \mid h \in \mathcal{H}\}$ (a family of hash functions). If $\mathcal{H}$ is collision-resistant, then so is $MD_{\mathcal{H}}$.*

Proof While the proof can be carried out in the style of our library-based security definitions, it's actually much easier to simply show the following: given any collision under MD_h, we can efficiently find an collision under h. This means that any successful adversary violating the collision-resistance of $\mathsf{MD}_{\mathcal{H}}$ can be transformed into a successful adversary violating the collision resistance of $\mathcal{H}$. So if $\mathcal{H}$ is collision-resistant, then so is $\mathsf{MD}_{\mathcal{H}}$.

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 $\mathsf{MD}_h(x)$. Similarly, define $x'_1, \ldots, x'_{k'+1}$ and $y'_1, \ldots, y'_{k'+1}$ as in the computation of $\mathsf{MD}_h(x')$. Note that, in general, k may not equal k'.

Recall that:

$$\begin{aligned} \mathsf{MD}_h(x) &= y_{k+1} = h(y_k \| x_{k+1}) \\ \mathsf{MD}_h(x') &= y'_{k'+1} = h(y'_{k'} \| x'_{k'+1}) \end{aligned}$$

Since we are assuming $\mathsf{MD}_h(x) = \mathsf{MD}_h(x')$, 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 $\mathsf{MD}_h(x)$ and $\mathsf{MD}_h(x')$. 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 $\mathsf{MD}_h(x)$ and $\mathsf{MD}_h(x')$? 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. ■

to-do *Discuss PGV constructions of compression functions from block ciphers. Will have to introduce ideal cipher model, though.*

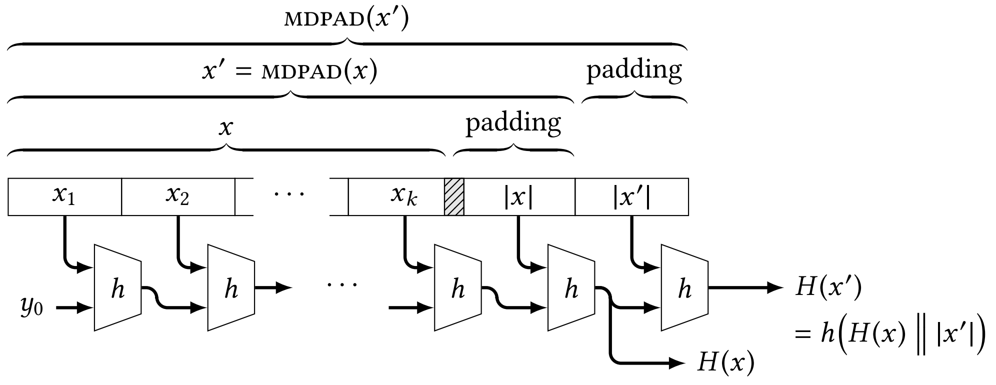


Figure 12.1: *Length-extension attack on the Merkle-Damgård construction*

12.4 Length-Extension Attacks

We showed that $\text{MAC}(k, H(m))$ is a secure MAC, when MAC is a secure MAC for n-bit messages, and H is collision-resistant. A very tempting way to construct a MAC from a hash function is to simply let $H(k\|m)$ be the MAC of m under key k.

Unfortunately, this method turns out to be insecure in general (although in some special cases it may be safe). In particular, the method is insecure when H is public and 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 $\textsc{mdpad}(x)$.*

In more detail, suppose H is a Merkle-Damgård hash function with compression function h. Imagine computing the hash function on both x and on $x' = \textsc{mdpad}(x)$ separately. The same sequence of blocks will be sent through the compression function, except that when computing $H(\textsc{mdpad}(x))$ we will also call the compression function on an additional padding block (encoding the length of x'). To compute the value of $H(\textsc{mdpad}(x))$, we don't need to know anything about x other than $H(x)$!

The idea applies to any string that begins with $\textsc{mdpad}(x)$. That is, given $H(x)$ it is possible to predict $H(z)$ for any z that begins with $\textsc{mdpad}(x)$. This property is called **length-extension** and it is a side-effect of the Merkle-Damgård construction.

Keep in mind several things:

- The MD construction is still collision-resistant. Length-extension does not help find collisions! We are not saying that x and $\textsc{mdpad}(x)$ have the *same* hash under H, only that knowing the hash of one allows you to predict the hash of the other.

- Length-extension works as long as the compression function h is public, even when x is a secret. As long as we know $H(x)$ and $|x|$, we can compute $\textsc{mdpad}(x)$ and predict the output of H on any string that has $\textsc{mdpad}(x)$ as a prefix.

Going back to the faulty MAC construction, suppose we take $t = H(k\|m)$ to be a MAC of m. For simplicity, assume that the length of $k\|m$ is a multiple of the block length.

Knowing t, we can predict the MAC of $m\|z$, where z is the binary encoding of the length of $k\|m$ (i.e., $k\|m\|z = \textsc{mdpad}(k\|m)$). In particular, the MAC of $m\|z$ is $h(t\|z')$ where z' is the binary encoding of the length of $k\|m\|z$.

Importantly, no knowledge of the key k is required to predict the MAC of $m\|z$ given the MAC of m.

to-do *Work through an example*

to-do *Discuss alternatives to Merkle-Damgård. SHA-3 winner Keccak uses sponge construction and supports $H(k\|m)$ as a secure MAC by design.*

Exercises

12.1. 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.

12.2. 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.

12.3. I've designed an n-bit hash function. One of my ideas is to make $h(x) = x$ if x is an n-bit string (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?

12.4. 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.

This means that if $\mathcal{H}$ is a collision-resistant hash family then $\mathcal{H}^{(t)} = \{H^{(t)} \mid H \in \mathcal{H}\}$ must also be collision-resistant.

12.5. Generalize the Merkle-Damgård construction so that it works for arbitrary input lengths (and arbitrary values of t in the compression function).

12.6. Let F be a secure PRF with n-bit inputs, and let $\mathcal{H}$ be a collision-resistant hash function family with n-bit outputs. Define the new function $F'((k,H),x) = F(k,H(x))$, where we interpret (k,H) to be its key ($H \in \mathcal{H}$). Prove that F' is a secure PRF with arbitrary-length inputs.

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

1. 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.
2. 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.
3. 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.