Chapter 3: Authenticated storage, Hash functions and MACs

Suggested reading:

- Security Engineering Chapters 5.3.1, 5.6

Advanced reading:

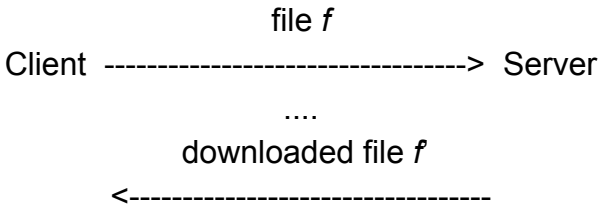
- GCAC 8 (Hash functions)

Bonus reading:

- The first SHA-1 collision (2017)
- More practical SHA-1 collisions (2019)
- SHA-256 visualization
- SHA-3 visualization

One more essential primitive to introduce is the **cryptographic hash function**. It's perhaps the most widely used primitive in all of cryptography. Hash functions are the workhorse underlying almost every protocol from encryption to cryptocurrency.

We'll start with a simple application, a client uploading a large file to a cloud server:



The client doesn't care about keeping the file secret, but wants to be sure that later when it downloads the file, it receives the same file $f'=f$. Obviously the client could just store the entire original file, but then why use a cloud server?

Instead, we'll have the client **store the hash of the file** $H(f)$, and then later check if $H(f)=H(f')$. Hash functions are similar to PRFs in that the output looks somewhat random. Hash functions only take one input, not two and that input can be infinitely long. That's right: hash functions are defined as a function $H: \{0,1\}^* \rightarrow \{0,1\}^t$, they take any number of bits as input and then produce a t-bit output, sometimes called a **digest**.

Our file-storage protocol will be secure as long as the server can't find any other file with the same hash. This property is called **second-preimage-resistance** and is defined with the following game:

1. The challenger randomly chooses x and sends it to the adversary.
2. The adversary generates a value x'.

3. The adversary wins if H(x)=H(x')

You should be able to convince yourself that if this property is true, the above file storage protocol is secure. This approach is widely used in practice. For example, official releases of software packages like Ubuntu Linux often publish the hash of the authentic version of the software. If you obtain the hash of the desired file securely, you can then download the (often quite large) file from any untrusted service like BitTorrent and be sure you've obtained the correct version by checking its hash.

Cryptographic hash functions are also designed to be one-way, but the definition is quite similar to that for one-way functions as above so we'll skip it here.

Hash functions and collision-resistance

Cryptographic hash functions are actually designed to ensure an even stronger property, **collision-resistance**, defined in the simplest of all security games:

1. The adversary outputs values x,x'.
2. The adversary wins if $x \neq x'$ and H(x)=H(x').

To break collision-resistance, the adversary simply has to find any two inputs that have the same hash (they "collide" under this function). The challenger doesn't do anything in this game, the hash function H itself is essentially the challenge.

Note that collisions must exist by the pigeon-hole principle, since the input space is infinite and the output space is finite (all t-bit strings). If we simply start hashing random values and storing the results in a table, eventually we will find a collision.[1]

In fact, for a hash function with t-bit outputs, we expect to find a collision after about $2^{t/2}$ trials due to the **Birthday Paradox**. This type of exhaustive search is called a **birthday attack**. We can defend against this attack by choosing a larger t. To defend against an attacker that can do 2^t operations, we need $2t$ bits of output.

Relation between properties and security reductions

There's an intuitive explanation for why collision-resistance is a stronger property than second-preimage-resistance. In the second-preimage-resistance game, the challenger

[1] This naive collision-finding approach requires a lot of storage in addition to requiring many evaluations of the hash function. It's possible to find collisions with only a small amount of storage by adapting cycle-finding techniques like Floyd's "tortoise and hare" algorithm.

picks x and then the adversary then has to find $x' \neq x$ such that $H(x)=H(x')$. In the collision-resistance game, the adversary gets to pick both x and x' (subject only to the constraint that $x' \neq x$). The collision resistance game is easier to win for the adversary since they have an extra degree of freedom (a free choice of x).

In fact, we can show that *any* collision-resistant hash function is *guaranteed* to be second-preimage-resistant, but not necessarily the other way around. This is a good opportunity to outline our first **security reduction**: showing that some desired property is true if a simpler assumption is true. The security and cryptography literature frequently calls these *security proofs* or (even worse) *proofs of security* but this text will aim to avoid both terms. The word *proof* makes people think security is assured, when typically all we can do is *reduce* security to well-defined assumptions.[2]

Cryptographers would say that second-preimage resistance **reduces to** collision-resistance, or equivalently that collision resistance **implies** second-preimage-resistance. Implication is always the reverse of reduction. It's important to remember which direction the two go: if A reduces to B, then A is always true if B is true (note that A may still be true even if B isn't). Similarly if B implies A, then then A is always true if B is true.

Intuitively, we can see that a collision-resistant hash function must be second-preimage-resistant, since second-preimages are a specific type of collision. So if collisions are infeasible to find then second preimages must also be infeasible to find.

A more formal way to prove this relation is to demonstrate that, given **black-box access** (or **oracle access**) to an algorithm A which wins the second-preimage finding game, it's trivial to design an algorithm A' which wins the collision-finding game. We can show this algorithm for collision-finding in pseudocode as follows:

```
def find_collision(H):
      while True:
            x1 = random()
            x2 = find_second_preimage(H, x1)
            if (x1 != x2):
                  return (x1, x2)
```

[2] There are a few cases in which security can be proven mathematically with no computational assumptions. An example is *one-time pad encryption* which we'll see. This *unconditional security* or *information-theoretic security* is rare though.

The algorithm A' (`find_collision`), chooses a random value *x* and asks the algorithm A (`find_second_preimage`) to find a second preimage for *x*. If A succeeds in finding some other value $x' \neq x$ such that $H(x)=H(x')$, then A' outputs *x', x* as a collision and stops. Since A will find such an *x*' with non-negligible probability (by definition, since it can win the second preimage game), A will therefore win the collision-finding game with non-negligible probability.

Establishing these types of relations between assumptions is very important in cryptography and we'll see several examples.

Because collision resistance implies second-preimage resistance, cryptographic hash functions are almost always built with collision resistance as the goal (with second preimage-resistance coming "for free"). As soon as any collisions are demonstrated in a hash function, cryptographers will call the hash function **broken** and consider it deprecated for further use. Even though it might still be second-preimage-resistant, conservatively it is best to move on once collisions are found.

An even stronger model for hash functions is the **random oracle model**. This is somewhat similar to the concept for PRFs. A true random oracle function would choose a completely random (but consistent) output for any given input. This is impossible to actually construct, since the function table would be infinitely large, but it's a useful conceptual model. A random oracle is collision-resistant by definition: if the function table is truly random, there can be no strategy for finding collisions better than brute force searching. Brute force is always a possibility of course, and will not succeed with greater than negligible probability as the size of the hash function output grows.

The pseudorandom nature of a hash function also explains why it is called a "hash" function. The randomness and one-wayness is similar to cooking hash browns: after hashing the input (potatoes) you can't recover the input from the output (hash browns).

Building a hash function

Designing a hash function (or any cryptographic primitive) is extremely challenging and error-prone. It takes many years of careful design work and review, and even after that many algorithms are found to be insecure.

Many designs share a common strategy, though: take a simpler collision-resistant **compression function** that takes fixed-size inputs and build an arbitrary length hash function out of that. The most common design template is called **Merkle-Damgård** and

it works as follows. Assume F is a collision-resistant compression function that takes 2*t* bits of input and produces *t* bits of output. Divide the input up into *t*-bit blocks x_0, x_1, ... and compute the output as follows:

$y_0 = F(IV, x_0)$
$y_1 = F(y_0, x_1)$
...
$y_{n-1} = F(y_{n-2}, x_{n-1})$
$y_n = F(y_{n-1}, x_n)$
$y = finalize(y_n)$

The value IV is a fixed **initialization vector**, which is fixed as part of the definition of F. We'll ignore details like how IV is chosen, how to pad the input to be a multiple of *t* bits, and what the finalize function does. The important takeaway is that a hash function is built from a simpler component. There are also formal security proofs stating that if F is collision-resistant, then the resulting hash construction is collision-resistant. We'll talk a bit more about how a compression function like F is designed later.

Practical hash functions

In the early days of the internet, the most common hash function was called **MD5**. MD5 only outputs 128 bits, which is too small as today's computers are powerful enough to try 2^{64} values to find a collision by brute force. Even worse, there are many algorithmic weaknesses to MD5. Collisions can now be found in a matter of minutes. It has been deprecated since the mid-2000s.

SHA-1 (Secure Hash Algorithm) was a standard hash function for most of the 2000s and into the 2010s. SHA-1 outputs 160 bits, and a 2^{80} exhaustive search is still prohibitive. But after many years of warnings about weakness in SHA-1's design, researchers published the first collision in 2017. This collision took thousands of years of CPU time (roughly 2^{67} values were searched) and cost around a million dollars to find (donated by Google). A new approach in 2019 reduced the amount of effort required substantially (roughly 2^{63} values were searched), now estimated to be about $100,000 worth of CPU time. SHA-1 collisions still take a lot of work to find . But the existence of these attacks is enough that nobody should use SHA-1 in any new applications and it should be retired promptly from legacy applications. Unfortunately, there are many: SHA-1 is still baked into PGP, DNSSEC, git, and many other security-critical protocols. It will take many years before SHA-1 is no longer in use.

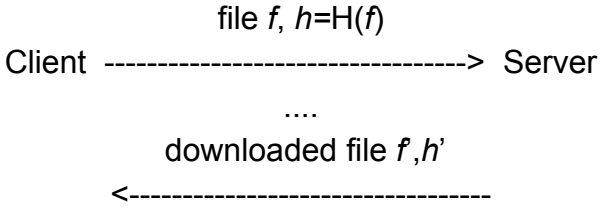
SHA-2 is now the most common hash function and has been since its introduction in 2001. It is actually a family of hash functions with several varieties named for their output length. The most common is SHA-256 (which produces 256 bits of output). SHA-512 is also used sometimes (with 512 bits of output); SHA-384 is defined but rarely used. Although the SHA-2 design is similar to the broken SHA-1, there are so far no serious signs of weakness, despite a significant amount of research. But remember: there is no proof whatsoever that collisions can't be efficiently found. All we know for sure is that no collisions have been published to date.

Finally, there is already a standard for **SHA-3** as an eventual replacement for SHA-2. SHA-3 was chosen in 2015 as the winner of a public contest to design a SHA-2 replacement. Like SHA-2, SHA-3 has variants with 256, 384 and 512 bits of output,[3] though SHA-3's design is fundamentally different from SHA-2. Instead of the Merkle–Damgård structure, SHA-3 uses a different structure called the *sponge construction* (which also builds a hash function taking unlimited input from a smaller fixed-input permutation function). Because of the new structure, it is very unlikely that any weaknesses discovered in SHA-2 would affect SHA-3 (or vice-versa). So far SHA-3 has seen limited deployment, but it is great to have as a reserve in case problems arise with SHA-2. It also has some advanced features-for example, SHA-3 is not limited to producing a fixed-size output, it can actually produce as much output as is needed.

Ensuring integrity with a message authentication code (MAC)

Let's go back to the example we started with: outsourced storage. Imagine that a client wants to store a file *f* on a server and later be sure it is receiving the same file, but *not store the hash* of the file locally. For example, the client may be storing many files and not want the burden of remembering the hash of all of them.

A broken idea is to ask the server to store the file and its hash:



[3]In fact, SHA-3's sponge construction can produce *unlimited* output like a pseudorandom generator, though the standard limits this to only the fixed amounts described.

After downloading, the client checks that h'=H(f'). This is broken of course, because a cheating server can just change the file, recompute h'=H(f'), and the client will be none the wiser.

But this suggests an avenue for solving the problem: what if we use a function H that the server is unable to compute? The client could try designing a secret hash function and not telling the server what it is, but this would be a flagrant violation of Kerckhoffs's principle. We should assume the server knows H (for example, the client may be open source software).

Instead, we should use a public function but add a secret input k (which we'll again call a key). The protocol now looks like this:

file f, t=MAC(k,f)
Client ----------> Server
....
downloaded file f',t'
<----------

This function is called a **MAC**, for **message authentication code**, and its output is a fixed-length **tag**. Like a hash function, a MAC can take any amount of input and produces a short fixed-length t-bit output:

$$\text{MAC}(k,m)\text{: F(): } \{0,1\}^{\lambda} \times \{0,1\}^{*} \rightarrow \{0,1\}^{t}$$

The security requirements here, at a high-level, are two-fold:

1. The server should not be able to learn the secret key k from seeing many outputs of the MAC function.
2. The server should not be able to compute tags for new messages, even if it has seen many message/tag pairs before.

The first security property is very much like the one-way property for a one-way function or PRF, so we'll skip it here. The second is defined with a new security game:

1. The challenger generates a random key k.
2. The adversary may submit a series of messages m_i to which the challenger responds with MAC(k, m_i).
3. The adversary outputs a message m_* and a tag t_*.
4. The adversary wins if MAC(k, m_*) = t_* and m_* is not equal to any of the queries m_i.

Note that the adversary is just trying to guess the MAC output for any message for which it has not previously seen the correct output. This message doesn't need to have any special structure. This is called an **existential forgery** (it just needs to exist, not have any other properties).

Output length: Since we're using a hash function, one might assume 256-bit outputs are required again due to the Birthday Paradox. However, collision-resistance is not a security goal of a MAC function. The adversary can't evaluate the function without the key, so it isn't possible to do an exhaustive search for collisions. The primary risk is an adversary guessing an output randomly. 128-bit outputs are long enough to negate this risk and are the standard choice.

Constructing a MAC

MAC functions have a few things in common with hash functions. Most notably, they take an arbitrary amount of input and produce a fixed-length output. You might think building a MAC is as simple as just "hash the message plus a key," and it almost is. The most popular construction for MACs is called HMAC, which builds a MAC from a cryptographic hash function as follows (where $\|$ denotes concatenation) :

$$\text{HMAC}(k, m) = \text{H}(k_1 \| \text{H}(m \| k_2))$$

The two keys k_1 and k_2 are just k xored with different fixed constants.

This design might look a bit mysterious-where do the fixed constants come from? Why is the hash function evaluated twice? These are tweaks which allow this construction to be a provably secure MAC for any secure hash function.

A much simpler construction is sometimes seen which is just:

$$\text{BROKENMAC}(k, m) = \text{H}(k \| \text{m})$$

It turns out this is not secure for hash functions with the Merkle-Damgård structure due to **length-extension attacks**, which includes SHA-1 and SHA-256. It actually is secure for SHA-3, but it's best to just avoid this and use the standard HMAC.

This construction is simple and secure, but it has one performance issue: parallelism is not possible when evaluating the hash. This can be fixed by instead hashing the blocks of message in a tree-like structure (called TreeMAC) but we won't go into details here.

There are other important MACs in practice, like the **Galois MAC**, which we'll discuss later when discussing authenticated encryption.

Comparing MACs to PRFs

Both MACs and PRFs are functions computed with a secret key which are designed to be unpredictable. As defined here, MACs take an unbounded input whereas PRFs take a fixed-size input, but this is a minor difference.

If we compare the security game for a MAC to that for a PRF, we see that a MAC is actually much weaker: with a PRF the adversary cannot distinguish the PRF output on a chosen input from random, but with a MAC they simply can't compute the *exact* output on a chosen input. Any PRF is therefore a valid MAC.

We can show a security reduction that if F is a PRF, then F is also a MAC (over fixed-size inputs at least). Another way of saying this is that a MAC reduces to a PRF. We can prove this reduction by contradiction. Assume a function F is a PRF but not a MAC. This means we assume an adversary A exists which is capable, with non-negligible probability, of winning the existential forgery game (the MAC security game) against F. We can show how to use algorithm A to win the PRF game with probability non-negligibly greater than one half.

To do so, we construct a new adversary A' which wins the PRF game using A as a subroutine. A' plays the PRF game with a challenger, while also acting as a challenger and playing the MAC game with A. Whenever A makes a query in the MAC game, A' forwards the query to its challenger in the PRF game (and returns the answers back to A). Finally, A attempts to output a forgery (m_*, t_*) on a new value m_*. A' forwards the value m_* to the challenger as one final query, then checks to see if the response it gets from the challenger is equal to A's output t_*. If it is, A' outputs a guess of 1, otherwise 0.

Now we can consider two cases:

- if b=0 and A' is receiving random responses from the challenger, then A will only output a correct pair (m_*, t_*) with exponentially small probability (since it would need to guess the output of a truly random function). Thus, A' almost always correctly outputs 0 in this case.
- If b=1 then A will output a correct pair (m_*, t_*) with non-negligible probability, by assumption. Thus A' will correctly guess b=1 with non-negligible probability

Overall, A' will (almost) always be correct if b=0 (half the time minus a negligible probability) plus it will be correct if b=1 whenever A wins the MAC game (a

non-negligible probability). This gives A' a total probability of success non-negligibly greater than 1/2.

Therefore, the existence of A implies that F cannot be a PRF, meaning our assumption was invalid. By contradiction, we've proved that any PRF is a MAC.

Strong and weak assumptions

The above reduction means that any PRF is guaranteed to be impossible to guess output for. Therefore we say the PRF's indistinguishability-from-random is a **stronger property** than the weaker property of unforgeability.

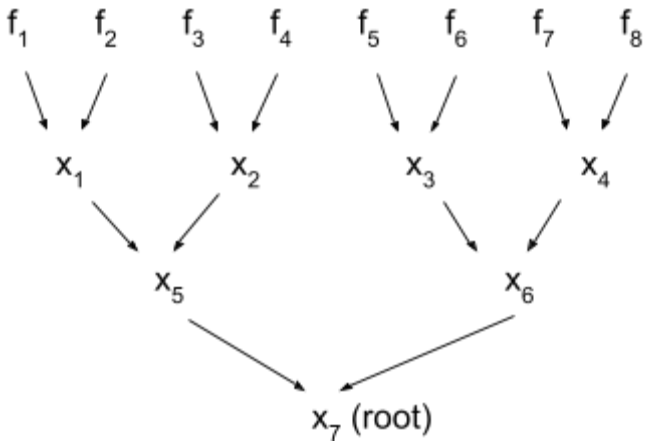
It's useful to prove results that a stronger property implies a weaker property (or equivalently that a weaker property reduces to a stronger property). It's perfectly okay (and quite common) to use a property that is stronger than what we need. Going back to our original door-opening example, we could have just used a MAC instead of a PRF. Even if a PRF is overkill, we know it's a secure choice.

Merkle trees and vector commitments

Going back to the file storage example one more time, let's consider a slightly different scenario. Imagine again that the client wants to upload many files (or many chunks of a file) to the server, but only store a constant amount of data used to verify if the server has returned the correct data.

Storing a MAC along with each chunk and storing only the key with the client is one approach. Another approach is to use a simple hash function, arranging the files in a special structure called a **Merkle tree** (after its inventor, Ralph Merkle).

Imagine a client wants to upload a series of files $\{f_1, f_2, f_3, \ldots f_n\}$. They are arranged into a tree as follows:



The arrows in this diagram indicate hashing. For example, the intermediate node x_1 is computed as $H(f_1 \| f_2)$ and the node x_6 is computed as $H(x_3 \| x_4)$ and so on. We say that each node in a Merkle tree **commits** to its children. It is impossible to find a different pair of child nodes that will produce the same hash, because that would be a collision. Because the tree structure is recursive, each node commits to its entire left and right subtrees and the root node commits to the entire structure.

When a client downloads a file, say f_4, the server can prove it is correct by providing a **Merkle inclusion proof**. The proof consists of all neighbors on the path from f_4 to the root. For example, to prove that transaction f_4 is included in the tree above, the server would provide the following as a proof:

- f_3 which allows the client to recompute x_2
- x_1 which allows the client to recompute x_5
- x_6 which allows the client to recompute the root x_7.

The client then checks the recomputed root against their stored value. This proof will require $\Theta(\lg n)$ items for a tree with n data items, and take $\Theta(\lg n)$ hashes to verify.

The root of a Merkle tree commits to all of its elements in a fixed order. Thus Merkle trees are an example of a **vector commitment**, committing to the vector $(f_1, f_2, \ldots f_n)$. Other vector commitment schemes are known with different trade-offs and properties, though Merkle trees remain the most common approach.

General authenticated data structures

Notice that a Merkle tree is really just a binary tree like you may have learned in a data structures class. The only difference is that instead of each node having a pointer to its child nodes' address in memory, each node is the hash of its child nodes.

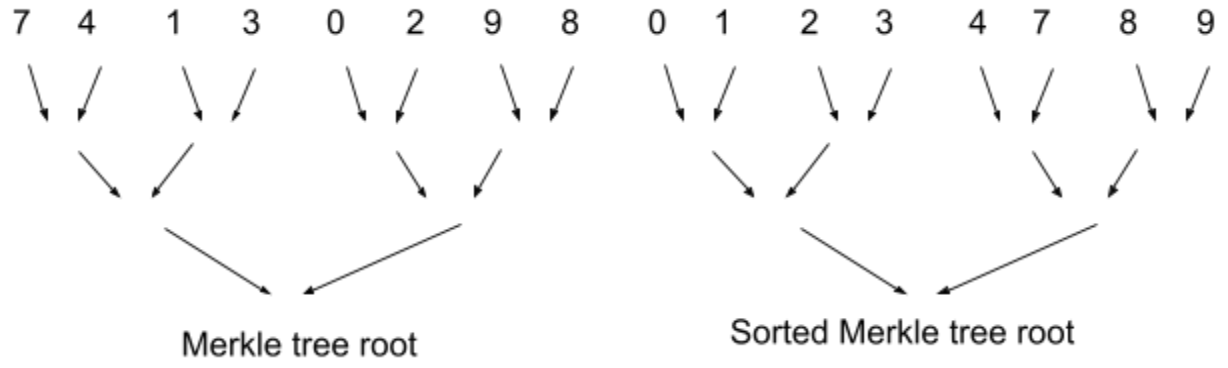
The basic idea behind Merkle trees is quite powerful. We can actually take any classic pointer-based data structure and replace the pointers with hashes to get an **authenticated data structure**. A Merkle tree is really just an **authenticated binary tree**. Blockchains, which we'll learn about later, are really just **authenticated linked lists**. The full Bitcoin blockchain is a hybrid of the two: a linked list of blocks, each block containing the root of a Merkle tree of transactions (typically a few thousand transactions per block). Other authenticated data structures exist (such as authenticated skip lists) but these two are by far the two most common.

The only restriction is that authenticated data structures must be **acyclic** in their references. A singly linked list works, but a doubly linked list is impossible, since we can't find two blocks that both include the hash of the other.

Sorted Merkle Trees, non-membership proofs and vector commitments

Authenticated binary trees provide one approach for generating proofs of inclusion in $\Theta(\lg n)$ time. But what about non-inclusion, that is, proving that some value f is not in the tree? A simple approach is to provide all n members of the binary tree as an $\Theta(n)$-sized "proof." The verifier checks that none are equal to f and then re-computes the root of the tree from scratch (requiring $\Theta(n \lg n)$ time) to confirm that the correct n values were sent. Alternately, an inclusion proof could be provided for all n elements (a total proof size $\Theta(n \lg n)$) allowing the verifier to check that each is not equal to f and that all inclusion proofs are valid (also requiring $\Theta(n \lg n)$ time).

These approaches are not very efficient. They're worse, in fact, than just hashing the list of elements directly. If there is no ordering to the tree, they're the best option possible. However, we can greatly improve this by using an authenticated binary search tree with the leaves in a sorted order (e.g. lexicographic). The result is a **sorted merkle tree**:



Ordering allows us to know exactly where a value would appear in the data structure if it were included (rather than an unsorted Merkle tree where the data could be anywhere). To prove non-inclusion, it suffices to show that the value isn't in this place where it would have it to be. Specifically, to show that a value f is not in a sorted Merkle tree, it suffices to provide two inclusion proofs for *neighboring* elements f_{left} and f_{right} such that $f_{left} < f < f_{right}$. In the example above, to show that the value 6 is not included in the sorted Merkle tree, an inclusion proof is provided for the neighboring values 4 and 7 (which satisfy $4 < 6 < 7$).

These inclusion proofs are just like any other for a Merkle tree. The verifier knows they are neighboring by the pattern of right and left neighbors that each provides to get to the root. This non-inclusion proof is twice as large as an inclusion proof in the worst case, but still $\Theta(\lg n)$. It will often in fact be much less than twice as large as the two inclusion proofs will share many elements if the neighbors are close. In the example above, in fact, the non-inclusion proof for 6 is the exact same proof as the inclusion proof for 4 (or 7) would be, since they share a subtree of size 2. The worst case would be a non-inclusion proof of 3.5. But these details don't matter too much, generally we're happy that the asymptotic size is $\Theta(\lg n)$.

Another interesting property is that the non-inclusion proof for 6 in our example would be a non-inclusion proof for 5 as well. In fact, non-inclusion proofs with sorted Merkle trees function as proofs of non-inclusion of the entire range (f_{left} , f_{right}).

A downside of sorted Merkle trees is that they are no longer vector commitments, because the elements cannot be committed to in an arbitrary order (they must be included in sorted order). For example with an unsorted merkle tree you could commit to the vectors <5, 7, 2, 6> or <6, 2, 5, 7> which will produce different trees with unique roots. Whereas in a sorted Merkle tree you can only commit to the set {2, 5, 6, 7}. Sorted Merkle trees are an example of **set commitments**, committing to an unordered set of elements.

There exist other constructions of set commitments as well which are not vector commitments (such as RSA accumulators). There are also even more powerful authenticated data structures, such as map commitments (commiting to a mapping like {1: "cat", 3: "dog", 8: "chinchilla"} or polynomial commitments (committing to a polynomial function, and proving its evaluation on arbitrary points).

Another important caveat with a sorted Merkle tree is that the verifier must at some point verify that all of the elements are indeed in sorted order, otherwise a malicious

prover might be able to show both inclusion proofs and non-inclusion proofs for the same element. For example, if the verifier believed the unsorted Merkle tree in the example above were actually a sorted Merkle tree, it's easy to show the inclusion of 2 and also the non-inclusion of 2 (because elements 1 and 3 are neighbors).

In real-life, perhaps the verifier originally computed the root itself and then uploaded it to an untrusted server. Or perhaps the verifier trusts that some third party has, at some point, verified that the entire data structure is sorted.