

Cryptographic Engineering

working draft

Matthew D. Green
Johns Hopkins University

Excerpt: Chapter Chapter 6, Hash functions and commitment schemes

§6.1 Hash functions, §6.1.1 Building hash functions; §6.2 Commitments; §6.3
Message authentication codes (book pages 99-112,121-128)

Posted for 601.445/601.645 Practical Cryptographic Systems, Fall 2026
Lecture 5, Wed Sep 16: hash functions and message authentication (updated Sep 21 with
the new §6.3)

<https://practicalcrypto.org>

This is an excerpt from an unfinished book draft: expect gaps, rough edges, and errors, and note that section numbers and page numbers may change in later drafts. Cross-references to other chapters point to material that is not included here. Copyright © 2026 Matthew D. Green. All rights reserved. Posted for the students of the course above; **please do not redistribute** — link to the course site instead. Built 2026-09-21.

Chapter 6

Hash functions, MACs and commitment schemes

So far we have focused on symmetric encryption, which is one of the earliest uses of cryptography. While encryption helps to ensure the confidentiality of data,

6.1 Hash functions

Hash functions are enormously important to cryptography. We use them to build digital signatures, encryption schemes, key derivation, and MACs — and this list is just a start. In fact, most cryptographers use hash functions so often that we don't spend much time thinking about their internals. Doing this is like asking where your fast food comes from: sometimes it's better not to know. More to the point, the science of designing new hash functions is so intricate and specialized that it's essentially a field of its own. So most cryptographers gratefully accept standardized hash functions like SHA2, SHA3 and Blake2.

But the construction of hash functions *is* important. It's particularly important because there's a lot riding on them. While this section won't be able to answer all of those questions, they should help us get a better understanding of what we're counting on.

What's a hash function?. In computer science, a hash function is a mathematical function, one realized by an efficient algorithm, that maps input strings into output values called *digests*. In general, most standard cryptographic hash functions take an arbitrary-length string as input, and produce a fixed-length k -bit digest, where k is typically some value on

the order of $256 - 512$ bits¹ We usually denote this by the function definition:

$$H : \{0, 1\}^* \rightarrow \{0, 1\}^k$$

Outside of cryptography, *non-cryptographic* hash functions are regularly used to implement data structures such as hash tables, which allow quick lookups in key/value stores. The problem with those hash function is that they’re *not* designed to resist adversarial attacks. In this book we’re going to focus exclusively on cryptographic hash functions, except where explicitly mentioned. That means we need to spend some time talking about the type of security properties we need.

Collision-resistance. The most important property we require from all cryptographic hash functions is *collision-resistance*. As we’ve discussed earlier in this book, this property requires that it should be “hard” to find two different inputs $M_1 \neq M_2$ such that $H(M_1) = H(M_2)$.

What does “hard” actually mean? If you ask a theoretical cryptographer to define collision-resistance, they’ll rattle off something like “no efficient algorithm should be able to find a collision pair M_1, M_2 except with negligible probability.”² Then they’ll look at your unkeyed hash function (say, SHA3) with a haughty sniff, and inform you there’s no way it meets that definition.³ And they’ll be right! There are two reasons for this. First, we know that there certainly *exist* collisions in functions like SHA3: this must be true due to the fact that there are more possible inputs than digests (the *pigeonhole principle*, see §4.3.1.) And if there exist collisions, then there must also exist polynomial-time algorithms that embed at least one of those collision pairs as a hard-coded value. Ergo, in the mathematical sense at least, there *exist* algorithms that can output collisions in SHA3! The only thing protecting the Internet is that we (humans) haven’t found them just yet.

If you’re annoyed and unsatisfied by this line of argument, welcome to the club. An easy fix is *don’t talk to theoretical cryptographers*, which is (sadly) what most practitioners have done. But there’s a better answer. In practice, a much cleaner way to phrase this definition is to use our common-sense understanding: namely, that collisions in standard hash functions exist, and hence algorithms that output those collisions must exist as well, but that hash functions are safe to us as long as we (humans) haven’t found any yet. The converse is this: if a scheme that uses some hash function becomes insecure, then it must be because a

¹While this is a general profile for most standardized functions, the cryptographic research literature is full of specialized hash functions that have different profiles: some take in specific data types, while others require an additional *key*. At the end of the day, many real-world implementations likely implement these by feeding the input into a standardized hash function. But a few special hash functions actually require inputs of a specific form: we’ll talk about those later on.

²As usual, *efficient* means “probabilistic polynomial-time in some security parameter like k ”, and *negligible* also refers to a function that’s negligible in the security parameter.

³In real life all the theoretical cryptographers I know are incredibly nice and humble and never act like this, but I still really enjoy this caricature.

real-world attacker actually *knows* one of those collisions. Some modern definitions formalize this mathematically [277].

Pre-image resistance. A second property of cryptographic hash functions is *pre-image resistance*. What this means is that it should be hard to invert a hash function. More concretely, given $h = H(M)$ it should be hard to find (any) M that hashes to h (called a *pre-image*).⁴ Even this simple definition contains a lot of detail. If M is drawn from a small set of possible inputs, then *of course* we can find the inverse $H(M)$: we just systematically try every candidate $M_1, \dots, M_n$ until we find one where $H(M_i) = h$. This is called a brute-force search or dictionary attack.

To avoid this pitfall, we usually assume that M is drawn from some larger set, where simply brute-forcing your way through the candidates isn't possible. For example, if M is a random string of length λ bits, where λ is something like $128 - 256$, then it's pretty unlikely that an attacker will be able to invert $H(M)$. In formal mathematical definitions, pre-image requires that the input literally be a random string. As a more general rule, it's reasonable to assume that pre-image resistance holds as long as the space of possible input messages is much too large to be practically brute-forced.⁵

Second pre-image resistance. A related notion is *second pre-image resistance*. Here the attacker is given some message M_1 and must find M_2 such that $H(M_1) = H(M_2)$. You might notice that this pair M_1, M_2 is a collision, and you might then be inclined to ask: *wait a second, didn't collision-resistance already cover this?* The answer is that it does. The main reason we treat second pre-image resistance as a separate problem is it's even stronger than collision-resistance. In normal collision-resistance, the attacker wins if they find *any* pair that collide. In second pre-image resistance we're pinning the attacker down to a specific choice of M_1 , and they have to find a value M_2 that collides with the message we chose. The point of this security notion is that it's *strong*. Even if an attacker breaks the collision-resistance of a hash function, a cryptosystem that only depends on the hash function being immune to second pre-image resistance might still hold up.

Random oracles. While all of the properties above are commonly taught in cryptography courses, many cryptographic schemes and protocols need even stronger properties from hash functions. If you ask a novice cryptographer to describe what they want from a hash function, they'll usually say something like “*the output of the hash function should look random.*” This isn't a very scientific statement, since hash functions are the farthest thing from random: in fact, they're completely deterministic! What we're really saying is that the output of a hash function, on some new input we haven't hashed before, should *look* effectively

⁴Note that it doesn't even matter if we find the same M . Finding any value (even a different value M') that hashes to $H(M)$ is considered a “win” for this purpose.

⁵For formal definitions, see *e.g.*, [283].

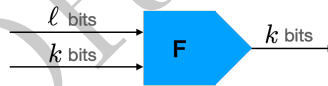
like random bits. In proofs we will often *model* this requirement formally using something called the *random oracle model*, which assumes that hash functions behave like truly random functions ???. As we’ll discuss in §??, this model is as theoretically sound as a bag of cats. No worries, we rely on it anyway.

6.1.1 Building hash functions

Now that we’ve discussed the requirements of a hash function, it’s time to dig just a bit deeper into the internals of the hash functions we actually use. For the moment, we won’t dig all the way to the center of the onion: we’re going to just peel off one more layer. Most standard hash functions follow one of two different paradigms. Both are designed to take a relatively “simple” fixed-length function and turn it into a variable-length hash function.

Merkle-Damgård

Many standardized hash functions up through SHA2 were built using the same approach, which is independently credited to Ralph Merkle and Ivan Damgård [226, 227, 96]. Here we require one basic cryptographic ingredient, which we call a *compression function*. Don’t be misled by the word “compression”: it doesn’t imply reversible compression like a ZIP file! In fact, compression functions are explicitly designed to be irreversible, or one-way. The main requirement of a compression function is that it takes in more bits as input than it produces as output. Let’s define our compression function F to take in $(\ell + k)$ bits as input, and to produce a k -bit output as illustrated here:



Internally, the ingredients of this function will often look like a cipher. In fact, standardized hash functions like SHA2 are literally built from special-purpose block ciphers, even if we don’t generally use those ciphers anywhere else.

Depending on what properties you want from the overall hash function, we’ll need to demand some security properties from this compression function F . At minimum, if our goal is to build a collision-resistant hash H , then F itself must be collision-resistant: *i.e.*, it should be hard to find two distinct inputs $(m_1, t_1) \neq (m_2, t_2)$ where $F(m_1, t_1) = F(m_2, t_2)$. To some extent this is a cop-out, since we’re pushing all of the really “interesting” security problems down into F ! On the other hand, it means that we just need to find this one “simple” function, and we can hash a more or less arbitrary amount of data through it.

The construction. Let’s assume we’ve got a fixed-input compression function F that satisfies our requirements. The problem now is to make it work for *arbitrary-length* inputs. The Merkle-Damgård construction takes in a message $M \in \{0, 1\}^*$ and processes it using the compression function F as follows:

1. First, pad the end of the message M so that its length is a multiple of ℓ bits.
2. Add the length of the message M (in bits) so that it's contained within the final ℓ -bit chunk of the message.⁶
3. Now chunk the padded message M into n blocks $M_1, \dots, M_n$ of size ℓ bits each.
4. Set the *chaining value* $t_0 \leftarrow \text{IV}$, where IV is a fixed (non-random) Initialization Vector.
5. For $i = 1$ to n : process the next input block as $t_i \leftarrow F(t_{i-1}, M_i)$.
6. Output the final chaining value t_n as the digest.

If this description leaves you cold, a much better illustration of the process is shown in Figure 6.1.

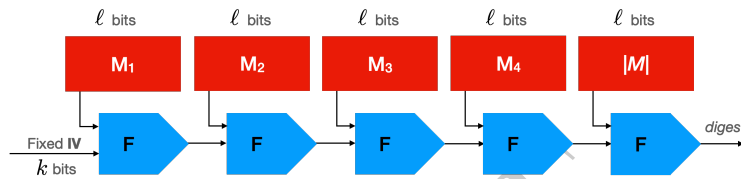


Figure 6.1: A simplified version of the Merkle-Damgård hash construction. Here the message M is chopped into blocks of length ℓ bits, padding the last block with a unique pattern such as “10...”. The bit-length of the unpadded message M is placed into a final block. These blocks are then iteratively fed into the function F along with the output of the previous iteration (called a “chaining value”), using a fixed IV for the first such value. This basic design provides collision-resistance if F is collision-resistant.

Arguing collision-resistance (for fun and profit.). It’s pretty easy to see the output of the full function H must be collision-resistant if F also is. We can argue this by considering what happens if the opposite is true, *i.e.*, that someone finds a collision in the full Merkle-Damgård function H . With a little effort, we can easily show that this *also* means that they must have found a collision in F .

Imagine that someone is able to find two different messages $M_A \neq M_B$ where $H(M_A) = H(M_B)$. Let’s consider the following possibilities:

1. If the length of the two messages is different, *i.e.*, $|M_A| \neq |M_B|$, then we must have found a collision at the final iteration of F . This happens because the final (length) blocks can’t be the same for both messages, since we put the length of the message into those blocks. That means that the final evaluation of F must have different inputs for each of M_A and M_B , but has the same output (digest.)

⁶Most implementations do these steps together: they’ll attach a unique padding consisting of a 1 bit followed by 0 bits to pad out the last block, and they also attach the length of the message immediately afterwards. In the illustration in Figure 6.1, I’ve put the bit-length of the message into the last block just to make everything as clean as possible, but performance-obsessed designers would probably object to using a whole extra block.

2. If $|M_A| = |M_B|$ (*i.e.*, both messages are n blocks long), the argument is slightly more involved. We start at the rightmost iteration of the function F and look at the inputs to F when hashing both M_A and M_B . If the inputs to F differ between the two different messages, but *the output of F is the same in both cases*, then we instantly have a collision in F . Starting from the rightmost iteration, we know the output (final digest) is different, and so we can look for differing inputs here.
3. If we determine that the inputs to F are exactly the same at iteration i , then this means the “chaining value” output by iteration $(i - 1)$ must have been the same between both M_A and M_B . Since we know that the *output* of iteration $(i - 1)$ is the same in both M_A, M_B we can look at the inputs to F at this point and see if they differ, at which point we’ve found a collision in F .

The important observation here is that if $M_A \neq M_B$ then at least one of the message blocks in some position $j \in \{1, \dots, n\}$ must be different between the messages M_A and M_B . This means that at some iteration, we’re guaranteed to find a place where the output of F (the chaining value) is identical between both messages, but the input to F is different. This means that any collision in H also means there’s a collision in F . By judo-flipping this logic around, we can argue that *if finding collisions in F is intractable* that must mean that finding collisions in H is as well.

What’s in F ? As mentioned above, the function F used in standardized hash functions like **SHA1** and **SHA2** is typically based on a specialized block cipher. This block cipher is often evaluated in a specific way, called the Davies-Meyer construction.⁷ Here the *message block* forms the “key” for the cipher, and the chaining value forms the “plaintext” input that is enciphered. Given an encipherment algorithm E , Davies-Meyer constructs the function F as:

$$F(t, M) = E_M(t) \oplus t$$

If you’re curious why we XOR the plaintext with the output of the cipher, the answer is simple: this approach makes F “one-way.” If we didn’t perform this additional operation, *i.e.*, we set $F(t, M)$ to simply be encipherment, then it would be easy for an attacker to find collisions in F . For example, given $x = F(t, M)$, the attacker could choose a new value M' and use the decipherment algorithm D to find:

$$t' = D_{M'}(x)$$

The additional XOR is surprisingly effective at blocking this attack. There are other ways to do this,⁸ it’s just that this approach is simple. For obvious reasons it’s very helpful that

⁷Inventorship of this idea seems slightly uncertain [330, 217], and even Davies has attributed the construction to Meyer. This is one of those beautiful things that happens in academic research.

⁸The “other ways” are called Matyas-Meyer-Oseas and Miyaguchi-Preneel and they’re honestly all pretty similar.

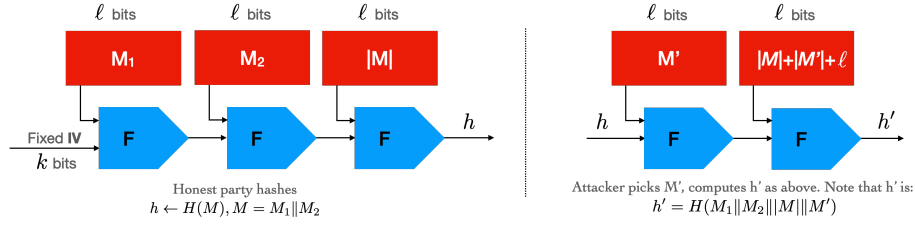


Figure 6.2: Illustration of a length-extension attack against the simplified Merkle-Damgård construction from Figure 6.1. At left: an honest party hashes a message $M = (M_1 || M_2)$. At right, the attacker uses $h = H(M)$ as a chaining value to compute a new digest on an extended version of M , where $h' = H(M || |M| || M')$. Notice that the attacker doesn't need to know M to do this, they only need to know h and the *length* of M .

the cipher **E** also features a large key size, since the cipher's key length will become the hash function's message block length. Since message blocks are used as keys, it's also very useful if the cipher does not have substantial key setup time (*e.g.*, building large key schedule tables) and if it is resistant to related key attacks. We'll briefly discuss some of the functions used in standardized hash functions further below.

Attacks on Merkle-Damgård functions. The arguments above do not mean that Merkle-Damgård functions are perfect. In fact there are several known problems with Merkle-Damgård functions in the real world, some of which can cause us a huge amount of heartburn. Just to give a few:

Length-extension attacks. An aggravating problem with basic Merkle-Damgård hash functions is a simple attack called a *length extension* [310, 268]. While this doesn't break collision-resistance, it can be bad news for many applications built with hash functions, particularly simple MACs of the form $\text{MAC}_K(M) = H(K || M)$. This attack lets an attacker who learns $h = H(M)$ (but not necessarily the message M itself) compute a new hash of the form $h' = H(M || \text{padding} || M')$ for some message M' the attacker chooses. The process is illustrated in Figure 6.2.

This attack takes advantage of the fact that the final hash digest output by many Merkle-Damgård functions is identical to the *chaining value* used within the function. That means an attacker can take the final digest and “tack on” new blocks by evaluating the function F directly. There are some ways to fix this problem, but unfortunately most standardized Merkle-Damgård hash functions like **MD5**, **SHA1** and **SHA2** were designed before these defenses were considered necessary.⁹ If you're unsure whether

⁹Just to give an example (*i.e.*, don't actually do this!), if the output of F is long enough, you can simply truncate the final output of F before emitting the digest: for example, in **SHA-384** the final output of the

length extensions are a concern, a good answer is to use a more recent “sponge” design like SHA3, which is not subject to these attacks at all.

Multicollisions. There’s a bit of a warning hiding in the collision-resistance argument for Merkle-Damgård. While it certainly means Merkle-Damgård hash functions are collision-resistant if F is collision-resistant, Antoine Joux pointed out that this argument also contains the seeds for the destruction of these functions [183]. Let’s imagine an attacker ever *does* find a collision pair in F , specifically: two distinct message blocks M_i, M'_i that produce the same output when fed the same chaining value. Then an attacker can construct many different collisions in H from one (or a small number) of such collisions. Even with one collision, it’s easy to “tack on” arbitrary new blocks following the collision to create many different message pairs that collide in the overall hash H . This means that when (simple) Merkle-Damgård functions break, they can break badly.

Merkle-Damgård in the real world. We’ve spent a bunch of time discussing Merkle-Damgård, and of course there’s a reason for this. Many hash functions, including MD5, SHA1, Whirlpool, RIPEMD and SHA2 all use this construction. We don’t have room to discuss them all, but it’s worth at least discussing SHA2.

The SHA2 family. The SHA2 family of functions was patented by Glenn M. Lilly at the NSA [202]. There are two main functions in the family, called SHA-256 and SHA-512. Some further named variants such as SHA-384 and SHA-512/256 are simply SHA-512 wearing glasses and a fake mustache, meaning that they use different IVs and the final digest is truncated to a shorter length.¹⁰ Should you care about U.S. government certifications, SHA2 is approved for FIPS 140 modules and is also part of the NSA’s Suite B [247].

For SHA256/SHA512 the compression function takes in a 256/512-bit chaining value and a 512/1024-bit message block. This core of the function is a specialized block cipher, which in the literature is nicknamed SHACAL-2 [207]. The cipher first computes a key schedule, by combining different elements of the “key” (message block) input in various ways, then processes the key and message over 64 rounds in SHA-256 and 80 rounds in SHA-512.¹¹ To obtain a non-invertible compression function F from the encipherment algorithm E , the

round function F is 512 bits, but is truncated to 384 bits. An attacker would have to guess the missing 128 bits in order to perform a simple length-extension attack. When implementing MACs, the correct solution is to use HMAC, which evaluates the hash function twice and is immune to length-extension attacks.

¹⁰I’m leaving out two further truncated variants, SHA-224 (based on SHA-256) and SHA-512/224 because they provide less than 128-bit collision security and probably shouldn’t be used in new protocols.

¹¹There are a few differences: calculations are performed on 32-bit words for SHA-256 and 64-bit words for SHA-512.

function uses a Davies-Meyer construction as follows (here $\overset{w}{+}$ indicates word-wise addition):¹²

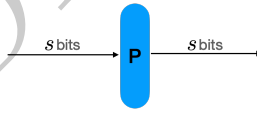
$$F(m \in \{0, 1\}^\ell, t \in \{0, 1\}^k) = E_m(t) \overset{w}{+} t$$

Every variant of the SHA2 family has its own IV. To process a message M , the end of the message is padded with a string of the form $(1\|0^p\|\text{len_bits}_{64}(M))$, where p is chosen so that the padded message is a multiple of 512 bits for SHA256, or 1024 bits for SHA-512. After processing each block, the final compression function output is optionally truncated, and becomes the final digest.

This design of SHA2 is optimized for efficiency, particularly in software. Messages can be hashed *online*, which means that the full message length does not need to be known before hashing begins (although it does need to be known before completing the hash.) The downside of the design is that many variants of SHA2 are vulnerable to practical length-extension attacks, which makes it unsuitable for some uses such as constructing simple MACs.¹³ Beyond these known attacks, no more serious attacks are known against the collision- or pre-image security of SHA2 as of this writing, and the function(s) are widely used in protocols even where strong requirements are required from the hash function.

Sponge constructions

The limitations of Merkle-Damgård have led cryptographers to develop other approaches. The most popular of these is the *sponge construction*, which is used in SHA3. Sponge functions replace the compression function of Merkle-Damgård with an unkeyed *permutation* of the form: $P : \{0, 1\}^s \rightarrow \{0, 1\}^s$:



The sponge construction initializes a state vector S of length s bits, using a fixed initialization vector (typically zero.) The state vector is subdivided into two contiguous sections: the first r bits are called the *rate*, and the second c bits section are called the *capacity*. To hash a message M , the message is first padded to a multiple of r bits in length,¹⁴ then broken into r -bit chunks $M_1, \dots, M_n$. The use of a sponge (Figure 6.3) typically involves two distinct types of operation, which are called “absorbing” and “squeezing” as follows:

¹²Specifically, $\overset{w}{+}$ means that each word of the inputs is added *mod* 2^w , where the word-length w is 32 bits for SHA-256 and 64 bits for SHA-512.

¹³An exception is the truncated versions, SHA-384 and SHA-512/256, where the truncation of the final output block makes length-extension attacks impractical.

¹⁴A common padding scheme is to append a 1 bit followed by zero or more 0 bits and then a single final 1 bit. If the smallest pad can’t fit within the existing final block, a full block of padding is added.

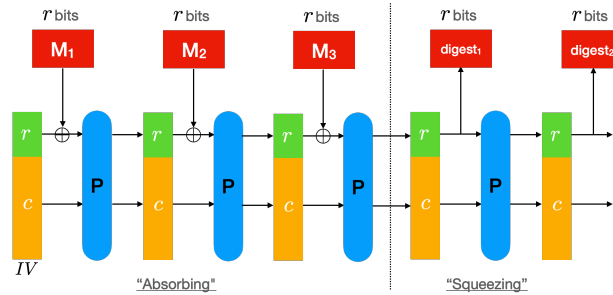


Figure 6.3: An example of the sponge construction. Each internal state block is divided into an r -bit *rate* and c -bit *capacity* section. The initial state contains a fixed IV, often zero. The full state is fed into an unkeyed permutation P at each round. In the “absorbing” phase, new input blocks are combined with the rate section at each round. During the “squeezing” phase, the rate portion is output as a digest. This example shows how multiple blocks of output can be extracted from the function; many real-world sponge hash functions like SHA3 typically require only one block of output.

Absorbing. During the absorbing phase, each message block M_i is fed into the sponge by exclusive-ORing it with the rate portion of the state vector. The capacity portion is left unmodified. The full state vector is then fed through the permutation P , resulting in an updated state vector. The use of the permutation is designed to “mix” all values of the input state in an unpredictable manner, so that every bit of the output state is a function of the input, which has also been modified by the message block fed into it. This process repeats until all message blocks have been processed.

Squeezing. Once the absorbing phase has been completed, hash output can now be extracted from the resulting state vector. This is done by copying and outputting the r -bit rate section of the state vector (or a subset of the bits), while keeping the capacity portion secret. If the digest length $k \leq r$, then only one block of output is needed (and sometimes only part of that block.) If more output is required, the entire state vector can be fed through the permutation P again, and an additional block of r bits can be extracted from the rate portion of the new state. This process can be repeated as many times as needed.

A nice feature of sponge constructions is that they aren’t vulnerable to length-extension attacks: this is because the hash function never outputs the *capacity* section of the internal state, making chain extension very difficult. In theory, this means you should be able to construct MACs using the simple construction $\text{MAC}_K(M) = H(K\|M)$ if your sponge construction is good, although NIST standards still urge the use of HMAC with functions like SHA3.

Why are we doing this? A very good question you might ask is *why* we are using this new construction. In particular, you might ask a different question: what’s the deal with using an invertible permutation, instead of the more traditional one-way compression function we saw in Merkle-Damgård? The simplest answer is that a permutation has some nice properties: it’s relatively easy to build permutations that have strong diffusion properties, without having to mess around with elements like key schedules and other elements that can expose your function to various types of linear cryptanalysis. A permutation also has the nice feature that it never “loses entropy”: if there are 2^s equally-likely states before the function P is evaluated, there will still be 2^s equally-likely states afterwards. By contrast, other types of function may produce some collisions, which causes the number of possible states to slightly decrease with each iteration. Whether or not these explanations convince you is a matter of personal taste; I’m told that they’re fairly compelling to many hash function designers, but certainly not all of them.

Security and efficiency. The efficiency and security of a sponge construction depends on the exact size of the rate and capacity portions (r, c) of the state vector. Larger values of r mean that more bits can be absorbed or squeezed during each iteration. At the same time, the security of the function depends on using a relatively large capacity section. Researchers estimate that brute-force classical (non-quantum) attacks can find collisions in the internal state with around $2^{c/2}$ work and pre-images with approximately 2^c work. This means the capacity length should be at least twice the desired security level in bits: for example, a 256-bit security level would imply $c \geq 512$, and a similar number for the final digest length. Using known quantum attacks implemented on (as yet) unknown quantum computers, collision search improves to $2^{c/3}$. To keep the rate and capacity at a useful level, this implies a fairly large total length for the internal state, typically well over 1000 bits.

Sponges in the real world: Keccak and SHA3. The most famous sponge construction is the Keccak function designed by Guido Bertoni, Joan Daemen, Michaël Peeters and Gilles Van Assche. Keccak was one of several submissions to a NIST competition intended to find a successor to SHA2 [57], and was ultimately selected to become the basis for SHA3 in 2012. During the subsequent standardization process there was a minor controversy when NIST proposed to reduce the capacity size c for some of the hash variants specified in the Keccak submission; this proposal was abandoned due to pushback from the research community [289]. As a result, the SHA3 standard is essentially identical to Keccak.

SHA3 was commissioned as a drop-in replacement for SHA2, and to provide some new features on top. Hence it supports standard hashing with a fixed-length digest (in several lengths) as well as an extensible-output function (XOF) mode that supports a variable-length output. All variants use the same fixed permutation with input size $s = 1600$ bits, based on a specific 24-round function proposed by the designers. For ordinary hashing, the standard specifies several digest output lengths: we’ll call these SHA3- k where $k \in \{224, 256, 384, 512\}$. Aside from the length of the digest, the difference between these variants is their rate and

capacity values (r, c) . Each version uses $c = 2k$ and the rate represents the remaining bits of the state.¹⁵ To ensure that the different variants produce unrelated outputs, SHA3 specifies a *multi-rate padding* structure for the end of the message. This padding simply ensures that the last r -bit block of the padded message always contains a pattern that is specific to the rate being used; since the final block is different for each value k , the outputs of SHA3-256 should be quite different from those of SHA3-384!¹⁶

Analysis. The SHA3 standard has been extensively analyzed. This analysis falls into two different categories: in one direction, the SHA3 design has been proven to be *indifferentiable*¹⁷ from a random oracle [58]! While this sounds amazing, the caveat is that this analysis requires the underlying permutation to itself be modeled as a random permutation, which (in real life) it obviously isn't. That sort of proof might not seem useful to you, but it's actually very helpful: it ensures that if there are flaws in SHA3, such as length-extension attacks, they aren't going to be simple like Merkle-Damgård length extensions; they will instead have to rely on some concrete problem within the Keccak permutation. To address this latter concern, substantial cryptanalytic work has been thrown at Keccak's permutation. I won't even try to summarize it all, but as of this writing the best known attacks appear to work against only 6 of the 24 rounds of Keccak's unkeyed permutation [298].

Other applications of sponges. Another nice feature of the sponge construction is that it's very flexible. For example, sponge constructions can be used to build *extensible output functions* (XOFs) like SHA3's SHAKE mode (§6.1.3), in which the output digest can be as long as you want. This is done by simply running the “squeezing” operation for as many iterations as needed to produce the desired output length. Some newer authenticated schemes are also based on sponges: here the absorbing phase is interleaved (or “duplexed”) with the squeezing phase, so that a key and message can first be absorbed, at the same time as a pseudorandom keystream and MAC is generated [60]. One notable scheme that does this is Ascon [107, 312], which NIST recently standardized for use in “lightweight” cryptography.

Tree hashing

One limitation of the iterative hash designs above is that message blocks must be processed *sequentially*: meaning that a first block must be processed before a second one can be. This makes it fundamentally difficult to parallelize the calculation of long hashes. An alternative approach to processing these messages is to use a *Merkle tree* data structure. While Merkle trees can in principle be constructed using any hash function, some functions (notably BLAKE3 [254]) explicit use tree hashing for their default construction.

¹⁵Note that since $r \geq k$ in all four cases, only one block of output is needed during the squeezing phase. This block is truncated to k bits.

¹⁶If you find this handwavy, so did I. See Appendix B.1 for a deeper explanation.

¹⁷See §10.3.1 for a slightly deeper explanation of what this means.

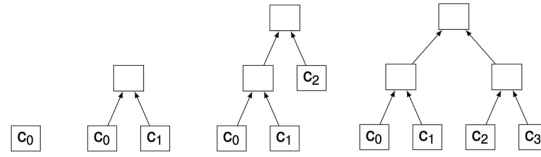


Figure 6.4: Four examples of Merkle tree hashing, from the BLAKE3 specification [254]. From left to right, messages with 1, 2, 3 and 4 chunks are shown.

To construct a Merkle tree, the message is first divided into n contiguous chunks, where the size of the chunks may depend on the function. For the moment I’ll assume the number of chunks is a power of 2, *i.e.*, $n = 2^\ell$, but I’ll relax this requirement in a moment. The hashing proceeds as:

1. Each chunk M_i is hashed using the function and some *domain separation* string that identifies it as a chunk, which we denote by $h_i^\ell \leftarrow H(\text{chunk}, M_i)$.
2. Once this has completed, the resulting hashes $h_1^\ell, \dots, h_n^\ell$ are concatenated in adjacent pairs. Each is labeled with a domain separation string indicating they are a “parent” hash at a given level $\ell - 1$, and the combined string is hashed:

$$h_1^{\ell-1} \leftarrow H(\text{“parent } \ell - 1”, h_1^\ell \| h_2^\ell), \dots, h_{n/2}^{\ell-1} \leftarrow H(\text{“parent } \ell - 1”, h_{n-1}^\ell \| h_n^\ell)$$

3. The previous step is repeated on the resulting set of hashes for levels $\ell - 2$ down to 1, until eventually only one pair h_1^1, h_2^1 remains. This final pair is hashed with a different *root* domain separator as:

$$h^{\text{root}} \leftarrow H(\text{root}, h_1^1 \| h_2^1)$$

The final hash is called a *Merkle root*. If the underlying function used to hash each previous step is collision-resistant, then the root is a collision-resistant hash of the message. The benefit of this approach is that hashing of large messages can be parallelized using multiple processors or threads; only the lightweight final steps involve sequential calculations. Real-world schemes such as BLAKE3 perform tree hashing over chunks of size 1024 bytes, and specify this as their default mode; however, other schemes support tree-hashing modes. In real life, messages will often consist of an arbitrary number of chunks n that is not a power of 2. This can be handled by using an unbalanced binary tree, as illustrated in Figure 6.4.

Even more constructions!

There are several alternative constructions for hash function, such as the HAIFA construction [64] used in the BLAKE series of functions [25, 26, 254]. Some functions like Skein use a “tweaked” compression function construction called UBI [126] where details such as the block number are fed into the function. More recently, there has been a great deal of interest

in *zero-knowledge friendly* hash functions that operate over addition and multiplication in finite fields instead of using strings and bit operations; one example is the Poseidon hash function [142]. While I could probably spend a dozen more pages on hash constructions, I don’t think you’d thank me for it.

6.1.2 Hash function security levels

A major concern in the design and usage of hash functions is the *security level* provided by the function. In most cases this refers to the number of operations required to find collisions in the function on a classical computer,¹⁸ and is usually measured in bits, *i.e.*, 128-bit security implies approximately 2^{128} operations to conduct an attack, and 256-bit security implies around 2^{256} operations. As we’ve discussed in earlier sections of this work, the *classical* collision-security of a function is upper-bounded by approximately half the function’s digest length k . This is because a stupid collision attack simply tests distinct inputs until it finds a collision, and for functions with uniformly-distributed outputs this will succeed (with high probability) after $\sqrt{2^k} = 2^{k/2}$ attempts. Of course, this only considers classical computers. Using the BHT collision-finding algorithm [80] on some future quantum computer, it’s possible to reduce the search effort from $O(2^{k/2})$ to $O(2^{k/3})$: this means that the digest size should be at least three times the desired security level to future-proof the function!

While the output size provides an *upper bound* on collision-security, it’s not necessarily the final word on the security of a given function. As we discussed above, many hash functions have internal round functions that are themselves subject to collision attacks. Most standardized hash functions compensate for this by adjusting both the internal round and state size upwards for higher security levels: this is partly why, for example, **SHA-512** operates over 64-byte internal state while **SHA-256** uses only 32 bytes. Most standards have already taken these internal attacks into account, which is why you should generally use standardized functions and not try to build your own! As a final note, some NIST standards mandate the use of more recent hash functions like **SHA3** for “high-security” applications such as Ed448 signing [249, §7.1].

6.1.3 Specialized hash functions

Keyed hash functions

Many research papers call for *keyed* cryptographic hash functions. As a general rule, keyed hash functions should not be confused with MACs. Although sometimes the primitives can be built using the exact same ingredients, you should develop the habit of treating these as different animals. The reason is that MACs have specific security requirements. This

¹⁸Hash functions have other types of attack consideration as well, such as pre-image resistance and pseudo-randomness. However, since collision-resistance is relatively easier to quantify and also typically the attack type where functions are “weakest”, security levels are almost always based on collision security.

Δ -universality and XOR-universality. As mentioned above, some MAC constructions require the stronger property of Δ -universality, which means that an attacker can't find Δ such that $h_K(M_2) = h_K(M_1) + \Delta$. (Note that once again we are assuming a random key $K \neq 0$.) Informally this is true of the function above: for any pair of messages $M_1 \neq M_2$ represented by the coefficients $m_1^1, \dots, m_n^1$ and $m_1^2, \dots, m_n^2$ respectively, and any Δ chosen by the attacker, we can subtract one polynomial from the other and simplify as follows:

$$\begin{aligned} & (m_1^2 - m_1^1) \cdot K^1 + \dots + (m_n^2 - m_n^1) \cdot K^n = \Delta \\ \iff & (m_1^2 - m_1^1) \cdot K^1 + \dots + (m_n^2 - m_n^1) \cdot K^n - \Delta = 0 \end{aligned}$$

Note that since at least one coefficient $(m_i^2 - m_i^1)$ for $i \in [1, N]$ is non-zero, we can be certain that this final polynomial is not the zero polynomial.²⁹ Once again, the number of points K where this equation is satisfied is bounded by the degree of the polynomial, and the number of possible values for K is $p - 1$. Hence the probability that any triple (M_1, M_2, Δ) satisfies the equation should be bounded by $\frac{n}{p-1}$.

Computing the polynomial efficiently. A convenient detail about these hash functions is that they can be computed very efficiently, with roughly one addition and one multiplication per message field element. This is done using Horner's method (see §4.2), where the value K^i is incrementally calculated with one multiplication for every iteration through the loop.

6.2 Commitments

Some protocols require a means to *commit* to a specific value, in a manner that preserves the confidentiality of the value until some later time, at which point the author can “open” the commitment and reveal the committed value. For a typical real-world example, imagine bidding in a sealed-bid auction: here it's very important that your bid stays secret from the other bidders. At the same time, it's very important that when you do open your bid, you can't change the value you committed to. Out in the physical world we can solve this problem with an envelope: simply write your bid down, seal it in the envelope, and then open the envelope when it's time to determine the winner. To accomplish the same thing in the digital world, we can use a digital *commitment scheme*. A commitment scheme consists of two algorithms:

1. **Setup**(1^λ) $\rightarrow$ **pp**. This (optional) algorithm may need to be run by a trusted party. On input a security parameter λ it generates global parameters **pp** for the scheme.

²⁹Note that we've made a deliberate decision not to put any block of the message into the constant term of the polynomial hash, and always to include K^{n+1} in our calculation. An alternative implementation might place one block of the message into the constant term, and omit the K^{n+1} term. This might still provide collision security, but might not provide Δ -universality.

2. $\text{Commit}(\text{pp}, M; r) \rightarrow \text{Com}$. On input a message M and some explicit commitment randomness r (which we call an *opening*), this algorithm produces a commitment Com .

Using a commitment scheme is very simple. A party who wishes to commit to some message M first selects a random input r and generates a commitment Com by running the Commit algorithm above. It can then send the resulting commitment Com to its counterparties, or even post it on a website, all while retaining a strong guarantee that the message will remain secret. When the committing party is ready to “open” the commitment, it publishes the opening values (M, r) . Anyone in the world can verify that the opening is correct, simply by testing that:

$$\text{Com} \stackrel{?}{=} \text{Commit}(\text{pp}, M; r)$$

Security. Commitment schemes have two crucial security properties, which are known as the *hiding* and *binding* properties. The hiding property ensures that an attacker who sees only Com should learn nothing about the committed message M . This is analogous to the strong security properties used in encryption schemes. The binding property ensures that a given commitment Com can only be “opened” to one possible message M . More formally, this states that an adversarial party should not be able to produce two “opening” pairs (M_1, r_1) , (M_2, r_2) where $M_1 \neq M_2$, such that $\text{Commit}(\text{pp}, M_1; r_1) = \text{Commit}(\text{pp}, M_2; r_2)$.

Hash-based commitments. A common approach to building commitments uses cryptographic hash functions, combined with some randomness (salt) to prevent dictionary attacks. These can be built directly from hash functions, or they can be constructed using constructions like HMAC. Let HMAC be the HMAC function (§??) computed using a strong cryptographic hash function such as SHA2 or SHA3. No other global setup is required. To commit to a message M :

1. Pick a random string $r \in \{0, 1\}^\lambda$ (here λ is typically $128 - 256$ bits.)
2. Output $\text{Com} \leftarrow \text{HMAC}(r, M)$.

Many real-world constructions will skip the use of HMAC, setting $\text{Com} \leftarrow H(r \| M)$. Assuming the hash function is decent, this construction works fine as a commitment scheme. I don’t like it because of the possibility of length-extension attacks when using hash functions like SHA2 (see §??.) Although these attacks do not affect the security of the commitment scheme itself, they could allow another party who sees only Com to construct a new commitment Com' that commits to the value $M \| M'$ for some chosen M' . Whether this is an attack you care about really depends on your application, but the HMAC version shouldn’t be vulnerable to it at all.³⁰

³⁰At a deeper level, the security of the HMAC approach depends on the fact that the underlying hash function H is both a PRF and that it is collision-resistant. These assumptions are considered generally “ok” for modern hash functions.

Pedersen commitments. Another popular commitment scheme was proposed by Torben Pedersen in the 1990s [262]. Pedersen commitments have various advantages, including the fact that they enjoy perfect (Shannon) secrecy. The downside is that Pedersen commitments use stronger ingredients that are more commonly associated with public-key encryption, namely a cyclic group where the discrete logarithm problem is hard.

Setup. Let $\mathbb{G}$ be a multiplicative cyclic group of prime order q , with generator g .³¹ The **Setup** routine selects a second random generator $h \in \mathbb{G}$ such that $\log_g(h)$ is unknown. This can be done by a trusted party, or it can be performed by using a hash function to select h based on some input, such as a “nothing-up-my-sleeve” number. The resulting parameters **pp** are $(\mathbb{G}, g, h)$.

Commit. To commit to a message, it must first be encoded as an integer $m \in \mathbb{Z}_q$. This can be done, for example, by using an appropriate hash function to map longer messages into this range, as discussed in §6.1.3. To commit to m , sample a random integer $r \in \mathbb{Z}_q$ and then compute:

$$\text{Com} \leftarrow g^m h^r$$

Security. A nice feature of Pedersen commitments is that they provide perfect (Shannon) secrecy for committed messages. Since the random scalar r is as long as the message m , then the product $g^m h^r$ can be viewed as a one-time pad “encryption” of m . The binding property of Pedersen is not Shannon-secure, but is exactly as hard as solving the discrete logarithm for h base g .³² The implication of this is that you must also really trust that no party knows this discrete logarithm, or else they will be able to open commitments to any message of their choice!

6.3 Message Authentication Codes

One of the most useful authentication primitives in the cryptographer’s toolbox is the Message Authentication Code, or MAC. MACs are a symmetric and *keyed* primitive that facilitate the authentication of arbitrary messages, even when sent over a fully untrusted channel. MACs are often presented as the symmetric analog of digital signatures, although they provide slightly different security properties. While MACs can in principle be built from any components, most common MACs tend to be built from standard symmetric components such as block ciphers and hash functions.

³¹Here I’m using multiplicative notation, but Pedersen commitments can also be implemented in additive groups such as elliptic curves.

³²Let $x = \log_g(h)$ be the unknown discrete logarithm of h . If an attacker is able to find two different pairs $(m_1, r_1) \neq (m_2, r_2)$ such that $g^{m_1} h^{r_1} = g^{m_2} h^{r_2}$ then this implies that $m_1 + x r_1 \equiv m_2 + x r_2 \pmod{q}$. By re-arranging the terms, this equation can be solved for the unknown value x , which implies that the ability to find collisions in Pedersen commitments can be directly used to solve the discrete logarithm of h base g .

6.3.1 Algorithms and security notions.

Before we get to constructions, it’s helpful to explain what a MAC is. The standard notion of a MAC assumes that we have a key K_{MAC} that is typically a random string of an appropriate length, such as 128 bits. The MAC itself consists of a single algorithm:

1. $\text{MAC}(K_{\text{MAC}}, M) \rightarrow T$. On input a key and an arbitrary-length message $M \in \{0, 1\}^*$, this (deterministic) algorithm produces a MAC tag T .

Note that there is usually no explicit “verify” algorithm for a MAC scheme. To verify a tag T , a verifier with the key and message simply re-computes the MAC algorithm and compares the result to the expected tag, as follows:³³

$$\text{MAC}(K_{\text{MAC}}, M) \stackrel{?}{=} T$$

Security. The security for a MAC scheme is described by a definition that we refer to as (strong) existential unforgeability under chosen message attack, or **SUF-CMA**. This is a mouthful, so we’ll break this down into two parts:

- *Existential unforgeability*: this means that no attacker should be able to efficiently “forge” (or compute, without access to the secret key) a valid message/tag pair (M, T) where T is a valid tag for M under a secret and randomly-generated key K_{MAC} . The word “existential” here means that it does not matter what M contains: the scheme is considered insecure if an attacker can forge a valid MAC tag on literally *any* string, even a message that consists of arbitrary junk.
- Under *chosen message attack*: we require that unforgeability must hold even if an attacker can obtain many valid MAC tags on messages of its choosing. This models real-world systems, where an attacker will almost certainly see MAC tags flowing over the wire, and may even be able to trick a system into MACing some messages of its own. Note that forgery in this setting means coming up with a new pair (M, T) that is different from the ones it is given. This implies either a new message the attacker did not ask for a MAC on, or even a different tag T' that is not the same as the tag it was given for that message. (This last attack is what makes the attack “strong”.)

This strong unforgeability property is what makes MACs into an excellent primitive for protecting messages against tampering. Even if an attacker sees many different messages and tags flow over the wire, it should not be able to efficiently tamper with any of the messages (or

³³This describes every MAC that we use in practice. In theory it’s possible to describe a MAC algorithm that is randomized, and thus requires an explicit “verify” algorithm to check the MAC. However, this occurs mainly in the research literature. Most deployed MACs are verified by simply re-computing and comparing the tag.

even change their tags). An important caveat here is the word “efficiently”. There are several obvious attacks that will allow forgery against any MAC scheme, if the attacker has enough resources. The dumbest of these is to simply brute-force the MAC key K_{MAC} . Naturally this is why MAC keys need to be long enough that brute-force is not a viable strategy. MACs also admit an even sillier attack: the attacker can attempt to simply “guess” a valid MAC tag, for example, by producing a random string of the appropriate length for T . This attack will succeed against any MAC, but typically with very low probability — ideally no better than $2^{-|T|}$, where $|T|$ is the length of the tag in bits. Preventing this attack requires that tags be long enough to ensure that guessing is not feasible.³⁴

Notice that MACs do not protect a system against simple *replay attacks*. If an attacker sees a valid pair (M, T) on the wire, it can always replay the pair to the receiver, and the MAC will verify — assuming the key has not changed. If your application wishes to prevent message replays, this must be done by checking the contents of the message, for example, by keeping track of messages or including a message counter.

6.3.2 Building MACs

In the theoretical literature, MACs can be built from a variety of mathematical objects. If we restrict ourselves to widely deployed systems, there are three common techniques for building MACs. These make use of three different base ingredients: block ciphers, hash functions, and polynomial hashing.

6.3.3 MACs from hash functions

One common approach to building MACs is to use hash functions as a building block. These functions rely on various properties of the hash function, typically the *pseudorandomness* of its output, to ensure that the MAC is unforgeable. Before we get to the correct constructions, it’s worth noting the ideas that *aren’t* safe.

Pre-pending the key (potentially insecure).

In the earliest days of industrial cryptography, many protocol designers had the clever idea of producing a MAC by pre-pending a secret key K to a message M and feeding it into a hash function H as follows:

$$T \leftarrow H(K\|M)$$

The problem with this approach is that it appears so intuitively reasonable. If we make some very strong assumptions about the behavior of our hash function, then the output of

³⁴Sometimes for bandwidth reasons, protocols will use unusually short MACs: these can even be as short as 16 or 32 bits long. This is a risky proposition, since it does make the probability of guessing a valid MAC tag relatively high. A 32-bit tag can be guessed with probability 1 in approximately 4 billion, which is amazingly risky for a cryptographic scheme. Don’t use short MACs!

this function should look effectively “random” and unrelated to the key K . And worse, this logic is so reasonable that modern functions such as SHA-3 and Blake are evaluated under criteria that require that they be safe when used in this manner (see §6.1.)

Length-extensions attacks. The problem with the pre-pended design is that many common hash functions do *not* behave quite as we’d expect. The hash functions in question belong to the iterated Merkle-Damgård family, which we described in §???. The main problem here those functions’ vulnerability to *length-extension attacks*, which we described in §???. In a length-extension attack we are given the hash of some *unknown* message M , as $D \leftarrow H(M)$, and our goal is to construct some new message $M' = M \parallel E$ and a new digest D' such that $D' = H(M \parallel E)$.

In practice, this means that an attacker who receives a MAC tag $T \leftarrow H(K \parallel M)$ can forge a new tag T' that is a valid tag for some new message $M \parallel E$. The attacker does this by choosing E , and computing a length extension to find $T' \leftarrow H(K \parallel M \parallel E)$. Most crucially, the attacker never needs to learn the key K to execute this attack. This attack was discovered in the 1990s [310, 268], and so most damage was avoided. But it did lead to a few real breaks against some later protocols [114].

HMAC. The vulnerability of Merkle-Damgård constructions led to the development of alternative approaches. The most famous and widely-used of these is HMAC [41], which is standardized by FIPS 198 [245]. HMAC was designed to safely convert iterated Merkle-Damgård hash functions into secure MACs.³⁵ The HMAC construction is a sort of wrapper, which uses an underlying hash function as an ingredient. This means we typically refer to HMAC constructions according to the hash function being used: for example, **HMAC-SHA256** refers to the HMAC construction deployed with **SHA-256** as the underlying hash function.

HMAC requires a hash function H , which has an internal block size of b bits. The algorithm first computes a padded key K' that is computed from K_{MAC} as follows:

1. If $|K_{\text{MAC}}| \leq b$ then K' is simply K_{MAC} with (zero or more) additional zero bits added to the end of the key, until K' is b bits in length.
2. If $|K_{\text{MAC}}| > b$ then compute $K' \leftarrow H(K_{\text{MAC}})$ and append zero bits (if needed) until K' is b bits in length.

HMAC also makes use of two fixed constants **opad** $\neq$ **ipad**, each of which is b bits in length. To compute the tag, HMAC computes:

$$\text{HMAC}(K_{\text{MAC}}, M) = H((K' \oplus \text{opad}) \parallel H(K' \oplus \text{ipad} \parallel M))$$

Notice that HMAC appears to require more computation (hash calls) than the pre-pended construction above. However, this is a bit of an illusion. Assuming the message M is

³⁵HMAC can also be used safely with more recent functions such as SHA-3 and BLAKE, even if it is no longer strictly as necessary. Many standards still require this.

relatively long, the overwhelming bulk of the work in this function occurs within the internal call to $H(K' \oplus \text{ipad} \parallel M)$. The outer call to H only uses the digest, which means that it's typically only two internal blocks in length. Thus, at a cost of about two additional blocks of hashing, the HMAC design provably eliminates the risk of length extension attacks.

Security of HMAC. At first glance, the design of HMAC seems a bit arbitrary. Why *two* calls to the hash, and what's going on with `ipad` and `opad`? The answer is that HMAC was one of the earliest symmetric schemes to make use of *practice-oriented* provable security techniques. HMAC's designers modeled the internal round function of the Merkle-Damgård function as a random function, and then carefully analyzed every possible sequence of inputs and outputs to ensure that no attacker could devise an attack such as a length-extension attack, even if the underlying hash function does not prevent them. The distinct constants `ipad`, `opad` are one part of this: they ensure that the exact same key value is not fed into the round function twice.

Despite these efforts, HMAC has one small glitch that might be worth knowing about. Because the key K_{MAC} is only sometimes hashed to form K' , it is possible to find two different keys $K_{\text{MAC}1}, K_{\text{MAC}2}$ which appear different, but internally produce exactly the same value K' . This isn't a violation of **SUF-CMA** security, which always assumes a random key, and I'm not aware of any situation in which it should produce a vulnerability. But it's worth knowing that nothing is perfect.

6.3.4 Block cipher MACs

A second contender for the most popular MAC construction is the set of algorithms based on CBC-MAC using block ciphers as the base primitive. These are particularly popular in symmetric authenticated encryption schemes, and in settings where block ciphers are particularly efficient (for example, when hardware acceleration for AES is available.) The most popular secure MAC constructions based on this principle are `GMAC` and `CMAC`, which we'll describe below. Additionally, schemes like CCM-mode authenticated encryption (§??) make use of CBC-MAC as an ingredient.

Naïve CBC-MAC (not secure).

As a warm-up for the secure constructions below, it's worth explaining the basic CBC-MAC construction and then discussing why it is not, by itself, a secure MAC for arbitrary length messages. The basic idea of CBC-MAC is to employ the Cipher Block Chaining (CBC) mode of symmetric encryption, with the goal of using the final output of this construction to produce a MAC. Of course, since MACs are deterministic, this requires that the construction must not use a random initialization vector. Instead, CBC-MAC assumes a fixed, well-known IV that will be used for every message.

Let E, D be a block cipher and b be the block size. Let $M = (M_1, \dots, M_B)$ be a message that can be evenly divided into B blocks of length b bits each. (If the message is not a multiple of b bits, then it can be padded into one, using a non-ambiguous padding scheme.) Let IV be a fixed constant of length b bits. Set $T_0 \leftarrow IV$. For $i = 1$ to B , compute:

$$T_i \leftarrow E(K, T_{i-1} \oplus M_i)$$

Unlike CBC-mode encryption, we are not interested in all B output blocks. Instead, we will keep only the final block as our MAC tag, i.e., $T \leftarrow T_B$.

Insecurity against variable lengths. The main problem with the naïve CBC-MAC algorithm above is that it is not secure when messages are allowed to be of different, variable lengths. The threat here is a form of length-extension attack, which is quite similar to the attack leveraged against Merkle-Damgård iterated hash functions. For simplicity, we'll illustrate this by considering short (single-block) messages M_A, M' . Imagine that an attacker has the ability to query an oracle to obtain MACs under some key K_{MAC} . It first queries on M_A to obtain T_A . The attacker now constructs the message M_B as follows: let $M_B \leftarrow IV \oplus T_A \oplus M'$. It next queries for a MAC on M_B , to obtain T_B . Notice that so far it has queries on two messages, and hence it should not be able to produce a forgery on a third message — that is, a MAC on an entirely new message it has never queried on. And yet, it can now set $T_C \leftarrow T_B$ and output the pair (M_C, T_C) , which will validate. This works because if we decompose $M_C = (M_1, M_2)$, we get:

$$\begin{aligned} T_A &= E(K_{MAC}, IV \oplus M_A) \\ T_B &= E(K_{MAC}, IV \oplus M_B) = E(K_{MAC}, IV \oplus (IV \oplus T_A) \oplus M') = E(K_{MAC}, T_A \oplus M') \\ T_C &= MAC(K_{MAC}, M_A \| M') \end{aligned}$$

The nature of these MACs ensures that any tag is also the internal “state” of a MAC being computed on a longer message. Hence by carefully manipulating a tag and sending it to the oracle as part of a longer message, we can obtain a MAC tag on the longer message. Critically, we don't even need to know the actual message text of the message we are extending, which means that we can add on arbitrary text even to a hidden message — provided that we can obtain access to an oracle.

Dealing with variable-length messages. There are several ways to modify CBC-MAC to securely handle messages of variable length, and different deployed standards employ all of them. One simple approach is to pre-pend a single block that specifies the length of the message being MACed (either in bits, or blocks.) This eliminates length extension attacks, since it ensures that messages of each distinct length will have a very different internal state. The downside of this approach is that the length of the message must be known when the MAC function begins processing; this rules out *streaming* functions that do not know the length of the data at the time the MAC function begins taking input.