CHAPTER

5

Hash Functions

A *hash function* is a function that takes as input an arbitrarily long string of bits (or bytes) and produces a fixed-size result. A typical use of a hash function is digital signatures. Given a message m, you could sign the message itself. However, the public-key operations of most digital signature schemes are fairly expensive in computational terms. So instead of signing m itself, you apply a hash function h and sign $h(m)$ instead. The result of h is typically between 128 and 1024 bits, compared to multiple thousands or millions of bits for the message m itself. Signing $h(m)$ is therefore much faster than signing m directly. For this construction to be secure, it must be infeasible to construct two messages m_1 and m_2 that hash to the same value. We'll discuss the details of the security properties of hash functions below.

Hash functions are sometimes called *message digest* functions, and the hash result is also known as the *digest*, or the *fingerprint*. We prefer the more common name *hash function*, as hash functions have many other uses besides digesting messages. We must warn you about one possible confusion: the term ''hash function'' is also used for the mapping function used in accessing hash tables, a data structure used in many algorithms. These so-called hash functions have similar properties to cryptographic hash functions, but there is a huge difference between the two. The hash functions we use in cryptography have specific security properties. The hash-table mapping-function has far weaker requirements. Be careful not to confuse the two. When we talk about hash functions in this book, we always mean cryptographic hash functions.

Hash functions have many applications in cryptography. They make a great glue between different parts of a cryptographic system. Many times when

you have a variable-sized value, you can use a hash function to map it to a fixed-size value. Hash functions can be used in cryptographic pseudorandom number generators to generate several keys from a single shared secret. And they have a one-way property that isolates different parts of a system, ensuring that even if an attacker learns one value, she doesn't learn the others.

Even though hash functions are used in almost every system, we as a community currently know less about hash functions than we do about block ciphers. Until recently, much less research had been done on hash functions than block ciphers, and there were not very many practical proposals to choose from. This situation is changing. NIST is now in the process of selecting a new hash function standard, to be called SHA-3. The SHA-3 hash function selection process is proving to be very similar to the process that selected the AES as the new block cipher standard.

5.1 Security of Hash Functions

As we mentioned above, a hash function maps an input m to a fixed-size output $h(m)$. Typical output sizes are 128–1024 bits. There might be a limit on the input length, but for all practical purposes the input can be arbitrarily long. There are several requirements for a hash function. The simplest one is that it must be a one-way function: given a message m it is easy to compute $h(m)$, but given a value x it is not possible to find an m such that $h(m) = x$. In other words, a one-way function is a function that can be computed but that cannot be inverted—hence its name.

Of the many properties that a good hash function should have, the one that is mentioned most often is *collision resistance*. A collision is two different inputs m_1 and m_2 for which $h(m_1) = h(m_2)$. Of course, every hash function has an infinite number of these collisions. (There are an infinite number of possible input values, and only a finite number of possible output values.) Thus, a hash function is never collision-free. The collision-resistance requirement merely states that, although collisions exist, they cannot be found.

Collision resistance is the property that makes hash functions suitable for use in signature schemes. However, there are collision-resistant hash functions that are utterly unsuitable for many other applications, such as key derivation, one-way functions, etc. In practice, cryptographic designers expect a hash function to be a random mapping. Therefore, we require that a hash function be indistinguishable from a random mapping. Any other definition leads to a situation in which the designer can no longer treat the hash function as an idealized black box, but instead has to consider how the hash function properties interact with the system around it. (We number our definitions globally throughout this book.)

Definition 4 *The ideal hash function behaves like a random mapping from all possible input values to the set of all possible output values.*

Like our definition of the ideal block cipher (Section 3.3), this is an incomplete definition. Strictly speaking, there is no such thing as a random mapping; you can only talk about a probability distribution over all possible mappings. However, for our purposes this definition is good enough.

We can now define what an attack on a hash function is.

Definition 5 *An attack on a hash function is a non-generic method of distinguishing the hash function from an ideal hash function.*

Here the ideal hash function must obviously have the same output size as the hash function we are attacking. As with the block ciphers, the ''non-generic'' requirement takes care of all the generic attacks. Our remarks about generic attacks on block ciphers carry over to this situation. For example, if an attack could be used to distinguish between two ideal hash functions, then it doesn't exploit any property of the hash function itself and it is a generic attack.

The one remaining question is how much work the distinguisher is allowed to perform. Unlike the block cipher, the hash function has no key, and there is no generic attack like the exhaustive key search. The one interesting parameter is the size of the output. One generic attack on a hash function is the birthday attack, which generates collisions. For a hash function with an n-bit output, this requires about $2^{n/2}$ steps. But collisions are only relevant for certain uses of hash functions. In other situations, the goal is to find a pre-image (given x, find an m with $h(m) = x$), or to find some kind of structure in the hash outputs. The generic pre-image attack requires about 2^n steps. We're not going to discuss at length here which attacks are relevant and how much work would be reasonable for the distinguisher to use for a particular style of attack. To be sensible, a distinguisher has to be more efficient than a generic attack that yields similar results. We know this is not an exact definition, but—as with block ciphers—we don't have an exact definition. If somebody claims an attack, simply ask yourself if you could get a similar or better result from a generic attack that does not rely on the specifics of the hash function. If the answer is yes, the distinguisher is useless. If the answer is no, the distinguisher is real.

As with block ciphers, we allow a reduced security level if it is specified. We can imagine a 512-bit hash function that specifies a security level of 128 bits. In that case, distinguishers are limited to 2^{128} steps.

5.2 Real Hash Functions

There are very few good hash functions out there. At this moment, you are pretty much stuck with the existing SHA family: SHA-1, SHA-224, SHA-256, SHA-384, and SHA-512. There are other published proposals, including

submissions for the new SHA-3 standard, but these all need to receive more attention before we can fully trust them. Even the existing functions in the SHA family have not been analyzed nearly enough, but at least they have been standardized by NIST, and they were developed by the NSA.[1]

Almost all real-life hash functions, and all the ones we will discuss, are iterative hash functions. Iterative hash functions split the input into a sequence of fixed-size blocks $m_1, \ldots, m_k$, using a padding rule to fill out the last block. A typical block length is 512 bits, and the last block will typically contain a string representing the length of the input. The message blocks are then processed in order, using a compression function and a fixed-size intermediate state. This process starts with a fixed value H_0, and defines $H_i = h'(H_{i-1}, m_i)$. The final value H_k is the result of the hash function.

Such an iterative design has significant practical advantages. First of all, it is easy to specify and implement, compared to a function that handles variable-length inputs directly. Furthermore, this structure allows you to start computing the hash of a message as soon as you have the first part of it. So in applications where a stream of data is to be hashed, the message can be hashed on the fly without ever storing the data.

As with block ciphers, we will not spend our time explaining the various hash functions in great detail. The full specifications contain many details that are not relevant to the main goals of this book.

5.2.1 A Simple But Insecure Hash Function

Before discussing real hash functions, however, we will begin by giving an example of a trivially insecure iterative hash function. This example will help clarify the definition of a generic attack. This hash function is built from AES with a 256-bit key. Let K be a 256-bit key set to all zeros. To hash the message m, first pad it in some way and break it into 128-bit blocks $m_1, \ldots, m_k$; the details of the padding scheme aren't important here. Set H_0 to a 128-bit block of all zeros. And now compute $H_i = \mathrm{AES}_K(H_{i-1} \oplus m_i)$. Let H_k be the result of the hash function.

Is this a secure hash function? Is it collision resistant? Before reading further, try to see if you can find a way of breaking this hash function yourself.

Now here's a non-generic attack. Pick a message m such that after padding it splits into two blocks m_1 and m_2. Let H_1 and H_2 denote the values computed as part of the hash function's internal processing; H_2 is also the output of the hash function. Now let $m'_1 = m_2 \oplus H_1$ and let $m'_2 = H_2 \oplus m_2 \oplus H_1$, and let m' be the message that splits into m'_1 and m'_2 after padding. Due to properties of the hash function's construction, m' also hashes to H_2; you can verify this in the exercises at the end of this chapter. And with very high probability, m and m' are different strings. That's right—m and m' are two distinct messages

[1]Whatever you may think about the NSA, so far the cryptography it has published has been quite decent.

that produce a collision when hashed with this hash function. To convert this into a distinguishing attack, simply try to mount this attack against the hash function. If the attack works, the hash function is the weak one we described here; otherwise, the hash function is the ideal one. This attack exploits a specific weakness in how this hash function was designed, and hence this attack is non-generic.

5.2.2 MD5

Let's now turn to some real hash function proposals, beginning with MD5. MD5 is a 128-bit hash function developed by Ron Rivest [104]. It is a further development of a hash function called MD4 [106] with additional strengthening against attacks. Its predecessor MD4 is very fast, but also broken [36]. MD5 has now been broken too. You will still hear people talk about MD5, however, and it is still in use in some real systems.

The first step in computing MD5 is to split the message into blocks of 512 bits. The last block is padded and the length of the message is included as well. MD5 has a 128-bit state that is split into four words of 32 bits each. The compression function h' has four rounds, and in each round the message block and the state are mixed. The mixing consists of a combination of addition, XOR, AND, OR, and rotation operations on 32-bit words. (For details, see [104].) Each round mixes the entire message block into the state, so each message word is in fact used four times. After the four rounds of the h' function, the input state and result are added together to produce the output of h'.

This structure of operating on 32-bit words is very efficient on 32-bit CPUs. It was pioneered by MD4, and is now a general feature of many cryptographic primitives.

For most applications, the 128-bit hash size of MD5 is insufficient. Using the birthday paradox, we can trivially find collisions on any 128-bit hash function using 2^{64} evaluations of the hash function. This would allow us to find real collisions against MD5 using only 2^{64} MD5 computations. This is insufficient for modern systems.

But the situation with MD5 is worse than that. MD5's internal structure makes it vulnerable to more efficient attacks. One of the basic ideas behind the iterative hash function design is that if h' is collision-resistant, then the hash function h built from h' is also collision-resistant. After all, any collision in h can only occur due to a collision in h'. For over a decade now it has been known that the MD5 compression function h' has collisions [30]. The collisions for h' don't immediately imply a collision for MD5. But recent cryptanalytic advances, beginning with Wang and Yu [124], have now shown that it is actually possible to find collisions for the full MD5 using much fewer than 2^{64} MD5 computations. While the existence of such efficient collision finding attacks may not immediately break all uses of MD5, it is safe to say that MD5 is very weak and should no longer be used.

5.2.3 SHA-1

The Secure Hash Algorithm was designed by the NSA and standardized by NIST [97]. The first version was just called SHA (now often called SHA-0). The NSA found a weakness with SHA-0, and developed a fix that NIST published as an improved version, called SHA-1. However, they did not release any details about the weakness. Three years later, Chabaud and Joux published a weakness of SHA-0 [25]. This is a weakness that is fixed by the improved SHA-1, so it is reasonable to assume that we now know what the problem was.

SHA-1 is a 160-bit hash function based on MD4. Because of its shared parentage, it has a number of features in common with MD5, but it is a far more conservative design. It is also slower than MD5. Unfortunately, despite its more conservative design, we now know that SHA-1 is also insecure.

SHA-1 has a 160-bit state consisting of five 32-bit words. Like MD5, it has four rounds that consist of a mixture of elementary 32-bit operations. Instead of processing each message block four times, SHA-1 uses a linear recurrence to ''stretch'' the 16 words of a message block to the 80 words it needs. This is a generalization of the MD4 technique. In MD5, each bit of the message is used four times in the mixing function. In SHA-1, the linear recurrence ensures that each message bit affects the mixing function at least a dozen times. Interestingly enough, the only change from SHA-0 to SHA-1 was the addition of a one-bit rotation to this linear recurrence.

Independent of any internal weaknesses, the main problem with SHA-1 is the 160-bit result size. Collisions against any 160-bit hash function can be generated in only 2^{80} steps, well below the security level of modern block ciphers with key sizes from 128 to 256 bits. It is also insufficient for our design security level of 128 bits. Although it took longer for SHA-1 to fall than MD5, we now know that it is possible to find collisions in SHA-1 using much less work than 2^{80} SHA-1 computations [123]. Remember that attacks always get better? It is no longer safe to trust SHA-1.

5.2.4 SHA-224, SHA-256, SHA-384, and SHA-512

In 2001, NIST published a draft standard containing three new hash functions, and in 2004 they updated this specification to include a fourth hash function [101]. These hash functions are collectively referred to as the SHA-2 family of hash functions. These have 224-, 256-, 384-, and 512-bit outputs, respectively. They are designed to be used with the 128-, 192-, and 256-bit key sizes of AES, as well as the 112-bit key size of 3DES. Their structure is very similar to SHA-1.

These hash functions are new, which is generally a red flag. However, the known weaknesses of SHA-1 are much more severe. Further, if you want more security than SHA-1 can give you, you need a hash function with a larger result. None of the published designs for larger hash functions have received

much public analysis; at least the SHA-2 family has been vetted by the NSA, which generally seems to know what it is doing.

SHA-256 is much slower than SHA-1. For long messages, computing a hash with SHA-256 takes about as much time as encrypting the message with AES or Twofish, or maybe a little bit more. This is not necessarily bad, and is an artifact of its conservative design.

5.3 Weaknesses of Hash Functions

Unfortunately, all of these hash functions have some properties that disqualify them according to our security definition.

5.3.1 Length Extensions

Our greatest concern about all these hash functions is that they have a length-extension bug that leads to real problems and that could easily have been avoided. Here is the problem. A message m is split into blocks $m_1, \ldots, m_k$ and hashed to a value H. Let's now choose a message m' that splits into the block $m_1, \ldots, m_k, m_{k+1}$. Because the first k blocks of m' are identical to the k blocks of message m, the hash value $h(m)$ is merely the intermediate hash value after k blocks in the computation of $h(m')$. We get $h(m') = h'(h(m), m_{k+1})$. When using MD5 or any hash function from the SHA family, you have to choose m' carefully to include the padding and length field, but this is not a problem as the method of constructing these fields is known.

The length extension problem exists because there is no special processing at the end of the hash function computation. The result is that $h(m)$ provides direct information about the intermediate state after the first k blocks of m'.

This is certainly a surprising property for a function we want to think of as a random mapping. In fact, this property immediately disqualifies all of the mentioned hash functions, according to our security definition. All a distinguisher has to do is to construct a few suitable pairs (m, m') and check for this relationship. You certainly wouldn't find this relationship in an ideal hash function. This is a non-generic attack that exploits properties of the hash functions themselves, so this is a valid attack. The attack itself takes only a few hash computations, so it is very quick.

How could this property be harmful? Imagine a system where Alice sends a message to Bob and wants to authenticate it by sending $h(X \parallel m)$, where X is a secret known only to Bob and Alice, and m is the message. If h were an ideal hash function, this would make a decent authentication system. But with length extensions, Eve can now append text to the message m, and update the authentication code to match the new message. An authentication system that allows Eve to modify the message is, of course, of no use to us.

This issue will be resolved in SHA-3; one of the NIST requirements is that SHA-3 not have length-extension properties.

5.3.2 Partial-Message Collision

A second problem is inherent in the iterative structure of most hash functions. We'll explain the problem with a specific distinguisher.

The first step of any distinguisher is to specify the setting in which it will differentiate between the hash function and the ideal hash function. Sometimes this setting can be very simple: given the hash function, find a collision. Here we use a slightly more complicated setting. Suppose we have a system that authenticates a message m with $h(m \parallel X)$, where X is the authentication key. The attacker can choose the message m, but the system will only authenticate a single message.[2]

For a perfect hash function of size n, we expect that this construction has a security level of n bits. The attacker cannot do any better than to choose an m, get the system to authenticate it as $h(m \parallel X)$, and then search for X by exhaustive search. The attacker can do much better with an iterative hash function. She finds two strings m and m' that lead to a collision when hashed by h. This can be done using the birthday attack in only $2^{n/2}$ steps or so. She then gets the system to authenticate m, and replaces the message with m'. Remember that h is computed iteratively, so once there is a collision and the rest of the hash inputs are the same, the hash value stays the same, too. Because hashing m and m' leads to the same value, $h(m \parallel X) = h(m' \parallel X)$. Notice that this attack does not depend on X—the same m and m' would work for *all* values for X.

This is a typical example of a distinguisher. The distinguisher sets its own ''game'' (a setting in which it attempts an attack), and then attacks the system. The object is still to distinguish between the hash function and the ideal hash function, but that is easy to do here. If the attack succeeds, it is an iterative hash function; if the attack fails, it is the ideal hash function.

5.4 Fixing the Weaknesses

We want a hash function that we can treat as a random mapping, but all well-known hash functions fail this property. Will we have to check for length-extension problems in every place we use a hash function? Do we check for partial-message collisions everywhere? Are there any other weaknesses we need to check for?

[2]Most systems will only allow a limited number of messages to be authenticated; this is just an extreme case. In real life, many systems include a message number with each message, which has the same effect on this attack as allowing only a single message to be chosen.

Leaving weaknesses in the hash function is a very bad idea. We can guarantee that it will be used somewhere in a way that exposes the weakness. Even if you document the known weaknesses, they will not be checked for in real systems. Even if you could control the design process that well, you would run into a complexity problem. Suppose the hash function has three weaknesses, the block cipher two, the signature scheme four, etc. Before you know it, you will have to check hundreds of interactions among these weaknesses: a practical impossibility. We have to fix the hash function.

The new SHA-3 standard will address these weaknesses. In the meantime, we need short-term fixes.

5.4.1 Toward a Short-term Fix

Here is one potential solution. Ultimately, we'll recommend the fixes in the subsequent subsections, and this particular proposal has not received significant review within the community. But this discussion is illustrative, so we include it here.

Let h be one of the hash functions mentioned above. Instead of $m \mapsto h(m)$, one could use $m \mapsto h(h(m) \parallel m)$ as a hash function.[3] Effectively we put $h(m)$ before the message we are hashing. This ensures that the iterative hash computations immediately depend on all the bits of the message, and no partial-message or length-extension attacks can work.

Definition 6 *Let h be an iterative hash function. The hash function h_{DBL} is defined by $h_{\text{DBL}}(m) := h(h(m) \parallel m)$.*

We believe that if h is any of the newer SHA-2 family hash functions, this construction has a security level of n bits, where n is the size of the hash result.

A disadvantage of this approach is that it is slow. You have to hash the entire message twice, which takes twice as long. Another disadvantage is that this approach requires the whole message m to be buffered. You can no longer compute the hash of a stream of data as it passes by. Some applications depend on this ability, and using h_{DBL} would simply not work.

5.4.2 A More Efficient Short-term Fix

So how do we keep the full speed of the original hash function? We cheat, kind of. Instead of $h(m)$, we can use $h(h(0^b \parallel m))$ as a hash function, and claim a security level of only $n/2$ bits. Here b is the block length of the underlying compression function, so $0^b \parallel m$ equates to prepending the message with an all zero block before hashing. The cheat is that we normally expect an n-bit

[3]The notation $x \mapsto f(x)$ is a way of writing down a function without having to give it a name. For example: $x \mapsto x^2$ is a function that squares its input.

hash function to provide a security level of n bits for those situations in which a collision attack is not possible.[4] The partial-message collision attacks all rely on birthday attacks, so if we reduce the security level to $n/2$ bits, these attacks no longer fall within the claimed security level.

In most situations, reducing the security level in this way would be unacceptable, but we are lucky here. Hash functions are already designed to be used in situations where collision attacks are possible, so the hash function sizes are suitably large. If we apply this construction to SHA-256, we get a hash function with a 128-bit security level, which is exactly what we need.

Some might argue that all n-bit hash functions provide only $n/2$ bits of security. That is a valid point of view. Unfortunately, unless you are very specific about these things, people will abuse the hash function and assume it provides n bits of security. For example, people want to use SHA-256 to generate a 256-bit key for AES, assuming that it will provide a security level of 256 bits. As we explained earlier, we use 256-bit keys to achieve a 128-bit security level, so this matches perfectly with the reduced security level of our fixed version of SHA-256. This is not accidental. In both cases the gap between the size of the cryptographic value and the claimed security level is due to collision attacks. As we assume collision attacks are always possible, the different sizes and security levels will fit together nicely.

Here is a more formal definition of this fix.

Definition 7 *Let h be an iterative hash function, and let b denote the block length of the underlying compression function. The hash function h_d is defined by $h_d(m) := h(h(0^b \parallel m))$, and has a claimed security level of* $\min(k, n/2)$ *where k is the security level of h and n is the size of the hash result.*

We will use this construction mostly in combination with hash functions from the SHA family. For any hash function SHA-X, where X is 1, 224, 256, 384, or 512, we define SHA$_d$-X as the function that maps m to SHA-X(SHA-X($0^b \parallel m$)). SHA$_d$-256 is just the function $m \mapsto$ SHA-256(SHA-256($0^{512} \parallel m$)), for example.

This particular fix to the SHA family of iterative hash functions, in addition to being related to our construction in Section 5.4.1, was also described by Coron et al. [26]. It can be demonstrated that the fixed hash function h_d is at least as strong as the underlying hash function h.[5] HMAC uses a similar hash-it-again approach to protect against length-extension attacks. Prepending the message with a block of zeros makes it so that, unless something unusual happens, the

[4]Even the SHA-256 documentation claims that an n-bit hash function should require 2^n steps to find a pre-image of a given value.

[5]We're cheating a little bit here. By hashing twice, the range of the function is reduced, and birthday attacks are a little bit easier. This is a small effect, and it falls well within the margin of approximation we've used elsewhere.

first block input to the inner hash function in h_d is different than the input to the outer hash function. Both h_{DBL} and h_d eliminate the length extension bug that poses the most danger to real systems. Whether h_{DBL} in fact has a security level of n bits remains to be seen. We would trust both of them up to $n/2$ bits of security, so in practice we would use the more efficient h_d construction.

5.4.3 Another Fix

There is another fix to some of these weaknesses with the SHA-2 family of iterative hash functions: Truncate the output [26]. If a hash function produces n-bit outputs, only use the first $n - s$ of those bits as the hash value for some positive s. In fact, SHA-224 and SHA-384 both already do this; SHA-224 is roughly SHA-256 with 32 output bits dropped, and SHA-384 is roughly SHA-512 with 128 output bits dropped. For 128 bits of security, you could hash with SHA-512, drop 256 bits of the output, and return the remaining 256 bits as the result of the truncated hash function. The result would be a 256-bit hash function which, because of birthday attacks, would meet our 128-bit security design goal.

5.5 Which Hash Function Should I Choose?

Many of the submissions to NIST's SHA-3 competition have revolutionary new designs, and they address the weaknesses we've discussed here and other concerns. However, the competition is still going on and NIST has not selected a final SHA-3 algorithm. Much additional analysis is necessary in order to have sufficient confidence in the SHA-3 submissions. In the short term, we recommend using one of the newer SHA hash function family members—SHA-224, SHA-256, SHA-384, or SHA-512. Moreover, we suggest you choose a hash function from the SHA_d family, or use SHA-512 and truncate the output to 256 bits. In the long term, we will very likely recommend the winner of the SHA-3 competition.

5.6 Exercises

Exercise 5.1 Use a software tool to generate two messages M and M', $M \neq M'$, that produce a collision for MD5. To generate this collision, use one of the known attacks against MD5. A link to example code for finding MD5 collisions is available at: `http://www.schneier.com/ce.html`.

Exercise 5.2 Using an existing cryptography library, write a program to compute the SHA-512 hash value of the following message in hex:

```
48 65 6C 6C 6F 2C 20 77 6F 72 6C 64 2E 20 20 20.
```

Exercise 5.3 Consider SHA-512-n, a hash function that first runs SHA-512 and then outputs only the first n bits of the result. Write a program that uses a birthday attack to find and output a collision on SHA-512-n, where n is a multiple of 8 between 8 and 48. Your program may use an existing cryptography library. Time how long your program takes when n is 8, 16, 24, 32, 40, and 48, averaged over five runs for each n. How long would you expect your program to take for SHA-512-256? For SHA-512-384? For SHA-512 itself?

Exercise 5.4 Let SHA-512-n be as in the previous exercise. Write a program that finds a message M (a pre-image) that hashes to the following value under SHA-512-8 (in hex):

```
A9.
```

Write a program that finds a message M that hashes to the following value under SHA-512-16 (in hex):

```
3D 4B.
```

Write a program that finds a message M that hashes to the following value under SHA-512-24 (in hex):

```
3A 7F 27.
```

Write a program that finds a message M that hashes to the following value under SHA-512-32 (in hex):

```
C3 C0 35 7C.
```

Time how long your programs take when n is 8, 16, 24, and 32, averaged over five runs each. Your programs may use an existing cryptography library. How long would you expect a similar program to take for SHA-512-256? For SHA-512-384? For SHA-512 itself?

Exercise 5.5 In Section 5.2.1, we claimed that m and m' both hash to H_2. Show why this claim is true.

Exercise 5.6 Pick two of the SHA-3 candidate hash function submissions and compare their performance and their security under the currently best published attacks. Information about the SHA-3 candidates is available at `http://www.schneier.com/ce.html`.