Chapter 2
Preliminaries

There are $2^{18446744073709551616} - 1$ *possible [SHA-256] inputs.*
—Thomas Pornin

It seems very hard to make a mathematical definition that captures the idea that human beings can't find collisions in SHA1.
—Mihir Bellare

This chapter introduces the reader to cryptographic hash functions, starting with an informal review of the most common applications, from modification detection and digital signature to key update and timestamping. We then present slightly more formally the security notions associated with hash functions, discussing in particular what being "one-way" means (which is less simple than it sounds). Getting more technical, we review state-of-the-art generic collision search methods, and constructions of hash functions. Finally, we conclude with an overview of the SHA1 and SHA2 standards, as well as of the SHA3 finalists.

2.1 Applications

As previously commented, hash functions are a cryptographer's Swiss Army knife, for they can serve a large variety of purposes in security systems, acting as one-way functions, collision-resistant functions, or pseudorandom functions, among others.

We review the most common applications of hash functions, starting with the best known, namely modification detection, message authentication, and digital signatures.

2.1.1 Modification Detection

Modification detection is probably the oldest and most straightforward application of cryptographic hash functions, as it only aims to protect integrity of data rather than authenticity—no secret key is involved; one just computes $H(M)$ and compares it with the received value for verification. This is used in a model where a communication channel is secure but unreliable (accidental transmission errors may occur).

J.-P. Aumasson et al., *The Hash Function BLAKE*, Information Security and Cryptography, DOI 10.1007/978-3-662-44757-4_2

In this context, hash values are often called *checksums* when used for modification detection. Checksums are, for example, added at the end of a transmitted packet so that the recipient can check that the received hash value matches the value computed from the received data. Simple, insecure, checksum algorithms such as cyclic redundancy checks (CRCs) are widely used for detection of accidental errors, due to their simplicity and efficiency (as found in trailers of Ethernet frames). However, secure modification codes should protect not only against accidental modification but also against malicious ones. In particular, they should be second-preimage-resistant hash functions, to avoid forgery of data matching the published checksum.

An example of the use of hash functions as checksums is by websites proposing the download of software packages: the website publishes a URL to a software package along with the hash of the target file so that users can verify that they downloaded the legitimate data. This protects against accidental errors as well as straightforward malicious man-in-the-middle modifications of the downloaded content, but not against more clever attackers who would adapt the checksum to the modified file. Also, mere hashing with no secret key does not authenticate the origin of the file (unless, indirectly, if within an HTTPS tunnel).

Checksums are typically used in peer-to-peer file-sharing systems. For example, BitTorrent protects the integrity of data transferred by hashing individually each piece (between 32 kB and 4 MB) of a file with SHA1, and recording it in the torrent file. As previously mentioned, computer forensics uses hash functions to provide proofs of nonmodification of collected evidence. Plenty of other applications use hash functions for ensuring data integrity: intrusion detection systems (e.g., Artillery, Samhain), version control systems (e.g., Git, Perforce), integrity-checking filesystems (e.g., ZFS), cloud storage systems (e.g., OpenStack Swift), distributed filesystems (e.g., Tahoe-LAFS), etc.

2.1.2 Message Authentication

Message authentication codes (MACs) are essentially keyed versions of modification detection codes, and thus provide authenticity in addition to integrity protection—i.e., they ensure that the hash obtained was computed by a party sharing the secret key. The values $H_K(M)$ produced by MACs are sometimes called *authenticators* or just *MACs*. Not all MACs are constructed as keyed hash functions, however: UMAC [44, 110] and Poly1305-AES [25] are examples of MACs that instead rely on a universal hash function combined with a pseudorandom function or an encryption function.

The main security requirement of a MAC is resistance to *forgery*, i.e., the inability for an adversary to determine a legitimate authenticator for some message. As with the definition of preimage resistance, different notions of forgery exist depending on the distribution of the message. From the weakest to the strongest notions:

- **Existential forgery** puts no restriction on the message; it can, for example, be a meaningless, random-looking value;

- **Selective forgery** obliges the attacker to choose the message prior to the attack, for example, as being in a "weak" subset of messages;
- **Universal forgery** is the ability to create a valid signature for any given message.

In all definitions, the attacker is assumed to be able to request a valid MAC of any message of its choice. A forgery is thus successful if the values returned have not been obtained with such a query. It is known that a keyed hash function is a secure MAC if it is pseudorandom.

The most common hash-function-based construction of MACs is the NIST standard HMAC, which for a hash function H, a key K, and a message M returns

$$\text{HMAC-}H(K,M) = H\big((K \oplus \mathsf{opad}) \| H\left((K \oplus \mathsf{ipad}) \| M\right)\big) \ ,$$

where K is padded with zero bytes `00` to fill a data block, $\mathsf{opad} = \texttt{5c5c}\ldots\texttt{5c}$, and $\mathsf{ipad} = \texttt{3636}\ldots\texttt{36}$. We refer to [109, 132] for a complete specification detailing particular cases—like how to handle keys longer than a block.

In practice, MACs are often sent jointly with the ciphertext of some message, the authenticator being computed on the plaintext or on the ciphertext; for example, IPsec's encapsulated security payloads are protected with HMAC-SHA1 (i.e., a MAC is computed on the encrypted data). A disadvantage of HMAC, however, is its suboptimal efficiency on short data: at least two blocks of data have to be processed, and three if the outer hash cannot be precomputed.

2.1.3 Digital Signatures

Digital signature schemes are *asymmetric* (i.e., public-key) cryptographic schemes composed of three algorithms:

- A **key generation** algorithm Gen, which given a security level as parameter creates a key pair (sk, pk), where sk is the (secret) signing key and pk is the (public) verification key
- A **signing algorithm** Sign that takes a signing key and data M and that returns a signature $s := \mathsf{Sign}_{sk}(M)$
- A **verification** algorithm Verif that takes a verification key, a signature s, and data M and that returns $\mathsf{Verif}_{pk}(s,M)$ either valid or $\mathsf{invalid}$

Common signature schemes rely on hard mathematical problems (RSA, discrete logarithm) and involve big-number arithmetic operations, which makes them orders of magnitude slower than typical block ciphers or hash functions. Moreover, signature schemes generally accept inputs of limited and small length, such as fewer than 2,048 bits. For these reasons, the signature of a message is computed by applying the signing algorithm to the messages digest rather than to the message itself (the so-called hash-then-sign paradigm). Typical signature schemes are RSA as per the Public-Key Cryptographic Standard #1 (PKCS#1) [89] and the Digital Signature Standard (DSS) [133].

The main security requirement for a signature scheme is that *forgery* of valid signature-and-message pairs should be infeasible except for the signer, even when many such pairs are known. It is easy to see that, if the hash function used is not second-preimage resistant, one can forge a valid signature-and-message pair by "recycling" a known pair. In fact, collision resistance is necessary if the attacker can choose messages to be signed by the legitimate party. Collision resistance is also necessary to ensure nonrepudiation of signatures by the signer.

Many different types of signatures have been proposed, with various functionalities and security requirements, for example:

- **Undeniable signatures** are verified through interactions between the signer and the prover rather than with a single algorithm on the verifier side, and allow the signer to prove the invalidity of a signature. These two features allow the signer to choose who can verify a given signature.
- **Group signatures** allow a member of a group to anonymously sign data on behalf of the group.
- **Randomized signatures** are like normal signatures but with randomized hashing rather than deterministic hashing. This allows one to drop the requirement of collision resistance for a weaker form called target collision resistance.

One of the most common uses of digital signatures is in HTTPS-secured websites, which can prove their identity by sending a *certificate* signed by a certification authority (CA), verified on the client side using the public key of the CA embedded in one's browser. Signatures are also used to prevent the execution of arbitrary code on smartphones and game consoles via the implementation of a chain of trust, although these protection mechanisms are regularly broken due to flaws in design and/or implementation.

2.1.4 Pseudorandom Functions

Pseudorandom functions (PRFs) are objects that satisfy *computational indistinguishability* from uniform outputs. In practice hash-based PRFs (as defined in Definition 5) are seldom used isolatedly, but rather as part of a protocol whose security relies on the indistinguishability property of the PRF.

A common way to construct a hash-based PRF is to use the HMAC construction (see Section 2.1.2); for example, the Internet Key Exchange (IKEv2) protocol of the IPsec suite uses HMAC-SHA-256 as a PRF to set up a shared secret key [97, 98].

PRFs are found in many widely used protocols. For example, the transport layer security (TLS) handshake makes use of two dedicated PRF constructions (analyzed in detail in [69]). The Kerberos V authentication protocol also needs PRFs to ensure provable security. Finally, PRFs are at the core of certain password-based key derivation and hashing schemes, as discussed in the next section. Fundamentally, MACs and PRFs are identical objects.

2.1.5 Entropy Extraction and Key Derivation

The ability of hash functions to eliminate structures and symmetries of related inputs to produce random-looking outputs is leveraged for the following applications:

- **Entropy extraction**, that is, exploiting the possibly nonuniform[1] randomness of some entropy pool to produce uniformly distributed strings, thus maximizing the per-bit entropy. For a formal definition of entropy extractors and theoretical results, we refer the reader to the work of Dodis et al. [62]
- **Key derivation**, that is, the generation of cryptographic keys from secret and public parameters; for example, from a serial number, timestamp, and secret global key. A common application is password-based key derivation, for which the notion of key stretching [101] was introduced to mitigate bruteforce attacks and simulate extra entropy by enforcing additional computation per password.

The PKCS standard PBKDF2 is a common password-based key derivation function [94, 95]. It uses a pseudorandom function to produce a key from a salt and a password, using a variable number of iterations of the PRF—the more iterations, the slower the bruteforce. The standard recommends at least 1,000 iterations. The use of PBKDF2 with only one iteration in a previous version of the Blackberry software was shown to be a major security flaw[2] For comparison, Apple's mobile OS iOS3 used 2,000 iterations, and its subsequent versions 10,000 iterations.

2.1.6 Password Hashing

Password databases are regularly "dumped" from servers by malicious intruders. To mitigate the risk to users, a good practice is to store hashes of the passwords rather than the passwords themselves. However, a simple $H(\text{password})$ exposes users' passwords to efficient time-memory tradeoffs—commonly known as rainbow tables, the main technique employed. Slightly better is to hash with a salt, but this still leaves passwords exposed to bruteforce and dictionary attacks with tools such as John the Ripper of Hashcat, especially when ran on graphical processing units (GPUs).

A much better practice is to use a dedicated *password hashing function*, that is sufficiently slow (among other properties) to mitigate bruteforce and dictionary attacks. Such functions include the PBKDF2 construction (originally intended for key derivation) and the dedicated designs bcrypt [150] and scrypt [143].

These last years, several major organizations fell victim of "passwords thefts", actually the extraction of databases of weak hashes (sometimes as simple as salt-less MD5) following a compromise of one of their servers. Hundreds of thousands

[1] A source of bits from a physical phenomenon may appear random, yet with respect to a distribution that is not the ideal, uniform, one. There may, for example, be more ones than zeroes, or the pattern "1010" may occur more often than the pattern "0101," etc.

[2] Cf. vulnerability CVE-2010-3741.

of users suffered the publication of their password, which often is reused through email accounts, social network services, etc.

To address the lack of research and solutions for secure password hashing—and password protection more generally—the Password Hashing Competition[3] was initiated in 2013, on the same principle as the SHA3 competition: the community is invited to submit password hashing schemes, which will be evaluated by a panel of experts. The submission deadline was set to March 31, 2014, and the selection of one or more designs is expected in Q2 2015.

2.1.7 Data Identification

The (practical) uniqueness of a file's fingerprint through a hash function is exploited in forensics investigations to facilitate identification of files within a large file system. The short length of fingerprints allows compact storage of large databases of target files. Target files are, for example, illegal content, specific system files, or files known to be dangerous. Refined techniques, such as piecewise hashing, are used to deal with partially modified files.

Hash functions have for example been used by antimalware tools to assist in identification of malicious payloads.

2.1.8 Key Update

Two or more parties that share a secret key K can agree to update as $K := H(K)$ at predefined times, so that the compromise of a K does not compromise earlier K's, thanks to the preimage resistance of H. This property is called *forward security*, and is also known as *backtracking resistance* in the context of PRNGs.

If the update of the key depends on the data exchanged—for instance, if K is updated as $K := H(K,M)$, where M is an aggregate of all the data exchanged since the last update—then the property of *backward security* (also known as *prediction resistance*) can be achieved; that is, the compromise of a K does not compromise future K's if the attack does not observe all the traffic.

2.1.9 Proof-of-Work Systems

Proof-of-work systems aim to force a party to perform some resource-consuming operation prior to performing some task, such as sending email, accessing to a ser-

[3] https://password-hashing.net

vice, etc. These aim to deter massive execution of a task and thus prevent abuse (e.g., spam email) or denial of service.

The one-wayness and unpredictability of hash functions is exploited by proof-of-work systems such as the famous Hashcash,[4] originally designed as an antispam measure. Given a header containing metadata (such as email address, timestamp, etc.), Hashcash clients seek a nonce that, when combined with the header, hashes to a value with a given number of leading 0 bits. Initially, Hashcash used SHA-1 and searches for a hash value with 20 leading zeroes. The famous cryptocurrency Bitcoin relies on Hashcash with a "double" SHA-256 instead of SHA1, and an adapted number of leading zeroes, varying over time: initially set to 32 in 2009, it is 63 at the time of writing, representing about $2^{63}/10$ double hashes per minute. Litecoin uses scrypt [143] rather than SHA-256, a password hashing scheme designed to use significant memory and thus mitigate the efficiency of GPUs, FPGAs, and ASICs.

2.1.10 Timestamping

One can use hash functions to commit to data while keeping it hidden, with the ability to later reveal the data as evidence of earlier knowledge. The said data can, for example, be a scientific result, a document establishing intellectual property, financial forecasts, informants' names, etc. A commercial trusted timestamping service can be used to guarantee the exact time of publication, although one may choose to just publish it on (say) Twitter.

The properties exploited are (a strong form of) collision resistance and preimage resistance of the hash function. It was shown that MD5 cannot guarantee secure commitments, when researchers "predicted" the outcome of the 2008 US presidential election [166] by revealing the MD5 digest of the president's name a year before the vote.

2.2 Security Notions

2.2.1 Security Models

Informally, a hash function is *preimage resistant* if it is hard to find data mapping to a given hash value. Although this definition sounds self-explanatory, one notices that "hard" needs to be precisely defined. There are essentially three definitions of "hard," whose acceptance depends on the community considered:

- *Theoretical cryptographers* and complexity theorists often define a problem as hard if it can be proven that solving it requires a number of operations growing

[4] http://hashcash.org.

faster than any polynomial function of the "size" of the problem. Although this definition captures well the intuitive notion of hardness for scalable classes of problems, it is not relevant for hash functions with fixed parameters, as used in practice.

- *Applied cryptographers* seldom use the term "hard" in security definitions. They rather consider a hash function preimage resistant if there is no method substantially faster than bruteforce to compute a preimage of some hash value. The exact definition of "substantially" is disputed, as well as that of "some value"—this point is discussed further in this section.
- *Security practitioners* tend to have a more pragmatic view, and understand "hard" as "infeasible in practice." That is, they tend to be satisfied with a theoretically suboptimal security level, as long as actual applications are not threatened and that the security level remains of an adequate order of magnitude; for example, a collision attack with time complexity 2^{120} and memory 2^{64} instead of only time 2^{128} is not really a concern for security. More pragmatically, and from a risk analysis standpoint, cryptography remains strong enough as long as the cost of breaking it overwhelms that of breaking other components of the system, which are generally much more fragile: software correctness, user behavior.

It follows that different communities have different definitions of a "break" and of an "attack." This can cause misunderstandings, for example, when information is relayed in general media: In 2009 cryptographers showed how to recover a key [41] of the AES-192 block cipher within approximately 2^{176} evaluations of the cipher, instead of 2^{192} ideally. This attack does break AES-192 according to the definition of applied cryptographers, although it is clearly infeasible.[5] Nonetheless, several news sites published headlines such as "New AES Attack," which caused some users to believe that their AES-192 keys were at risk. Such results, reducing the theoretical complexity but leaving it impractically high, are sometimes called *certificational attacks*.

This book uses the definition of applied cryptographers, which is also that of NIST in its call for SHA3 submissions, namely, that SHA3 should achieve n-bit security against preimage attacks to be considered unbroken. In other words, any method substantially faster than the generic 2^n-time search is viewed as an attack. More generally, a function is considered to be broken when a method does "something" (such as finding multicollisions, input/output linear relations, etc.) more efficiently than the best generic attack, regardless of whether that "something" can be exploited to compromise a system's assets. The reasoning is that, if it can resist a nuclear bomb today, it can certainly resist a Molotov cocktail, or any explosive devices that may be crafted by an imaginative attacker in the next 20 years.

Recall the above informal definition of preimage resistance: "if it is hard (...) a given hash value." Observe that this raises (at least) two other issues:

- How can one be sure that the problem is indeed hard (whatever "hard" means)? and

[5] To give an order of magnitude, there are approximately $\approx 2^{166}$ atoms on Earth, and fewer than 2^{59} seconds have passed since the Big Bang.

- How is the hash value given to the attacker?

While the first question relates to deep questions in complexity theory, the second one will find an answer in the next section.

2.2.2 Classical Security Definitions

We now formally define the properties that a cryptographic hash should achieve to be called secure. We distinguish between unkeyed and keyed hash functions. The latter, denoted H_K, are parametrized by a secret key K held only by legitimate users; attackers who know H but not K cannot compute H_K. This is a simple way to simulate a *secret algorithm*, since hiding a 128-bit key is easier than hiding a program or algorithm, and generating a large number of keys is easier than generating a large number of algorithms with similar security properties.

Ideally, knowledge of H should not help an attacker who does not know K, compared with one who also does not know H. In both the unkeyed and keyed settings, a hash function is assumed to accept data of arbitrary length (up to some bound) as input, and to produce n-bit hash values; for instance, SHA1 formally accepts data of length up to $2^{64}-1$ bits (that is, almost 16,384 pebibytes) and produces 160-bit hash values.

2.2.2.1 Unkeyed Hash Functions

A rigorous definition of preimage resistance can be of mainly two kinds: range based, and codomain based, as described below.[6]

Definition 1 (Preimage resistance, range based). A hash function H is *preimage resistant* if, given a random n-bit string h, finding M such that $H(M) = h$ requires approximately 2^n evaluations of H.

Definition 2 (Preimage resistance, codomain based). A hash function H is *preimage resistant* if, given $H(M)$ for some unknown random input M, finding M' such that $H(M') = H(M)$ requires approximately 2^n evaluations of H.

The 2^n bound follows from the fact that finding preimages for an ideal hash function cannot be done with fewer than about 2^n evaluations of the function; that is, bruteforce[7] is optimal.

---

[6] A random function $f : \{0,1\}^n \to \{0,1\}^n$ has $\{0,1\}^n$ as codomain—the set of what may possibly come out of f—but a range that is a strict subset of $\{0,1\}^n$, since a significant number of n-bit strings would not admit preimages; in other words, f would not be *surjective*.

[7] Note that we talk of "bruteforce" rather than "exhaustive search," for the latter applies to (say) key recovery for a block cipher, but not to preimage search.

Definitions 1 and 2 only differ in the way the challenge value is chosen. It is easy to see that, if the hash function behaves like a random function, then the distribution of the challenge is the same in the two definitions, making them equivalent. However, both definitions are imperfect, because they can define functions that are obviously weak as preimage resistant; for example, consider the following hash functions H:

- For all inputs, H evaluates to the all-zero string. This function is preimage resistant according to Definition 1, but not according to Definition 2.
- For all n-bit inputs, H evaluates to the all-zero string; for other inputs, H behaves like a random function. This function is preimage resistant according to both definitions, but is clearly insecure.

Fortunately, these examples are pathological cases that are unlikely to be met by actual human-designed hash functions. The above definitions are thus sufficient for practical purposes, and the identification of insecure functions that may happen to satisfy those notions is left to common sense.

We now define second-preimage resistance and collision resistance.

Definition 3 (Second-preimage resistance). A hash function H is *second-preimage resistant* if, given $H(M)$ for some known random data M, finding $M' \neq M$ such that $H(M') = H(M)$ requires approximately 2^n evaluations of H.

Note that, unlike in the codomain-based version of preimage resistance, M' must be distinct from M. Again, finding second preimages has complexity approximately 2^n for an ideal function.

Definition 4 (Collision resistance). A hash function H is *collision resistant* if finding distinct M and M' such that $H(M) = H(M')$ requires approximately $2^{n/2}$ evaluations of H.

The $2^{n/2}$ bound follows from the well-known birthday attack, the key idea being that with $2^{n/2}$ values of $H(M)$, one can construct approximately 2^n candidate pairs for $H(M) = H(M')$. Note that collision search algorithms with complexity about $2^{n/2}$ can be implemented with only negligible memory, without storing all $2^{n/2}$ values (see Section 2.3).

2.2.2.2 Keyed Hash Functions

Keyed hash functions incorporate a secret key K of k bits, and thus can only be computed by parties knowing K. Keyed hash functions are at the basis of message authentication codes and of pseudorandom functions. Definitions of security of keyed hash functions substantially differ from and extend the classical definitions. For the sake of simplicity, we only give informal definitions, which assume that the attacker is able to query H_K as a black box to obtain $H_K(M)$ for the messages of its choice:

Definition 5 (Pseudorandomness). A hash function H is *pseudorandom* if, for a random K, distinguishing H_K from a random function needs approximately 2^k queries to H_K.

Definition 6 (Unpredictability). A hash function H is unpredictable if for a random K, finding $H_K(M)$ for some unqueried M needs approximately 2^k queries to H_K.

Those two notions are similar but not identical: pseudorandomness implies unpredictability, but not the other way around.

2.2.3 General Security Definition

A general definition of a secure hash function is *a function that behaves like an ideal hash function*, which is, admittedly, a bit tautological. A less imprecise definition is given by Ferguson, Schneier, and Kohno [67]:

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

In other words, if one can "do something" for a hash function that one cannot do with the same (or lesser) effort for an ideal hash function (or for any other hash function), then this "distinguishes" it from an ideal one. The method employed is called a *distinguisher*; for example, a method to find preimages in 2^{n-4} is an attack, for ideal hash functions only admit preimage attacks in 2^n. More generally, a method more efficient than the best *generic attack*—i.e., one that works for any hash function—is a distinguisher.

There are some caveats, though:

- First, any hash function specified as an algorithm (rather than as an abstract "oracle") admits a trivial distinguisher because there exists a compact expression—namely, the algorithm—of the output as a function of the input. For an ideal hash function, such an expression is unlikely to exist. Actually, the most compact representation of a random hash function has exponential length; in other words, the program would not even fit in a computer's memory.
- Second, many distinguishers do not impact the actual security of the hash function, suggesting that the ideal hash function considered is too high an ideal for any practical purpose.
- Third, distinguishers are difficult to rigorously define formally, and there is no standard definition accepted by the community. Nevertheless, a distinguisher is generally an "elephant" test: you recognize it when you see it.

Whatever the goal of an attack, it should be compared with the generic method not only in terms of computational complexity, but also of memory requirements, probability of success, parallelism, and more generally in terms of cost-effectiveness when actually implemented (in the physical world).

The general definition of security has the advantage of capturing all security-critical properties such as preimage resistance, but a hash function that fails to satisfy it is not necessarily insecure. In cryptography theory, however, so-called security proofs of schemes that use hash functions as an underlying primitive generally assume that hash functions do satisfy that general definition. Some thus argue that it is risky to use a nonideal hash function in such a provably secure scheme. For more about this, we refer the reader to the notion of *indifferentiability* and its related literature (e.g. [55, 126, 154]).

2.3 Black-Box Collision Search

Generic collision search methods are one of the most interesting problems related to hash functions, due in part to the elegance of the techniques. Such methods do not depend on the internals of the hash functions, and rather view them as black boxes assumed to behave as random functions. Below we describe state-of-the-art methods applicable to cryptographic hash functions as well as to any function that behaves sufficiently randomly. Such functions include the core functions of public-key schemes based on factoring or discrete logarithms.

The general collision search problem is, given a function F with a finite range, to find distinct inputs x and x' such that $F(x)$ equals $F(x')$. Collision search is an important tool in cryptanalysis, most notably to evaluate the security of discrete logarithm-based schemes, to perform meet-in-the-middle attacks, or to find collisions in components of hash functions.

We refer to Joux's book [91, Chaps. 6–8] for a detailed overview of a comprehensive review of collision search methods.

2.3.1 Cycles and Tails

Let $\mathscr{F}_n$ denote the set of all functions from a domain D of size n to a codomain of size n, with n finite; for example, if D consists of all b-bit strings, then $n = 2^b$. Let F be a random element of $\mathscr{F}_n$—that is, a random mapping from and to n-element sets. A folklore result is that the range of F is expected to contain $n(1-1/e) \approx 0.63n$ *distinct* elements. Therefore, F is expected to have *collisions* $F(x) = F(x')$, $x \neq x'$. Efficient methods for finding such collisions exploit the structure of F as a collection of *cycles*.

Consider the infinite sequence $\{x_i = F(x_{i-1})\}_{0<i}$, for some arbitrary starting value x_0. Because D is finite, this sequence will eventually begin to cycle, that is, to repeat an identical sequence indefinitely. Hence, there exist two smallest integers $\mu \geq 0$ (the *tail length*) and $\lambda \geq 1$ (the *cycle length*) such that $x_i = x_{i+\lambda}$ for every $i \geq \mu$. Such a structure then yields a collision at the point where the cycle begins: $F(x_{\mu-1}) = F(x_{\mu+\lambda-1}) = x_\mu$.

The *birthday paradox* illustrates well the above structure: in a sequence of random numbers in $\{1,\ldots,n\}$, the expected number of draws before a number occurs twice is asymptotically $\sqrt{\pi n/2} \approx 1.25\sqrt{n}$. This is because the expected values of the tail length μ and of the cycle length λ sum to $\sqrt{\pi n/8}+\sqrt{\pi n/8}=\sqrt{\pi n/2}$. This value is sometimes called the *rho length*, because of the rho shape of the sequence, as noticed by Pollard [146].

A trivial collision search algorithm repeats the following: pick random points x and x', return them as a collision if $F(x)$ equals $F(x')$, otherwise pick another pair of random points. About n trials are required, since x and x' collide with probability $1/n$. A less trivial algorithm exploits the existence of cycles by storing a sequence $\{x_i = F(x_{i-1})\}_{0<i<\sqrt{\pi n/2}}$, sorting it, and looking for a collision. State-of-the-art methods eliminate the large memory requirements and the cost of sorting a large list. In the following we review these methods, starting with explicit cycle-detection methods, then presenting modern techniques optimized for efficiency on parallel computing infrastructure. Finally, we explain how to apply these methods to concrete cryptanalytic problems.

2.3.2 Cycle Detection

The low-memory cycle-detection method of Floyd is at the base of Pollard's rho method for factoring and computing discrete logarithms.[8] It is based on the following observation [108, 3.1, Ex. 6]:

Theorem 1. *For a periodic sequence $x_0, x_1, x_2, \ldots$, there exists a unique $i > 0$ such that $x_i = x_{2i}$ and the smallest such i lies in the range $\lambda \leq i \leq \lambda + \mu$.*

The values λ and μ are the cycle length and tail length, as defined in Section 2.3.1.

Based on Theorem 1, Floyd's method picks a starting value $x_0 = x'_0$ and compares the values $x_i = F(x_{i-1})$ and $F(F(x'_{i-1}))$, $i \geq 1$. The expected number of iterations before reaching a match is $\sqrt{\pi^5 n/288} \approx 1.03\sqrt{n}$ [19].

Floyd's algorithm detects that the sequence has reached a cycle but does not return the values of λ and μ, nor a collision for F. This can be done as follows, once $x_i = x_{2i}$ is found: generate x_j and x_{j+i}, $j \geq 0$, until finding $x_j = x_{j+i}$; at the first equality we have $j = \mu$. If none of the values x_{j+i} equals x_i then $\lambda = \mu$, otherwise λ is the smallest such j. This operation costs on average $2\sqrt{\pi n/2}$ evaluations of F. Finally, detecting the cycle and locating the collision with the above method costs

$$3\sqrt{\pi^5 n/288} + 2\sqrt{\pi n/2} \approx (3.09 + 2.51)\sqrt{n} = 5.60\sqrt{n}$$

[8] "Floyd's algorithm" was actually first described in [108], and credited to Floyd without citation. Floyd's 1967 paper [70] describes an algorithm for listing cycles in a directed graph, but that differs from the cycle-detection algorithm considered here.

evaluations of F, and requires negligible memory (storage of a few x_i's). Slightly more efficient variants of Floyd's algorithm were proposed by Brent [19] and Teske [169]. Sedgewick et al. showed how to eliminate the redundant computations by using a small amount of memory [161], but their algorithm is not as general as Floyd's (in particular, it cannot be combined with Pollard's rho factoring method).

2.3.3 Parallel Collision Search

A disadvantage of Floyd's algorithm (and thus of Pollard's rho method) is that it cannot be parallelized efficiently: m processors (e.g., CPU cores) do not provide a $1/m$ reduction of complexity. In other words, speed does not grow linearly with the number of processors. This is because one has to wait for a given invocation of F to end before the next can begin. Efficient parallelization of collision search takes a different approach, by using *distinguished points*. The idea of using distinguished points (i.e., points that have some predefined easily checkable property, such as having ten leading zero bits) was proposed by Quisquater and Delescaille [151] for searching collisions for the Data Encryption Standard (DES), and earlier noted by Rivest [60, p.100] in the context of Hellman's time-memory tradeoff. Below we describe a simple method for efficient parallelization of collision search using distinguished points, due to van Oorshot and Wiener [140].

Let m be the number of processors available, and consider some easily checked property $\mathsf{P} \subsetneq \mathsf{D}$ that a random point satisfies with probability $\theta < 1$. To perform the search, each processor runs the following procedure:

1. Select a starting value x_0;
2. Iterate $x_i = F(x_{i-1})$, $i > 0$, until a distinguished point $x_d \in \mathsf{P}$ is reached;
3. Add x_d (along with x_0 and d) to a common list for all processors;
4. Repeat the process (that is, go to 1.).

The algorithm halts when a same distinguished point appears twice in the common list, which means that two distinct sequences $(x_0, \ldots, x_i)$ and $(x'_0, \ldots, x'_j)$ lead to a same value $x_i = x'_j$ (one should ensure that a same starting value is not used twice). With high probability, one will easily deduce a collision from these two sequences (if the first sequence leads to the starting point of the second, then no collision will be found). Details can be found in [140].

The above algorithm runs in time about $\sqrt{\pi n/2}/m + 2.5/\theta$ to locate a collision, hence parallelism provides a linear speedup of the search.

2.3.4 Application to Meet-in-the-Middle

Parallel collision search using distinguished points can be directly applied to find collisions for hash functions. It can also be adapted to compute discrete logarithms

in cyclic groups. Here we show how it can be used to perform meet-in-the-middle (MitM) attacks.

The problem considered is, given two functions F_1 and F_2 in $\mathscr{F}_n$, to find x and x' (not necessarily distinct) such that $F_1(x)$ equals $F_2(x')$. A solution can be found by defining an easily checked property P, and by considering the function

$$F(x) = \begin{cases} F_1(x) & \text{if } x \in \mathsf{P} \\ F_2(x) & \text{otherwise} \end{cases}.$$

Under reasonable assumptions on F_1 and F_2, and assuming that a random x satisfies P with probability $1/2$, a collision $F(x) = F(x')$ will be useful as soon as x satisfies P but x' does not—meaning that the collision is between F_1 and F_2. When the cost of computing F_1 and F_2 significantly differs (for example, if one of them represents a shortcut preimage attack on some component), the property P can be adapted to optimize the complexity of the attack, so that F_1 is called more often than F_2, for example (e.g., if F_2 is slower to evaluate).

Note that the MitM problem considered here, and often encountered in cryptanalysis, differs from what is called MitM in [140]. Indeed, the latter attack looks for a single "golden value," and its complexity heavily depends on the domain size, whereas in the former the complexity only depends on the range size.

2.3.5 *Quantum Collision Search*

Although they do not exist (yet), and are sometimes believed to be physically impossible to construct (see for example [117]), quantum computers do represent a potential threat for cryptography. Indeed, efficient quantum algorithms exist for factoring integers and solving discrete logarithms, two problems whose alleged hardness guarantees the security of RSA, DSA, Diffie–Hellman, elliptic-curve cryptography, etc. Solutions in a world with efficient quantum computers are proposed in [27].

Symmetric cryptography is also concerned, to a lesser extent, with quantum attacks: using quantum Fourier transform and Grover's algorithm [73], a quantum search algorithm can recover an n-bit key in time about $2^{n/2}$, with negligible memory. This would require to double the length of hash values for the same preimage resistance.

Finding a (black-box) collision with a quantum algorithm takes $\Omega(2^{n/3})$ queries [1, 111]. The quantum search algorithm was adapted by Brassard, Høyer, and Tapp [48] to find collisions in time $O(2^{n/3})$, but it requires space $O(2^{n/3})$ of read-only quantum memory (for a detailed cost analysis, see [26]). This makes quantum collision search significantly *less efficient* than classical parallel search, which needs space of only $O(2^{n/6})$ to find collisions in time $O(2^{n/3})$. Therefore, quantum computers are unlikely to be a major threat as far as collisions are concerned.

2.4 Constructing Hash Functions

All general-purpose hash functions split the data to be hashed into blocks of fixed length and process them iteratively using a *compression function*. The compression function takes fixed-length input and produces fixed-length output. The combination of calls to a compression function to process arbitrary-length input is called an *iteration mode*.

We present the classical iteration mode used by MD5, SHA1, and SHA2 (the so-called Merkle–Damgård construction) as well as the state-of-the-art modes used by most recent designs, such as SHA3 candidates. Finally, we review constructions of compression functions based on block ciphers.

2.4.1 Merkle–Damgård

The Merkle–Damgård (shorthand MD) iteration mode proceeds in two steps: first, a preprocessing of the data to hash (the *padding* step), then the actual processing of the data. Below we describe those two steps and present some properties of the MD construction.

Note that the details of the MD construction may slightly vary in the literature. Here we describe how it is used in SHA1 and SHA2, as per the NIST standard [137].

2.4.1.1 Data Padding

The data to hash can be of arbitrary bit length. However, the iteration mode processes blocks of m bits. It is thus necessary to transform the data received into a sequence of m-bit blocks in an invertible way, so as to avoid trivial collision; in other words, the original data should be uniquely defined given the data after padding. We shall henceforth refer to the data before padding as the "original data," and to the data after padding as the "padded data."

In SHA1, SHA-224, and SHA-256, the block length m is 512 bits. Padding of ℓ-bit data proceeds as follows:

1. Append a "1" bit to the end of the data;
2. Append $k \geq 0$ "0" bits, where k is the smallest solution to the equation $\ell + 1 + k \equiv 448 \bmod 512$;
3. Append a 64-bit unsigned big-endian representation of ℓ.

This procedure guarantees that the bit length of the padded data is a multiple of 512.

In SHA-384 and SHA-512 m is 1,024 bits. Padding is similar, except that k should satisfy $\ell + 1 + k \equiv 896 \bmod 1{,}024$ and that ℓ is represented on 128 bits.

2.4.1.2 Data Processing

After data padding, a MD hash function processes a sequence of blocks $M_0, \ldots, M_{N-1}$ using a compression function compress by doing

$$h := \mathsf{compress}(M_i, h) \quad \text{for } i = 0, \ldots, N-1 \ ,$$

where the *chaining value* h is initialized to some predefined *initial value* (IV). The hash value returned is the final value of h obtained after processing M_{N-1}.

In SHA1, h is 160-bit and the IV is the four 32-bit words

```
67452301 efcdab89 98badcfe 10325476 c3d2e1f0 .
```

2.4.1.3 Security Properties

We summarize the main security properties of the MD mode.

Security Reductions

It can be shown that a collision $H(M) = H(M')$ on a MD hash function always implies a collision $\mathsf{compress}(h, M_i) = \mathsf{compress}(h', M_i')$ for the underlying compression function. One can thus reduce[9] the collision resistance of the hash function to that of its compression function. We call *internal collision* any collision for the compression function that occurs before processing the last data block.

A similar reduction exists with respect to preimage resistance; clearly, if one can find preimages of the hash function, then one can also find preimages of the compression function.

If padding of the data length is omitted, the collision resistance reduction no longer holds. To show this, suppose one knows a data block M_0 such that

$$\mathsf{compress}(h, M_0) = h$$

when h is set to the IV. It follows that, for any data M, the strings $M_0 \| M$ and $M_0 \| M_0 \| M$ hash to the same value.

Multicollision Attack

Observe that, in the MD mode, the chaining values are of the same length as the hash value. One can thus search for a collision of chaining values at the same cost as for the hash value, namely $2^{n/2}$ calls to the compression function, approximately.

[9] In the sense of a complexity reduction.

As discovered by Joux [90], one can find a *multicollision*[10] for a MD hash function with much less effort than ideally, by proceeding as follows:

1. Find blocks M_0^0 and M_0^1 such that $\mathsf{compress}(h, M_0^0) = \mathsf{compress}(h, M_0^1)$, for h set to the IV.
2. Find blocks M_1^0 and M_1^1 such that $\mathsf{compress}(h, M_1^0) = \mathsf{compress}(h, M_1^1)$, for h set to $\mathsf{compress}(h, M_0^0)$.
3. Repeat the procedure for N blocks, to obtain in total N pairs (M_i^0, M_i^1).

Thus all strings of the form $M_0^\star \| M_1^\star \| \cdots \| M_{N-1}^\star$, where $\star$ is a wildcard symbol for either 0 or 1, will hash to the same value. As there are 2^N such strings, we call them a 2^N-multicollision. The computational effort is about $N \cdot 2^{n/2}$ calls to the compression function.

This multicollision attack is only feasible in practice if it is feasible to find a collision in the first place. A collision-resistant function is thus also multicollision resistant.

Fixed Points and Second Preimages

In the "Security Reductions" paragraph above we explained how the data length padding thwarts an attack exploiting one fixed point $h = \mathsf{compress}(h, M)$. What about two fixed points? One may exploit two fixed points in such a way that the two forged inputs have the same length, that is, such that (any) fixed points are iterated the same number of times for both. This is the idea behind Dean's second preimage attack [59] on MD functions.

Dean's idea to bypass the length padding was improved by Kelsey and Schneier, who exploited Joux's multicollision trick to produce "expandable messages" (we refer to [100] for details of the attack). Using this attack, second preimages of 2^k-block messages can be found in time approximately 2^{n-k}, instead of 2^n ideally.

Length Extension

The length extension property allows one to determine the hash value of $M = M_p \| P \| M_s$ given only the hash value $H(M_p)$ and the suffix M_s. Here P stands for the padding data appended to M_p when computing $H(M_p)$. To find $H(M)$, it thus suffices to compute $\tilde{H}(M_s)$, where $\tilde{H}$ is identical to H except that it uses the value $H(M_p)$ as an IV.

One can use length extension in forgery attacks against MACs computed as $H(K \| M)$, where K is the key and M is the data to authenticate. This construction is known as *prefix MAC*, and is thus insecure when combined with a MD hash function.

[10] A k-multicollision is a set of k distinct inputs that hash to the same value.

2.4.2 HAIFA

The HAsh Iterative FrAmework (HAIFA) [35] is an enhanced version of the MD iteration mode that aims at improving security and flexibility. The main differences from MD are that:

- The compression function in HAIFA takes as additional inputs a salt and a counter of the number of bits already processed;
- The IV and the padding depend on the length of the hash value, which is of variable size (but not longer than the chaining value).

The counter foils attacks based on fixed points, including the second-preimage attack of Kelsey and Schneier. The built-in salt encourages and simplifies the use of a salt in password storage and digital signatures.

BLAKE uses a simplified version of HAIFA that retains all of HAIFA's desirable properties.

2.4.3 Wide-Pipe

The so-called wide-pipe [122] mode is similar to the MD mode except that the chaining value is larger than the returned hash value. A second compression function must thus be used to produce the final digest from the last chaining value—this function can be as simple as a simple truncation of a subset of the bits.

The wide-pipe mode mitigates attacks based on internal collisions, such as Joux's multicollisions or Kelsey and Schneier second preimages. This comes at a price, however: because the internal state is larger, more memory is necessary, and potentially more computation in order to achieve the same security as with a smaller chaining value.

The compression function BLAKE uses a construction that was called *local wide-pipe*, for it creates a larger internal state within the compression function, whereas chaining values have the same length as the final digest.

2.4.4 Sponge Functions

Sponge functions [29, 30] use a construction that deviates from the compression-based MD mode: given a chaining value h, the new chaining value is computed as $P(h \oplus M_i)$, where the data block M_i is significantly smaller than h and where P is a permutation, which may be efficiently invertible.

As depicted in Figure 2.1, a sponge function consists of a state of width $b = r + c$ bits, where:

- r is called the *rate*, and corresponds to the data block size;

- c is called the *capacity*, and defines the security level, being approximately $c/2$ bits.

The sponge function then modifies the internal state by a sequence of data block injections followed by an application of a permutation function (which may also be a noninvertible transform).

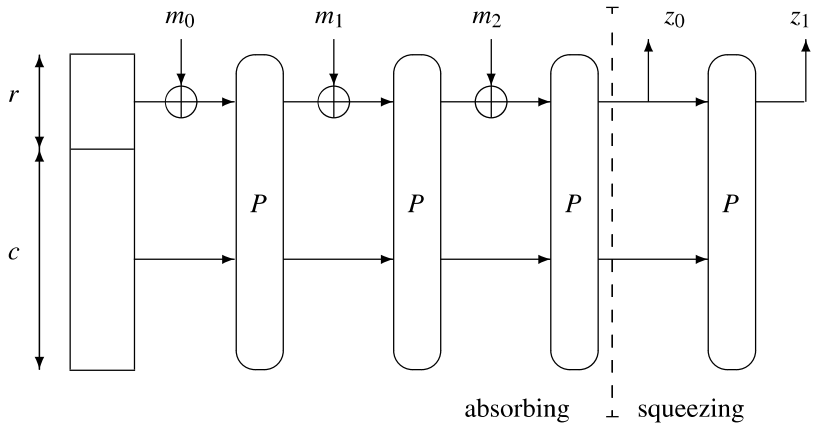


Fig. 2.1 The sponge construction, for the example of a 4-block ($4r$-bit) padded message.

Like the MD mode, sponge functions benefit from security reductions; for example, it has been proven that the function behaves ideally if the underlying permutation behaves ideally (cf. 2.2.3). An advantage of sponge functions is their flexibility: it is straightforward to vary parameters to achieve various efficiency/security tradeoffs. Examples of sponge functions are the SHA3 candidate Keccak and the lightweight hash function QUARK [14].

2.4.5 Compression Functions

Compression functions are the main building block of Merkle–Damgård and wide-pipe hash functions. As their name suggests, they return fewer bits than input bits received. However, unlike file compression methods, they do not aim to retain the information from the input (which would imply at least partial invertibility). Contrary to that, they aim to behave as random functions, thus eliminating any structure in their input values.

One strategy to construct compression functions is to *reuse block ciphers*—that is, keyed permutation, invertible transforms—to create a noninvertible transform. The main motivations for this approach are:

- **Confidence**: If the security of a hash function is reducible to that of its underlying block cipher, then using a well-analyzed block cipher gives more confidence than a new algorithm.

- **Compact implementations**: The code used for encryption with the block cipher may be reused by the hash function, thus reducing the space occupied by the cryptographic components in a program.

Another significant advantage specific to the reuse of AES is *speed*: native AES instructions in recent AMD, ARM, or Intel processors significantly speed up AES, and hash functions may take advantage of this. Besides faster execution, native AES instructions also avoid the risk of cache-timing attacks, as demonstrated on table-based implementations of AES.

Counterarguments to block cipher-based hashing are:

- **Structural problems**: Generally the block and key lengths of block ciphers do not match the values required for hash functions; e.g., AES uses 128-bit blocks, whereas a general-purpose hash function should return digests of at least 224 bits. One thus has to use constructions with several instances of the block cipher, which is less efficient.
- **Slow key schedule**: The initialization of block ciphers is typically slow, which motivates the use of fixed-key permutations rather than families of permutations. However, results indicate that this approach cannot yield compression functions that are both efficient and secure [43,155,156]. A proposal for fixing this problem was to use a *tweakable* block cipher [121], where an additional input, the tweak, is processed much faster than a key.

We now briefly summarize the historical development of block cipher-based hashing.

The idea of making hash functions out of block ciphers dates back at least to Rabin [153], who proposed in 1978 to hash $(m_1, \ldots, m_\ell)$ as

$$\mathsf{DES}_{m_\ell}(\ldots(\mathsf{DES}_{m_1}(\mathrm{IV})\ldots)\,.$$

Subsequent works devised less straightforward schemes, with one or two calls to the block cipher within a compression function [112, 125, 129, 148, 152].

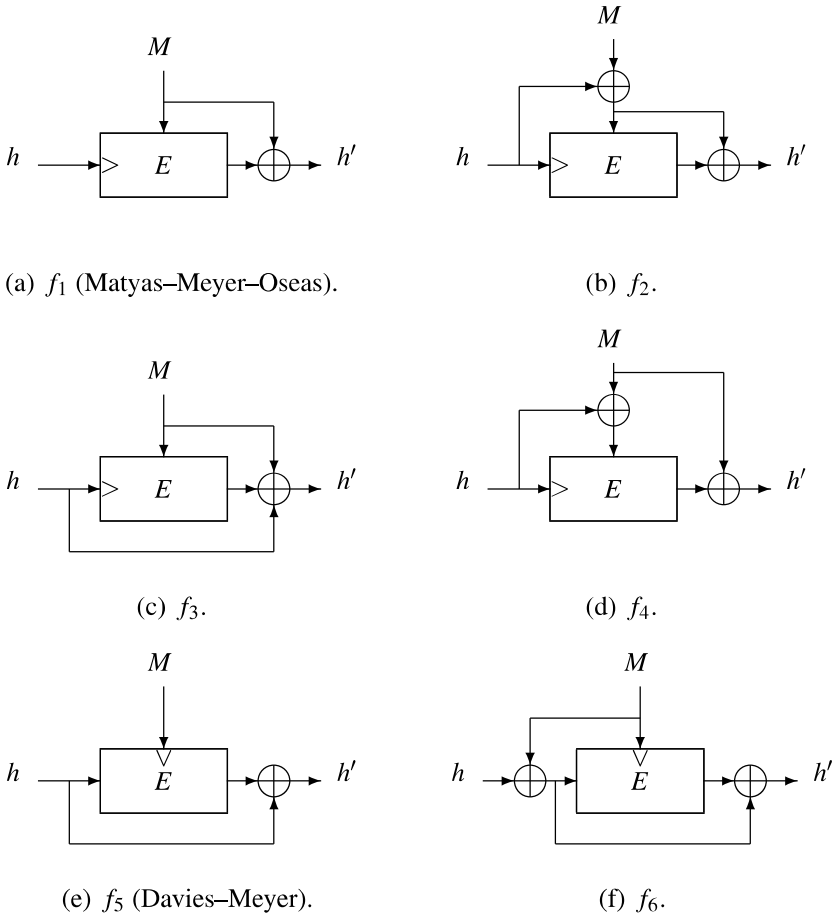
In 1993, Preneel, Govaerts, and Vandewalle (PGV) [149] considered 64 block cipher-based compression functions and identified 12 secure ones,[11] including

$$\begin{aligned}
&E_h(M) \oplus M\\
&E_h(h \oplus M) \oplus h \oplus M\\
&E_h(M) \oplus h \oplus M\\
&E_h(M) \oplus M\\
&E_M(h) \oplus h\\
&E_M(h \oplus M) \oplus h \oplus M
\end{aligned}$$

where $E_h(M)$ denotes encryption of the data block M with h as a key (see Figure 2.2). The fifth construction in this list is known as the Davies–Meyer construction and is used by MD5, SHA1, and SHA2. Note that some of the constructions implicitly assume that M and h have the same length. Some constructions have the

[11] A formal analysis of the security of these constructions can be found in [45].

undesirable property of easily found fixed points; for example, a fixed point for the construction $E_M(h) \oplus h$ can be found by choosing an arbitrary M and computing $h' := E_M^{-1}(0)$, leading to $E_M(h') \oplus h' = h'$ (see Section 2.4.1.3 for attacks exploiting fixed points).



(a) f_1 (Matyas–Meyer–Oseas). (b) f_2.

(c) f_3. (d) f_4.

(e) f_5 (Davies–Meyer). (f) f_6.

Fig. 2.2 Block cipher-based compression functions f_1 to f_6 by Preneel et al. [149], where a mark denotes the key input (we assume keys and message blocks of the same size).

Note that the so-called PGV schemes cannot be proved collision resistant under the pseudorandom permutation (PRP) assumption only; to see this, take a block cipher E and construct the block cipher $\tilde{E}$ as

$$\tilde{E}_k(m) = \begin{cases} k & \text{if } m = k \\ E_k(k) & \text{if } m = E_k^{-1}(k) \\ E_k(m) & \text{otherwise} \end{cases} .$$

If the Matyas–Meyer–Oseras (MMO) construction [125] $E_{h_{i-1}}(m_i) \oplus m_i$ is instantiated with $\tilde{E}$, then collisions are easy to find, yet $\tilde{E}$ inherits the PRP property from E.

BLAKE uses a construction similar to $E_M(h) \oplus h$, but where the block cipher is replaced by a noninvertible function—itself built on a block cipher. This makes fixed points difficult to find, among other properties.

Examples of pre-SHA3 designs based on block ciphers are Whirlpool [20], Maelstrom [68], and Grindahl [107] (subsequently broken [144]), which all build on AES. Some submissions to the SHA3 competition were based on AES: ECHO, Fugue, LANE, SHAMATA, SHAvite-3, and Vortex, to name a few. They all use an ad hoc construction to make a compression function out of AES. However, the security of AES as a block cipher is not always sufficient for the security of the compression function; for example, SHAMATA and Vortex were broken [12, 85] (ironically, one attack on Vortex works because AES is a good block cipher).

2.5 The SHA Family

This section gives a brief overview of the NIST-approved hash functions SHA1 and SHA2, and of the SHA3 candidates. Below we copy NIST's 2006 statement regarding the use of SHA1 and SHA2 [131]:

> The SHA-2 family of hash functions (i.e., SHA-224, SHA-256, SHA-384 and SHA-512) may be used by Federal agencies for all applications using secure hash algorithms. Federal agencies should stop using SHA-1 for digital signatures, digital time stamping and other applications that require collision resistance as soon as practical, and must use the SHA-2 family of hash functions for these applications after 2010. After 2010, Federal agencies may use SHA-1 only for the following applications: hash-based message authentication codes (HMACs); key derivation functions (KDFs); and random number generators (RNGs). Regardless of use, NIST encourages application and protocol designers to use the SHA-2 family of hash functions for all new applications and protocols.

In spite of that recommendation, SHA1 remains widely used, either for legacy reasons, efficiency reasons, or acceptance of the risk—which may or may not be justified.

2.5.1 SHA1

The NIST standard SHA1 [137] was designed by the US National Security Agency (NSA) and published in 1995. SHA1 superseded SHA0, a function almost identical to SHA1 published in 1993 and later withdrawn. Later, in 1998, a collision for SHA0 was published [49].

SHA1 produces 160-bit digests, and is thus expected to provide 80-bit security against collision attacks. As of 2013, SHA1 is with MD5 the most widely used hash function.

2.5.1.1 Internals

SHA1 follows the MD construction. The compression function of SHA1 processes 512-bit data blocks and 160-bit chaining values. It initializes an internal state of five 32-bit words a, b, c, d, e with the current chaining value, and transforms it by repeating a step function 80 times and XORing the final state with the initial state. The step function for the first 16 steps does

$$\begin{aligned}
&\mathsf{ch} = (b \wedge c) \vee ((\neg b) \wedge d) \\
&\mathsf{temp} = \mathsf{ch} + e + m_i + \mathtt{5A827999} \\
&e = d \\
&d = c \\
&c = (b \lll 30) \\
&b = a \\
&a = (a \lll 5) + \mathsf{temp}
\end{aligned}$$

where m_i is the ith word of the data block $i = 0, \ldots, 15$. The subsequent steps of SHA1 use different Boolean functions to compute ch, different constants to compute temp, and words w_i of the *expanded* data block computed as

$$w_i := (w_{i-3} \oplus w_{i-8} \oplus w_{i-14} \oplus w_{i-16}) \lll 1, \quad i = 16, \ldots, 79$$

with $w_i := m_i$ for $i = 0, \ldots, 15$.

2.5.1.2 Security

The first reported attack on the full SHA1 was a collision attack with complexity 2^{69} by Chinese researcher Xiaoyun Wang and her team [174], in 2005. As of 2011, attacks with complexity as low as 2^{57} [123] and 2^{52} [127] have been claimed. However, the refined complexity analysis by Manuel [124] argued that these estimates were too optimistic, and that the most efficient attack known then [93] may have complexity between 2^{65} and 2^{69}. In 2013, Stevens' refined analysis [165] led to a collision attack with estimated complexity of 2^{61}.

2.5.2 SHA2

The NIST standard SHA2 [137] was designed by the US National Security Agency (NSA) and published in 2001. SHA2 is a family of four hash functions: SHA-224, SHA-256, SHA-384, and SHA-512, which produce digests of bit size equal to their respective suffixes. SHA2 is supported by an increasing number of products, with SHA-256 being the most common instance.

2.5.2.1 Internals

Like SHA1, all functions of the SHA2 family follow the Merkle–Damgård construction. SHA-256 and SHA-512 are the two main instances of the family. They respectively work on 32- and 64-bit words, and thus use distinct algorithms. SHA-224 and SHA-384 are derived from SHA-256 and SHA-512, by using distinct initial values and truncating the final output.

The compression function of SHA-256 processes 512-bit data blocks and 256-bit chaining values. It initializes an internal state of eight 32-bit words a,b,c,d,e,f,g,h with the current chaining value, and transforms it by repeating a step function 64 times and XORing the final state with the initial state. The step function of SHA-256 does the following:

$$\begin{aligned}
\sigma_0 &= (a \ggg 2) \oplus (a \ggg 13) \oplus (a \ggg 22)\\
\sigma_1 &= (e \ggg 6) \oplus (e \ggg 11) \oplus (e \ggg 25)\\
\mathsf{maj} &= (a \wedge b) \oplus (a \wedge c) \oplus (b \wedge c)\\
\mathsf{ch} &= (e \wedge f) \oplus ((\neg e) \wedge g)\\
\mathsf{temp0} &= \sigma_0 + \mathsf{maj}\\
\mathsf{temp1} &= \sigma_1 + \mathsf{ch} + \mathsf{const}_i + h + w_i\\
h &= g\\
g &= f\\
f &= e\\
e &= d + \mathsf{temp1}\\
d &= c\\
c &= b\\
b &= a\\
a &= \mathsf{temp0} + \mathsf{temp1}
\end{aligned}$$

The values const_i are predefined step-dependent constants, and w_i's are words of expanded data block. For $i = 0,\ldots,15$, w_i is equal to the data block word m_i, and for $i = 16,\ldots,63$ w_i is recursively defined as

$$\begin{aligned}
s_0 &:= (w_{i-15} \ggg 7) \oplus (w_{i-15} \ggg 18) \oplus (w_{i-15} \gg 3)\\
s_1 &:= (w_{i-2} \ggg 17) \oplus (w_{i-2} \ggg 19) \oplus (w_{i-2} \gg 10)\\
w_i &:= w_{i-16} + s_0 + w_{i-7} + s_1
\end{aligned}$$

The compression function of SHA-512 uses 64-bit words, and thus processes 1,024-bit blocks and 512-bit chaining values.

2.5.2.2 Security

No attack is known on any of the four SHA2 instances. The best known results are attacks on versions with a reduced number of steps: in 2009, Aoki, Guo, Matusiezicz, Sasaki, and Wang [5] described preimage attacks on 43-step SHA-256 and 46-step SHA-512 that are twice as fast as bruteforce. The best known colli-

sion attacks [84] find collisions on 24-step SHA-256 and SHA-512 with respective complexities of $2^{28.5}$ and 2^{53}.

2.5.3 SHA3 Finalists

We give a brief overview of the other SHA3 finalists, highlighting their unique qualities and strengths compared with other algorithms.

2.5.3.1 Grøstl

Designed by a team of seven active cryptanalysts from the Technical University of Graz, Austria and from the Technical University of Denmark, Grøstl is the only SHA3 finalist that reuses components of the AES. Grøstl follows a wide-pipe MD construction, with a compression function returning

$$P(h \oplus M) \oplus Q(M) \oplus h \ ,$$

where P and Q are two permutations inspired by the AES, and h and M have the same length (512 or 1,024 bits). The security of Grøstl was extensively analyzed by its designers as well as third parties, leading to a well-understood design. These were the main reasons for its selection as a finalist.

2.5.3.2 JH

JH was designed by Hongjun Wu, from Nanyang Technologial University, Singapore. JH is the only SHA3 finalist to use 4-bit Sboxes combined with a linear diffusion layer, in the spirit of the AES finalist Serpent. The chopped wide-pipe MD construction of JH relies on a novel compression function that computes

$$E\left(h \oplus (M \| 0 \cdots 0)\right) \oplus (0 \cdots 0 \| M) \ ,$$

where E is a permutation, h is 1,024-bit, and M as well as the strings of zeroes are 512-bit. Due to its bit-oriented structure, JH is a good performer in hardware implementations. JH made the finals due to its security margin, all-round performance, and innovative design.

2.5.3.3 Keccak

Keccak is a creation of Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche, from the semiconductor companies STMicroelectronics and NXP. Keccak is a sponge function based on a bit-oriented permutation. It is thus very fast in

hardware implementations. Keccak was selected due to its security margin, high throughput, and simplicity of design.

2.5.3.4 Skein

Skein is the brainchild of a team of eight researchers and engineers from industry (BT, Hifn, Intel, Microsoft, PGP) and academia (Bauhaus-Universität Weimar, UCSD, University of Washington). Skein uses a construction similar to HAIFA, and builds on a compression function based on a tweakable block cipher in modified MMO mode

$$E_h(M) \oplus M \ ,$$

where E is a tweakable block cipher (called Threefish) using only modular addition, rotation, and XOR. Due to its construction optimized for 64-bit software architectures, Skein is the best performer on high-end desktop and server platforms. Skein was selected as a finalist due to its security margin and speed in software.