Chapter 3

Hash Functions and MACs

Hash functions are important tools in cryptography. In public key cryptography, they are used in key derivation functions, digital signatures and message authentication codes. We are unable to give a thorough presentation of hash functions. Instead, we refer to Chapter 4 of Katz and Lindell [334], Chapter 9 of Menezes, van Oorschot and Vanstone [418], Chapter 4 of Stinson [592] or Chapter 3 of Vaudenay [616].

3.1 Security Properties of Hash Functions

Definition 3.1.1. A **cryptographic hash function** is a deterministic algorithm H that maps bitstrings of arbitrary finite length (we denote the set of arbitrary finite length bitstrings by $\{0,1\}^*$) to bitstrings of a fixed length l (e.g., $l = 160$ or $l = 256$). A **cryptographic hash family** is a set of functions $\{H_k : k \in \mathbb{K}\}$, for some finite set $\mathbb{K}$, such that each function in the family is of the form $H_k : \{0,1\}^* \rightarrow \{0,1\}^l$.

The value k that specifies a hash function H_k from a hash family is called a key, but in many applications the key is not kept secret (an exception is message authentication codes). We now give an informal description of the typical security properties for hash functions.

1. **Preimage-resistance**: Given an l-bit string y it should be computationally infeasible to compute a bitstring x such that $H(x) = y$.
2. **Second-preimage-resistance**: Given a bitstring x and a bitstring $y = H(x)$ it should be computationally infeasible to compute a bitstring $x' \neq x$ such that $H(x') = y$.
3. **Collision-resistance**: It should be computationally infeasible to compute bitstrings $x \neq x'$ such that $H(x) = H(x')$.

In general one expects that for any $y \in \{0,1\}^l$ there are infinitely many bitstrings x such that $H(x) = y$. Hence, all the above problems will have many solutions.

To obtain a meaningful definition for collision-resistance it is necessary to consider hash families rather than hash functions. The problem is that an efficient adversary for collision-resistance against a fixed hash function H is only required to output a pair $\{x, x'\}$ of messages. As long as such a collision exists then there exists an efficient algorithm that outputs one (namely, an algorithm that has the values x and x' hard-coded into it). Note that there is an important distinction here between the running time of the algorithm and the running time of the programmer (who is obliged to compute the collision as part of their task). A full discussion of this issue is given by Rogaway [500]; also see Section 4.6.1 of Katz and Lindell [334].

Intuitively, if one can compute preimages then one can compute second-preimages (though some care is needed here to be certain that the value x' output by a pre-image oracle is not just x again; Note 9.20 of Menezes, van Oorschot and Vanstone [418] gives an artificial hash function that is second-preimage-resistant but not preimage-resistant). Similarly, if one can compute second-preimages then one can find collisions. Hence, in practice we prefer to study hash families that offer collision-resistance. For more details about these relations see Section 4.6.2 of [334], Section 4.2 of [592] or Section 10.3 of [572].

Another requirement of hash families is that they be **entropy smoothing**: If G is a "sufficiently large" finite set (i.e., $\#G \gg 2^l$) with a "sufficiently nice" distribution on it (but not necessarily uniform) then the distribution on $\{0,1\}^l$ given by $\Pr(y) = \sum_{x \in G : H(x) = y} \Pr(x)$ is "close" to uniform. We do not make this notion precise, but refer to Section 6.9 of Shoup [556].

In Section 23.2 we will need the following security notion for hash families (which is just a re-statement of second-preimage resistance).

Definition 3.1.2. Let X and Y be finite sets. A hash family $\{H_k : X \to Y : k \in \mathbb{K}\}$ is called **target-collision-resistant** if there is no polynomial-time adversary A with non-negligible advantage in the following game: A receives $x \in X$ and a key $k \in \mathbb{K}$, both chosen uniformly at random, then outputs an $x' \in X$ such that $x' \neq x$ and $H_k(x') = H_k(x)$.

For more details about target-collision-resistant hash families we refer to Section 5 of Cramer and Shoup [161].

3.2 Birthday Attack

Computing pre-images for a general hash function with l-bit output is expected to take approximately 2^l computations of the hash algorithm, but one can find collisions much more efficiently. Indeed, one can find collisions in roughly $\sqrt{\pi 2^{l-1}}$ applications of the hash function using a randomised algorithm as follows: Choose a subset $\mathcal{D} \subset \{0,1\}^l$ of distinguished points (e.g., those whose κ least significant bits are all zero, for some $0 < \kappa < l/4$). Choose random starting values $x_0 \in \{0,1\}^l$ (Joux [317] suggests that these should be distinguished points) and compute the sequence $x_n = H(x_{n-1})$ for $n = 1, 2, \ldots$ until $x_n \in \mathcal{D}$. Store (x_0, x_n) (i.e., the starting point and the ending distinguished point) and repeat. When the same distinguished point x is found twice then, assuming the starting points x_0 and x'_0 are distinct, one can find a collision in the hash function by computing the full sequences x_i and x'_j and determining the smallest integers i and j such that $x_i = x'_j$ and hence the collision is $H(x_{i-1}) = H(x'_{j-1})$.

If we assume that values x_i are close to uniformly distributed in $\{0,1\}^l$ then, by the birthday paradox, one expects to have a collision after $\sqrt{\pi 2^l / 2}$ strings have been

encountered (i.e., that many evaluations of the hash function). The storage required is expected to be

$$\sqrt{\pi 2^{l-1}}\frac{\#\mathcal{D}}{2^l}$$

pairs (x_0, x_n). For the choice of $\mathcal{D}$ as above, this would be about $2^{l/2-\kappa}$ bitstrings of storage. For many more details on this topic see Section 7.5 of Joux [317], Section 9.7.1 of Menezes, van Oorschot and Vanstone [418] or Section 3.2 of Vaudenay [616].

This approach also works if one wants to find collisions under some constraint on the messages (for example, all messages have a fixed prefix or suffix).

3.3 Message Authentication Codes

Message authentication codes are a form of symmetric cryptography. The main purpose is for a sender and receiver who share a secret key k to determine whether a communication between them has been tampered with.

A **message authentication code** (**MAC**) is a set of functions $\{\mathsf{MAC}_k(x) : k \in \mathbb{K}\}$ such that $\mathsf{MAC}_k : \{0,1\}^* \to \{0,1\}^l$. Note that this is exactly the same definition as a hash family. The difference between MACs and hash families lies in the security requirement; in particular the security model for MACs assumes the adversary does not know the key k. Informally, a MAC is secure against forgery if there is no efficient adversary that, given pairs $(x_i, y_i) \in \{0,1\}^* \times \{0,1\}^l$ such that $y_i = \mathsf{MAC}_k(x_i)$ (for some fixed, but secret, key k) for $1 \leq i \leq n$, can output a pair $(x, y) \in \{0,1\}^* \times \{0,1\}^l$ such that $y = \mathsf{MAC}_k(x)$ but $(x, y) \neq (x_i, y_i)$ for all $1 \leq i \leq n$. For precise definitions and further details of MACs see Section 4.3 of Katz and Lindell [334], Section 9.5 of Menezes, van Oorschot and Vanstone [418], Section 6.7.2 of Shoup [556], Section 4.4 of Stinson [592] or Section 3.4 of Vaudenay [616].

There are well-known constructions of MACs from hash functions (such as HMAC, see Section 4.7 of [334], Section 4.4.1 of [592] or Section 3.4.6 of [616]) and from block ciphers (such as CBC-MAC, see Section 4.5 of [334], Section 4.4.2 of [592] or Section 3.4.4 of [616]).

3.4 Constructions of Hash Functions

There is a large literature on constructions of hash functions and it is beyond the scope of this book to give the details. The basic process is to first define a **compression function** (namely a function that takes bitstrings of a fixed length to bitstrings of some shorter fixed length) and then to build a hash function on arbitrary length bitstrings by iterating the compression function (e.g., using the Merkle-Damgård construction). We refer to Chapter 4 of Katz and Lindell [334], Sections 9.3 and 9.4 of Menezes, van Oorschot and Vanstone [418], Chapter 10 of Smart [572], Chapter 4 of Stinson [592] or Chapter 3 of Vaudenay [616] for the details.

3.5 Number-Theoretic Hash Functions

We briefly mention some compression functions and hash functions that are based on algebraic groups and number theory. These schemes are not usually used in practice as the computational overhead is usually much too high.

An early proposal for hashing based on number theory, due to Davies and Price, was to use the function $H(x) = x^2 \pmod N$ where N is an RSA modulus whose factorisation is not known. Inverting such a function or finding collisions (apart from the trivial collisions $H(x) = H(\pm x + yN)$ for $y \in \mathbb{Z}$) is as hard as factoring N. There are a number of papers that build on this idea.

Another approach to hash functions based on factoring is to let N be an RSA modulus whose factorisation is unknown and let $g \in (\mathbb{Z}/N\mathbb{Z})^*$ be fixed. One can define the compression function $H : \mathbb{N} \to (\mathbb{Z}/N\mathbb{Z})^*$ by

$$H(x) = g^x \pmod N.$$

Finding a collision $H(x) = H(x')$ is equivalent to finding a multiple of the order of g. This is hard if factoring is hard, by Exercise 24.1.20. Finding pre-images is the discrete logarithm problem modulo N, which is also as hard as factoring. Hence, we have a collision-resistant compression function. More generally, fix $g, h \in (\mathbb{Z}/N\mathbb{Z})^*$ and consider the compression function $H : \mathbb{N} \times \mathbb{N} \to (\mathbb{Z}/N\mathbb{Z})^*$ defined by $H(x, y) = g^x h^y \pmod N$. A collision leads to either finding the order of g or h, or essentially finding the discrete logarithm of h with respect to g, and all these problems are as hard as factoring.

One can also base hash functions on the discrete logarithm problem in any group G. Let $g, h \in G$ have order r. One can now consider the compression function $H : \{0, \ldots, r-1\}^2 \to G$ by $H(x, y) = g^x h^y \pmod p$. It is necessary to fix the domain of the function since $H(x, y) = H(x + r, y) = H(x, y + r)$. If one can find collisions in this function then one can compute the discrete logarithm of h to the base g. A reference for this scheme is Chaum, van Heijst and Pfitzmann [130].

3.6 Full Domain Hash

Hash functions usually output binary strings of some fixed length l. Some cryptosystems, such as full domain hash RSA signatures, require hashing uniformly (or, at least, very close to uniformly) to $\mathbb{Z}/N\mathbb{Z}$, where N is large.

Bellare and Rogaway gave two methods to do this (one in Section 6 of [38] and another in Appendix A of [41]). We briefly recall the latter. The idea is to take some hash function H with fixed length output and define a new function $h(x)$ using a constant bitstring c and a counter i as

$$h(x) = H(c\|0\|x) \,\|\, H(c\|1\|x) \,\|\, \cdots \,\|\, H(c\|i\|x).$$

For the RSA application one can construct a bitstring that is a small amount larger than N and then reduce the resulting integer modulo N (as in Example 11.4.2).

These approaches have been critically analysed by Leurent and Nguyen [385]. They give a number of results that demonstrate that care is needed in assessing the security level of a hash function with "full domain" output.

3.7 Random Oracle Model

The random oracle model is a tool for the security analysis of cryptographic systems. It is a computational model that includes the **standard model** (i.e., the computational model mentioned in Section 2.1) together with an oracle that computes a "random" function from the set $\{0,1\}^*$ (i.e., binary strings of arbitrary finite length) to $\{0,1\}^\infty$ (i.e., bitstrings of countably infinite length). Since the number of such functions is uncountable, care must

be taken when defining the word "random". In any given application, one has a fixed bit-length l in mind for the output, and one also can bound the length of the inputs. Hence, one is considering functions $H : \{0,1\}^n \to \{0,1\}^l$ and, since there are $l2^n$ such functions we can define "random" to mean uniformly chosen from the set of all possible functions. We stress that a random oracle is a function: if it is queried twice on the same input then the output is the same.

A **cryptosystem in the random oracle model** is a cryptosystem where one or more hash functions are replaced by oracle queries to the random function. A cryptosystem is **secure in the random oracle model** if the cryptosystem in the random oracle model is secure. This does not imply that the cryptosystem in the standard model is secure, since there may be an attack that exploits some feature of the hash function. Indeed, there are "artificial" cryptosystems that are proven secure in the random oracle model, but that are insecure for any instantiation of the hash function (see Canetti, Goldreich and Halevi [115]).

The random oracle model enables security proofs in several ways. We list three of these ways, in increasing order of power.

1. It ensures that the output of H is truly random (rather than merely pseudorandom).

2. It allows the security proof to "look inside" the working of the adversary by learning the values that are inputs to the hash function.

3. It allows the security proof to "programme" the hash function so that it outputs a specific value at a crucial stage in the security game.

A classic example of a proof in the random oracle model is Theorem 20.4.11. An extensive discussion of the random oracle model is given in Section 13.1 of Katz and Lindell [334].

Since a general function from $\{0,1\}^n$ to $\{0,1\}^l$ cannot be represented more compactly than by a table of values, a random oracle requires $l2^n$ bits to describe. It follows that a random oracle cannot be implemented in polynomial space. However, the crucial observation that is used in security proofs is that a random oracle can be simulated in polynomial-time and space (assuming only polynomially many queries to the oracle are made) by creating, on-the-fly, a table giving the pairs (x, y) such that $H(x) = y$.