1. For a OWHF, $n \geq 80$ is required. Exhaustive off-line attacks require at most 2^n operations; this may be reduced with precomputation (Remark 9.35).
2. For a CRHF, $n \geq 160$ is required. Birthday attacks are applicable (Fact 9.33).
3. For a MAC, $n \geq 64$ along with a MAC key of 64-80 bits is sufficient for most applications and environments (cf. Table 9.1). If a single MAC key remains in use, off-line attacks may be possible given one or more text-MAC pairs; but for a proper MAC algorithm, preimage and 2nd-preimage resistance (as well as collision resistance) should follow directly from lack of knowledge of the key, and thus security with respect to such attacks should depend on the keysize rather than n. For attacks requiring on-line queries, additional controls may be used to limit the number of such queries, constrain the format of MAC inputs, or prevent disclosure of MAC outputs for random (chosen-text) inputs. Given special controls, values as small as $n = 32$ or 40 may be acceptable; but caution is advised, since even with one-time MAC keys, the chance any randomly guessed MAC being correct is 2^{-n}, and the relevant factors are the total number of trials a system is subject to over its lifetime, and the consequences of a single successful forgery.

These guidelines may be relaxed somewhat if a lower threshold of computational infeasibility is assumed (e.g., 2^{64} instead of 2^{80}). However, an additional consideration to be taken into account is that for both a CRHF and a OWHF, not only can off-line attacks be carried out, but these can typically be parallelized. Key search attacks against MACs may also be parallelized.

9.4 Unkeyed hash functions (MDCs)

A move from general properties and constructions to specific hash functions is now made, and in this section the subclass of unkeyed hash functions known as modification detection codes (MDCs) is considered. From a structural viewpoint, these may be categorized based on the nature of the operations comprising their internal compression functions. From this viewpoint, the three broadest categories of iterated hash functions studied to date are hash functions *based on block ciphers*, *customized hash functions*, and hash functions *based on modular arithmetic*. Customized hash functions are those designed specifically for hashing, with speed in mind and independent of other system subcomponents (e.g., block cipher or modular multiplication subcomponents which may already be present for non-hashing purposes).

Table 9.3 summarizes the conjectured security of a subset of the MDCs subsequently discussed in this section. Similar to the case of block ciphers for encryption (e.g. 8- or 12-round DES vs. 16-round DES), security of MDCs often comes at the expense of speed, and tradeoffs are typically made. In the particular case of block-cipher-based MDCs, a provably secure scheme of Merkle (see page 378) with rate 0.276 (see Definition 9.40) is known but little-used, while MDC-2 is widely believed to be (but not provably) secure, has rate $= 0.5$, and receives much greater attention in practice.

9.4.1 Hash functions based on block ciphers

A practical motivation for constructing hash functions from block ciphers is that if an efficient implementation of a block cipher is already available within a system (either in hardware or software), then using it as the central component for a hash function may provide

↓Hash function	n	m	Preimage	Collision	Comments
Matyas-Meyer-Oseas[a]	n	n	2^n	$2^{n/2}$	for keylength $= n$
MDC-2 (with DES)[b]	64	128	$2 \cdot 2^{82}$	$2 \cdot 2^{54}$	rate 0.5
MDC-4 (with DES)	64	128	2^{109}	$4 \cdot 2^{54}$	rate 0.25
Merkle (with DES)	106	128	2^{112}	2^{56}	rate 0.276
MD4	512	128	2^{128}	2^{20}	Remark 9.50
MD5	512	128	2^{128}	2^{64}	Remark 9.52
RIPEMD-128	512	128	2^{128}	2^{64}	–
SHA-1, RIPEMD-160	512	160	2^{160}	2^{80}	–

[a] The same strength is conjectured for Davies-Meyer and Miyaguchi-Preneel hash functions.
[b] Strength could be increased using a cipher with keylength equal to cipher blocklength.

Table 9.3: *Upper bounds on strength of selected hash functions. n-bit message blocks are processed to produce m-bit hash-values. Number of cipher or compression function operations currently believed necessary to find preimages and collisions are specified, assuming no underlying weaknesses for block ciphers (figures for MDC-2 and MDC-4 account for DES complementation and weak key properties). Regarding rate, see Definition 9.40.*

the latter functionality at little additional cost. The (not always well-founded) hope is that a good block cipher may serve as a building block for the creation of a hash function with properties suitable for various applications.

Constructions for hash functions have been given which are "provably secure" assuming certain ideal properties of the underlying block cipher. However, block ciphers do not possess the properties of random functions (for example, they are invertible – see Remark 9.14). Moreover, in practice block ciphers typically exhibit additional regularities or weaknesses (see §9.7.4). For example, for a block cipher E, double encryption using an encrypt-decrypt (E-D) cascade with keys K_1, K_2 results in the identity mapping when $K_1 = K_2$. In summary, while various necessary conditions are known, it is unclear exactly what requirements of a block cipher are sufficient to construct a secure hash function, and properties adequate for a block cipher (e.g., resistance to chosen-text attack) may not guarantee a good hash function.

In the constructions which follow, Definition 9.39 is used.

9.39 Definition An *(n,r) block cipher* is a block cipher defining an invertible function from n-bit plaintexts to n-bit ciphertexts using an r-bit key. If E is such a cipher, then $E_k(x)$ denotes the encryption of x under key k.

Discussion of hash functions constructed from n-bit block ciphers is divided between those producing *single-length* (n-bit) and *double-length* ($2n$-bit) hash-values, where single and double are relative to the size of the block cipher output. Under the assumption that computations of 2^{64} operations are infeasible,[3] the objective of single-length hash functions is to provide a OWHF for ciphers of blocklength near $n = 64$, or to provide CRHFs for cipher blocklengths near $n = 128$. The motivation for double-length hash functions is that many n-bit block ciphers exist of size approximately $n = 64$, and single-length hash-codes of this size are not collision resistant. For such ciphers, the goal is to obtain hash-codes of bitlength $2n$ which are CRHFs.

In the simplest case, the size of the key used in such hash functions is approximately the same as the blocklength of the cipher (i.e., n bits). In other cases, hash functions use

[3] The discussion here is easily altered for a more conservative bound, e.g., 2^{80} operations as used in §9.3.5. Here 2^{64} is more convenient for discussion, due to the omnipresence of 64-bit block ciphers.

larger (e.g., double-length) keys. Another characteristic to be noted in such hash functions is the number of block cipher operations required to produce a hash output of blocklength equal to that of the cipher, motivating the following definition.

9.40 Definition Let h be an iterated hash function constructed from a block cipher, with compression function f which performs s block encryptions to process each successive n-bit message block. Then the *rate* of h is $1/s$.

The hash functions discussed in this section are summarized in Table 9.4. The Matyas-Meyer-Oseas and MDC-2 algorithms are the basis, respectively, of the two generic hash functions in ISO standard 10118-2, each allowing use of any n-bit block cipher E and providing hash-codes of bitlength $m \leq n$ and $m \leq 2n$, respectively.

Hash function	(n, k, m)	Rate
Matyas-Meyer-Oseas	(n, k, n)	1
Davies-Meyer	(n, k, n)	k/n
Miyaguchi-Preneel	(n, k, n)	1
MDC-2 (with DES)	$(64, 56, 128)$	$1/2$
MDC-4 (with DES)	$(64, 56, 128)$	$1/4$

Table 9.4: *Summary of selected hash functions based on n-bit block ciphers. k = key bitsize (approximate); function yields m-bit hash-values.*

(i) Single-length MDCs of rate 1

The first three schemes described below, and illustrated in Figure 9.3, are closely related single-length hash functions based on block ciphers. These make use of the following predefined components:

1. a generic n-bit block cipher E_K parametrized by a symmetric key K;
2. a function g which maps n-bit inputs to keys K suitable for E (if keys for E are also of length n, g might be the identity function); and
3. a fixed (usually n-bit) initial value IV, suitable for use with E.

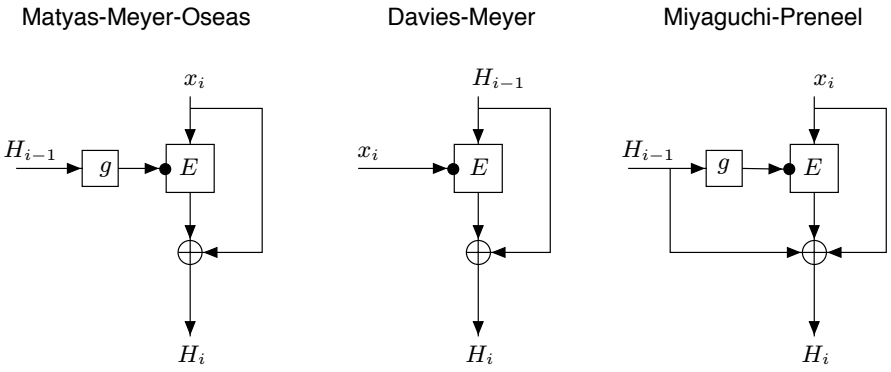

Figure 9.3: *Three single-length, rate-one MDCs based on block ciphers.*

9.41 Algorithm Matyas-Meyer-Oseas hash

INPUT: bitstring x.
OUTPUT: n-bit hash-code of x.

1. Input x is divided into n-bit blocks and padded, if necessary, to complete last block. Denote the padded message consisting of t n-bit blocks: $x_1 x_2 \ldots x_t$. A constant n-bit initial value IV must be pre-specified.
2. The output is H_t defined by: $H_0 = IV$; $H_i = E_{g(H_{i-1})}(x_i) \oplus x_i,\ 1 \leq i \leq t$.

9.42 Algorithm Davies-Meyer hash

INPUT: bitstring x.
OUTPUT: n-bit hash-code of x.

1. Input x is divided into k-bit blocks where k is the keysize, and padded, if necessary, to complete last block. Denote the padded message consisting of t k-bit blocks: $x_1 x_2 \ldots x_t$. A constant n-bit initial value IV must be pre-specified.
2. The output is H_t defined by: $H_0 = IV$; $H_i = E_{x_i}(H_{i-1}) \oplus H_{i-1},\ 1 \leq i \leq t$.

9.43 Algorithm Miyaguchi-Preneel hash

This scheme is identical to that of Algorithm 9.41, except the output H_{i-1} from the previous stage is also XORed to that of the current stage. More precisely, H_i is redefined as: $H_0 = IV$; $H_i = E_{g(H_{i-1})}(x_i) \oplus x_i \oplus H_{i-1},\ 1 \leq i \leq t$.

9.44 Remark (*dual schemes*) The Davies-Meyer hash may be viewed as the 'dual' of the Matyas-Meyer-Oseas hash, in the sense that x_i and H_{i-1} play reversed roles. When DES is used as the block cipher in Davies-Meyer, the input is processed in 56-bit blocks (yielding rate $56/64 < 1$), whereas Matyas-Meyer-Oseas and Miyaguchi-Preneel process 64-bit blocks.

9.45 Remark (*black-box security*) Aside from heuristic arguments as given in Example 9.13, it appears that all three of Algorithms 9.41, 9.42, and 9.43 yield hash functions which are provably secure under an appropriate "black-box" model (e.g., assuming E has the required randomness properties, and that attacks may not make use of any special properties or internal details of E). "Secure" here means that finding preimages and collisions (in fact, pseudo-preimages and pseudo-collisions – see §9.7.2) require on the order of 2^n and $2^{n/2}$ n-bit block cipher operations, respectively. Due to their single-length nature, none of these three is collision resistant for underlying ciphers of relatively small blocklength (e.g., DES, which yields 64-bit hash-codes).

Several double-length hash functions based on block ciphers are considered next.

(ii) Double-length MDCs: MDC-2 and MDC-4

MDC-2 and MDC-4 are manipulation detection codes requiring 2 and 4, respectively, block cipher operations per block of hash input. They employ a combination of either 2 or 4 iterations of the Matyas-Meyer-Oseas (single-length) scheme to produce a double-length hash. When used as originally specified, using DES as the underlying block cipher, they produce 128-bit hash-codes. The general construction, however, can be used with other block ciphers. MDC-2 and MDC-4 make use of the following pre-specified components:

1. DES as the block cipher E_K of bitlength $n = 64$ parameterized by a 56-bit key K;
2. two functions g and $\tilde{g}$ which map 64-bit values U to suitable 56-bit DES keys as follows. For $U = u_1 u_2 \ldots u_{64}$, delete every eighth bit starting with u_8, and set the 2nd and 3rd bits to '10' for g, and '01' for $\tilde{g}$:

$$g(U) = u_1\ 1\ 0\ u_4 u_5 u_6 u_7 u_9 u_{10} \ldots u_{63}.$$
$$\tilde{g}(U) = u_1\ 0\ 1\ u_4 u_5 u_6 u_7 u_9 u_{10} \ldots u_{63}.$$

(The resulting values are guaranteed not to be weak or semi-weak DES keys, as all such keys have bit 2 = bit 3; see page 375. Also, this guarantees the security requirement that $g(IV) \neq \tilde{g}(\widetilde{IV})$.)

MDC-2 is specified in Algorithm 9.46 and illustrated in Figure 9.4.

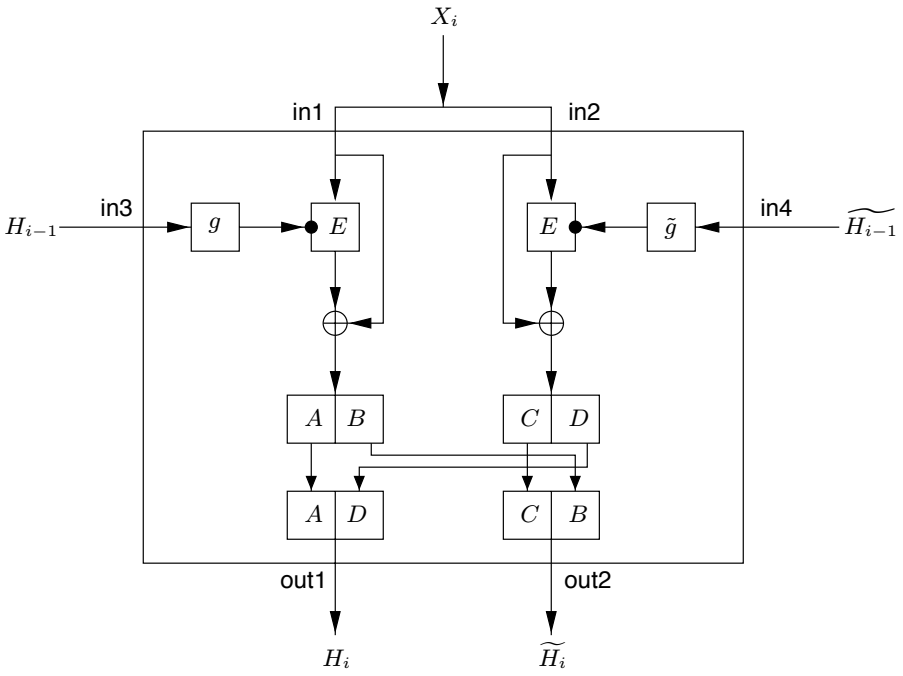

Figure 9.4: *Compression function of MDC-2 hash function. $E = DES$.*

9.46 Algorithm MDC-2 hash function (DES-based)

INPUT: string x of bitlength $r = 64t$ for $t \geq 2$.
OUTPUT: 128-bit hash-code of x.

1. Partition x into 64-bit blocks x_i: $x = x_1 x_2 \ldots x_t$.
2. Choose the 64-bit non-secret constants $IV, \widetilde{IV}$ (the same constants must be used for MDC verification) from a set of recommended prescribed values. A default set of prescribed values is (in hexadecimal): $IV =$ `0x5252525252525252`, $\widetilde{IV} =$ `0x2525252525252525`.

3. Let $||$ denote concatenation, and C_i^L, C_i^R the left and right 32-bit halves of C_i. The output is $h(x) = H_t \,||\, \widetilde{H_t}$ defined as follows (for $1 \le i \le t$):

$$H_0 = IV; \quad k_i = g(H_{i-1}); \quad C_i = E_{k_i}(x_i) \oplus x_i; \quad H_i = C_i^L \,||\, \widetilde{C_i}^R$$
$$\widetilde{H_0} = \widetilde{IV}; \quad \widetilde{k_i} = \widetilde{g}(\widetilde{H_{i-1}}); \quad \widetilde{C_i} = E_{\widetilde{k_i}}(x_i) \oplus x_i; \quad \widetilde{H_i} = \widetilde{C_i}^L \,||\, {C_i}^R .$$

In Algorithm 9.46, padding may be necessary to meet the bitlength constraint on the input x. In this case, an unambiguous padding method may be used (see Remark 9.31), possibly including MD-strengthening (see Remark 9.32).

MDC-4 (see Algorithm 9.47 and Figure 9.5) is constructed using the MDC-2 compression function. One iteration of the MDC-4 compression function consists of two sequential executions of the MDC-2 compression function, where:

1. the two 64-bit data inputs to the first MDC-2 compression are both the same next 64-bit message block;
2. the keys for the first MDC-2 compression are derived from the outputs (chaining variables) of the previous MDC-4 compression;
3. the keys for the second MDC-2 compression are derived from the outputs (chaining variables) of the first MDC-2 compression; and
4. the two 64-bit data inputs for the second MDC-2 compression are the outputs (chaining variables) from the opposite sides of the previous MDC-4 compression.

9.47 Algorithm MDC-4 hash function (DES-based)

INPUT: string x of bitlength $r = 64t$ for $t \ge 2$. (See MDC-2 above regarding padding.)
OUTPUT: 128-bit hash-code of x.

1. As in step 1 of MDC-2 above.
2. As in step 2 of MDC-2 above.
3. With notation as in MDC-2, the output is $h(x) = G_t \,||\, \widetilde{G_t}$ defined as follows (for $1 \le i \le t$):

$$G_0 = IV; \quad \widetilde{G_0} = \widetilde{IV};$$
$$k_i = g(G_{i-1}); \quad C_i = E_{k_i}(x_i) \oplus x_i; \quad H_i = C_i^L \,||\, \widetilde{C_i}^R$$
$$\widetilde{k_i} = \widetilde{g}(\widetilde{G_{i-1}}); \quad \widetilde{C_i} = E_{\widetilde{k_i}}(x_i) \oplus x_i; \quad \widetilde{H_i} = \widetilde{C_i}^L \,||\, {C_i}^R$$
$$j_i = g(H_i); \quad D_i = E_{j_i}(\widetilde{G_{i-1}}) \oplus \widetilde{G_{i-1}}; \quad G_i = D_i^L \,||\, \widetilde{D_i}^R$$
$$\widetilde{j_i} = \widetilde{g}(\widetilde{H_i}); \quad \widetilde{D_i} = E_{\widetilde{j_i}}(G_{i-1}) \oplus G_{i-1}; \quad \widetilde{G_i} = \widetilde{D_i}^L \,||\, {D_i}^R .$$

9.4.2 Customized hash functions based on MD4

Customized hash functions are those which are specifically designed "from scratch" for the explicit purpose of hashing, with optimized performance in mind, and without being constrained to reusing existing system components such as block ciphers or modular arithmetic. Those having received the greatest attention in practice are based on the MD4 hash function.

Number 4 in a series of hash functions (*Message Digest* algorithms), MD4 was designed specifically for software implementation on 32-bit machines. Security concerns motivated the design of MD5 shortly thereafter, as a more conservative variation of MD4.

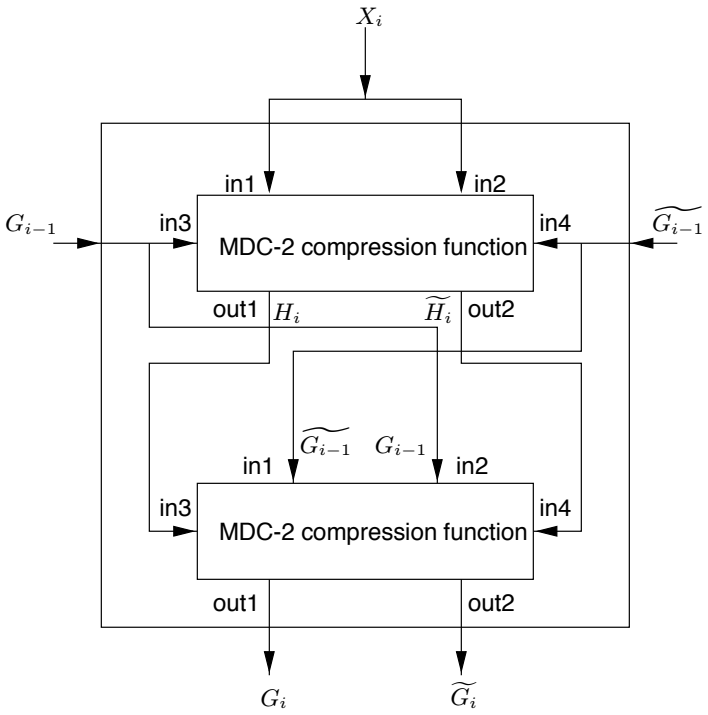

Figure 9.5: *Compression function of MDC-4 hash function*

Other important subsequent variants include the Secure Hash Algorithm (SHA-1), the hash function RIPEMD, and its strengthened variants RIPEMD-128 and RIPEMD-160. Parameters for these hash functions are summarized in Table 9.5. "Rounds $\times$ Steps per round" refers to operations performed on input blocks within the corresponding compression function. Table 9.6 specifies test vectors for a subset of these hash functions.

Notation for description of MD4-family algorithms

Table 9.7 defines the notation for the description of MD4-family algorithms described below. Note 9.48 addresses the implementation issue of converting strings of bytes to words in an unambiguous manner.

9.48 Note (*little-endian vs. big-endian*) For interoperable implementations involving byte-to-word conversions on different processors (e.g., converting between 32-bit words and groups of four 8-bit bytes), an unambiguous convention must be specified. Consider a stream of bytes B_i with increasing memory addresses i, to be interpreted as a 32-bit word with numerical value W. In *little-endian* architectures, the byte with the lowest memory address (B_1) is the least significant byte: $W = 2^{24}B_4 + 2^{16}B_3 + 2^8B_2 + B_1$. In *big-endian* architectures, the byte with the lowest address (B_1) is the most significant byte: $W = 2^{24}B_1 + 2^{16}B_2 + 2^8B_3 + B_4$.

(i) MD4

MD4 (Algorithm 9.49) is a 128-bit hash function. The original MD4 design goals were that breaking it should require roughly brute-force effort: finding distinct messages with the same hash-value should take about 2^{64} operations, and finding a message yielding a