# BIBLIOGRAPHY

[1] J. L. Massey, "Cryptography: Fundamentals and aplications," 1995. Copies of Transparencies, Advanced Technology Seminars.

[2] W. T. Penzhorn, "Hash functions and authentication," Tech. Rep. WP2, Ciphertec cc, 24 January 1995.

[3] B. Preneel, *Analysis and Design of Cryptographic Hash Functions*. PhD thesis, Katholieke Universiteit Leuven, 1993.

[4] J. L. Carter and M. N. Wegman, "Universal classes of hash functions," *Journal of Computer and Systems Sciences*, vol. 18, pp. 143–154, 1979.

[5] *Secure Electronic Transaction (SET) Specification Book 3: Formal Protocol Definition*, 24 June, Revised August 1 1996.

[6] D. W. Davies and W. L. Price, "The application of digital signatures based on public-key cryptosystems," in *Proc. Fifth Intl. Computer Communications Conference*, pp. 525–530, October 1980.

[7] R. Anderson and E. Biham, "Two practical and provably secure block ciphers: BEAR and LION," in *Fast Software Encryption, Third International Workshop* (D. Gollman, ed.), Lecture Notes in Computer Science No. 1039, Springer-Verlag, pp. 113–120, 1996.

[8] P. Morin, "Provably secure and efficient block ciphers," *Third Annual Workshop on Selected Areas in Cryptography*, pp. 30–37, 1996.

[9] C. H. Lim, "Message encryption and authentication using one-way hash functions," *Third Annual Workshop on Selected Areas in Cryptography*, pp. 38–48, 1996.

[10] R. L. Rivest, "The MD4 message digest algorithm," in *Advances in Cryptology - CRYPTO ' 90*, Lecture Notes in Computer Science vol 537, Springer-Verlag, pp. 303 – 311, 1991.

[11] B. den Boer and A. Bosselaers, "An attack on the last two rounds of MD4," in *Advances in Cryptology - CRYPTO ' 91*, Lecture Notes in Computer Science No. 576, Springer-Verlag, pp. 194–203, 1992.

[12] H. Dobbertin, "Cryptanalysis of MD5 compress," *Rump Session EUROCRYPT ' 96*, 1996.

[13] National Institute of Standards and Technology (NIST), *FIPS Publication 180-1: Secure Hash Standard (SHS)*, April 17, 1995.

[14] H. Dobbertin, "Cryptanalysis of MD4," in *Fast Software Encryption, Third International Workshop* (D. Gollman, ed.), Lecture Notes in Computer Science No. 1039, Springer-Verlag, pp. 53–69, 1996.

[15] H. Dobbertin, A. Bosselaers, and B. Preneel, "RIPEMD-160: a strengthened version of ripemd," in *Fast Software Encryption, Third International Workshop* (D. Gollman, ed.), Lecture Notes in Computer Science No. 1039, Springer-Verlag, pp. 71–82, 1996.

[16] R. C. Merkle, "One way hash functions and DES," *Crypto '89*, pp. 428–446, 1989.

[17] P. R. Kasselman, "Analysis of dedicated hash functions," Tech. Rep. Ciph-96-10, Ciphertec cc, November 1996.

[18] G. J. Simmons, "A survey of information authentication," in *Contemporary Cryptology, The Science of Information Integrity* (G. J. Simmons, ed.), pp. 379–419, New York: IEEE Press, 1991.

[19] D. G. Abraham, G. M. Dolan, G. P. Double and J. V. Stevens, "Transaction security system," *IBM Systems Journal*, vol. 30, no. 2, pp. 206–209, 1991.

[20] J.-J. Quisquater and J.-P. Delescaille, "How easy is collision search? New results and applications to DES," in *Advances in Cryptology - CRYPTO ' 89* (G. Brassard, ed.), Lecture Notes in Computer Science No. 435, Springer-Verlag, pp. 408–415, 1990.

[21] J.-J. Quisquater and J.-P. Delescaille, "How easy is collision search? Applications to DES," in *Advances in Cryptology - EUROCRYPT ' 89* (J. Quisquater and J. Vandewalle,

eds.), Lecture Notes in Computer Science No. 434, Springer-Verlag, pp. 428–433, 1990.

[22] I. Damgård, "A design principle for hash functions," in *Advances in Cryptology - CRYPTO ' 89* (G. Brassard, ed.), Lecture Notes in Computer Science No. 435, Springer-Verlag, pp. 416–427, 1990.

[23] R. C. Merkle, "One way hash functions and DES," in *Advances in Cryptology - CRYPTO ' 89* (G. Brassard, ed.), Lecture Notes in Computer Science No. 435, Springer-Verlag, pp. 428–446, 1990.

[24] D. Coppersmith, "Another birthday attack," in *Advances in Cryptology - CRYPTO ' 85* (H. C. Williams, ed.), Lecture Notes in Computer Science No. 218, Springer-Verlag, pp. 14–17, 1986.

[25] M. Girault, R. Cohen, and M. Campana, "A generalised birthday attack," in *Advances in Cryptology - EUROCRYPT ' 88* (C. G. Günther, ed.), Lecture Notes in Computer Science No. 330, Springer-Verlag, pp. 129–156, 1988.

[26] L. Knudsen, "Cryptanalysis of LOKI," in *Cryptography and Coding III*, vol. 45, pp. 223–236, The Institute of Mathematics and its Applications Conference Series, Clarendon Press, Oxford, 1993.

[27] M. J. B. Robshaw, "Block ciphers," Tech. Rep. TR 601, RSA Laborities, 2 August 1995.

[28] B. Kaliski and M. Robshaw, "Message authentication with MD5," *CryptoBytes*, vol. 1, pp. 5–8, Spring 1995.

[29] B. Preneel and P. C. van Oorschot, "MDx-MAC and building fast MACs from hash functions," in *Advances in Cryptology - CRYPTO ' 94* (D. Coppersmith, ed.), Lecture Notes in Computer Science No. 963, Springer-Verlag, pp. 1–14, 1995.

[30] D. W. Davies, "A message authenticator algorithm," in *Advances in Cryptology - CRYPTO ' 84* (G. R. Blakley and D. C. Chaum, eds.), Lecture Notes in Computer Science No. 196, Springer-Verlag, pp. 393–400, 1985.

[31] B. Preneel and P. C. van Oorschot, "On the security of two MAC algorithms," in *Advances in Cryptology - EUROCRYPT ' 96* (U. Maurer, ed.), Lecture Notes in Computer Science No. 1070, Springer-Verlag, pp. 19–32, 1996.

[32] R. S. Winternitz, "Producing a one-way hash function from DES," in *Advances in Cryptology - CRYPTO ' 83* (D. Chaum, ed.), (New York), pp. 203–207, Plenum Press, 1984.

[33] M. E. Hellman, R. Merkle, R. Schroeppel, L. Washington, W. Diffie, S. Pohlig, and P. Schweitzer, "Results of an initial attempt to cryptanalyze the NBS Data Encryption Standard," Tech. Rep. SEL 76–042, Stanford University, 1976.

[34] M. Kwan and J. Pieprzyk, "A general purpose technique for locating key scheduling weakness in DES-like cryptosystems," in *Advances in Cryptology–ASIACRYPT '91* (H. Imai, R. Rivest, and T. Matsumoto, eds.), Lecture Notes in Computer Science No. 739, Springer-Verlag, pp. 237–246, 1993.

[35] L. Brown, M. Kwan, J. Pieprzyk, and J. Seberry, "Improving resistance to differential cryptanalysis and the redesign of LOKI," in *Advances in Cryptology–ASIACRYPT '91* (H. Imai, R. Rivest, and T. Matsumoto, eds.), Lecture Notes in Computer Science No. 739, Springer-Verlag, pp. 36–50, 1993.

[36] J. Daemen, R. Govaerts, and J. Vandewalle, "Weak keys for IDEA," in *Advances in Cryptology - CRYPTO ' 93* (D. R. Stinson, ed.), Lecture Notes in Computer Science No. 773, Springer-Verlag, pp. 224–231, 1994.

[37] D. Boneh, R. A. DeMillo, and R. J. Lipton, "On the importance of checking computations (Extended abstract)." Internet, 1996.

[38] A. Shamir and E. Biham, "Research anouncement: A new cryptanalytic attack on DES." Internet: http://jya.com/dfa.htm, 1996.

[39] J.-J. Quisquater, "Short cut for exhaustive key search using fault analysis: Applications to DES, MAC, Keyed hash function, Identification protocols, . . . ." Internet, 1996.

[40] D. Coppersmith, "Analysis of ISO/CCITT Document X.509 Annex D." Internal Memo, IBM T.J. Watson Center, June 11, 1989.

[41] T. Beth, F. Bausspieß, and F. Damm, "Workshop on cryptographic hash functions," Tech. Rep. 92/11, E.I.S.S., 1992.

[42] G. Brassard, "The impending demise of RSA?," *CryptoBytes*, vol. 1, pp. 1–4, Spring 1995.

[43] F. Bausspieß and F. Damm, "Requirements for cryptographic hash functions," Tech. Rep. 92/2, E.I.S.S., 1992.

[44] R. L. Rivest, "The MD4 message digest algorithm." Internet Request for Comments (RFC), 1990. RFC 1320.

[45] R. L. Rivest, "The MD5 message-digest algorithm." Internet Request for Comments (RFC), April 1992. RFC 1321.

[46] C. E. Shannon, "Communication theory of secrecy systems," in *Claude Elwood Shannon - Collected Papers* (N. J. A. Sloane and A. D. Wyner, eds.), pp. 84–143, IEEE Press, 1983.

[47] R. Anderson and E. Biham, "Tiger: A fast new hash function." Internet, 1996.

[48] J. Daemen, *Cipher and Hash Function Design.* PhD thesis, Katholieke Universiteit Leuven, 1993.

[49] X. Lai and J. L. Massey, "Hash functions based on block ciphers," in *Advances in Cryptology - EUROCRYPT ' 92*, Lecture Notes in Computer Science No. 658, Springer-Verlag, pp. 55–70, 1992.

[50] B. Schneier, *Applied Cryptography: Protocols, Algorithms, and Source Code in C.* New York: John Wiley & Sons, 1993.

[51] G. J. Kühn, "S-box design and analysis," Tech. Rep. Ciph-96-13, Ciphertec cc, Desember 1996.

[52] R. R. Jueneman, "A high speed manipulation detection code," in *Advances in Cryptology - CRYPTO ' 86* (A. Odlyzko, ed.), Lecture Notes in Computer Science No. 263, Springer-Verlag, pp. 327–346, 1987.

[53] S. Vaudenay, "On the need of multipermuations: Cryptanalysis of MD4 and SAFER," in *Proceedings of the 1994 Leuven Workshop on Cryptographic Algorithms*, Lecture Notes in Computer Science vol 1008, Springer-Verlag, pp. 286–297, 1995.

[54] H. Dobbertin, "RIPEMD with two round compress is not collision-free," *Journal of Cryptology*, vol. 10, no. 1, pp. 51–70, 1997.

[55] H. Dobbertin, "The status of MD5 after a recent attack," *CryptoBytes*, vol. 2, pp. 1–6, Summer 1996.

[56] P. R. Kasselman, "A fast attack on the MD4 hash function," in *Comsig 97* (M. Ings, ed.), no. 97TH8312 in IEEE Catalog, pp. 147–150, South African Section IEEE, 1997.

[57] P. R. Kasselman, "Analysis and design of hash functions: Part II," Tech. Rep. Ciph/97/12/1(II), Ciphertec cc, December 1997.

[58] P. R. Kasselman, "Analysis and design of hash functions: Part I," Tech. Rep. Ciph/97/12/1(I), Ciphertec cc, December 1997.

[59] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone, *Handbook of Applied Cryptography*. New York: CRC Press, 1997.

[60] A. Meijer, "Galois field counters and linear feedback shift registers," 1997. Class notes.

[61] W. W. Peterson and E. J. Weldon, *Error Correcting Codes*. Massachsetts: MIT Press, 2 ed., 1972.

[62] Y. Zheng, J. Pieprzyk, and J. Seberry, "HAVAL - a one-way hashing algorithm with variable length of output," in *Advances in Cryptology — Auscrypt '92* (J. Seberry and Y. Zheng, eds.), (Berlin), pp. 83–104, 1993.

[63] J. F. Wakerly, *Digital Design Principles and Practices*. London: Prentice-Hall International Editions, 1990.

[64] E. K. Grossman and B. Tuckerman, "Analysis of a feistel-like cipher weakened by having no rotating key," Tech. Rep. RC 6375, IBM T.J. Watson Research, Jan 1977.

[65] E. Biham, "New types of cryptanalytic attacks using related keys," in *Advances in Cryptology — EUROCRYPT '93* (T. Helleseth, ed.), Lecture Notes in Computer Science Vol 735, (Berlin), Springer-Verlag, pp. 398–409, 1994.

[66] C. M. Adams, "Simple and effective key scheduling for symmetric ciphers (extended abstract)," *Workshop on Selected Areas in Cryptography*, pp. 129–133, 1994.

[67] V. Rijmen, *Cryptanalysis and Design of Iterated Block Ciphers*. PhD thesis, Katholieke Universiteit Leuven, 1997.

[68] M. N. Wegman and J. L. Carter, "New hash functions and their use in authentication and set equality," *Journal of Computer and Systems Sciences*, vol. 22, pp. 265–279, 1981.

[69] A. R. Meijer, "Universal hash functions in authentication," Tech. Rep. KT437122, Ciphertec cc, 28 February 1996.

[70] S. M. Matyas, C. H. Meyer, and J. Oseas, "Generating strong one-way functions with cryptographic algorithms," *IBM Tech. Disclosure Bull.*, vol. 27, no. 10A, pp. 5658–5659, 1985.

[71] B. Preneel, R. Govaerts, and J. Vandewalle, "Hash functions based on block ciphers: a synthetic approach," in *Advances in Cryptology - CRYPTO ' 93* (D. R. Stinson, ed.), Lecture Notes in Computer Science No. 773, Springer-Verlag, pp. 368–378, 1994.

[72] X. Lai, R. Rueppel, and J. Woollven, "A fast cryptographic checksum algorithm based on stream ciphers," in *Advances in Cryptology — AUSCRYPT '92* (J. Seberry and Y. Zheng, eds.), Lecture Notes in Computer Science No. 718, (Berlin), Springer-Verlag, pp. 339–348, 1993.

[73] M. Bellare, R. Canetti, and H. Krawczyk, "Keying hash functions for message authentication," in *Advances in Cryptology - CRYPTO ' 96* (N. Koblitz, ed.), Lecture Notes in Computer Science No. 1109, Springer-Verlag, pp. 1–15, 1995.

[74] M. Bellare, R. Canetti, and H. Krawczyk, "The HMAC construction," *CryptoBytes*, vol. 2, pp. 12–15, Spring 1996.

[75] G. Tsudik, "Message authentication with one-way hash functions," *ACM SIGCOMM, Computer Communication Review*, vol. 22, pp. 29–38, Oct. 1992.

[76] J. M. Galvin, K. McCloghrie, and J. R. Davin, "Secure management of snmp networks," *Integrated Network Management II*, pp. 703–714, 1991.

[77] J. Linn, "The Kerberos version 5 GSS-API mechanism." RFC 1964, 1996.

[78] M. Bellare, R. Canetti, and H. Krawczyk, "Keying hash functions for message authentication." Internet: http://www.research.ibm.com.security/ or http://www-cse.ucsd.edu/users/mihir/papers/papers.html, 1996.

[79] M. Bellare, R. Guérin, and P. Rogaway, "XOR MACs: New methods for message authentication using finite pseudorandom functions," in *Advances in Cryptology - CRYPTO ' 95* (D. Coppersmith, ed.), Lecture Notes in Computer Science No. 963, Springer-Verlag, pp. 15–28, 1995.

[80] M. Bellare, R. Guérin, and P. Rogaway, "XOR MACs: New methods for message authentication using finite pseudorandom functions." Internet: http://www-cse.ucsd.edu/users/mihir/papers/papers.html, 1995.

[81] W. T. Penzhorn, "Study into international standards," Tech. Rep. Ciph-96-12, Ciphertec cc, 6 December 1996.

[82] M. Bellare, J. Kilian, and P. Rogaway, "The security of cipher block chaining," in *Advances in Cryptology - CRYPTO ' 94* (Y. G. Desmedt, ed.), Lecture Notes in Computer Science No. 839, Springer-Verlag, pp. 341–358, 1994.

# APPENDIX A: ADDITIONAL HASH FUNCTION CONSTRUCTIONS

## A.1 INTRODUCTION

This Appendix describes a number of additional hash function constructions. Specific attention is given to tree constructions, the cascading of hash functions and the use of block and stream ciphers to construct round functions that can be used as part of the Dåmgard-Merkle construction. A number of generic techniques that allows the construction of MACs based on MACs are also considered. A short review of current international standards is also included.

## A.2 TREE CONSTRUCTIONS

This scheme is specifically intended for high speed hashing. The first construction along these lines was presented by Carter and Wegman [68]. In [69] this scheme is referred to as concatenation hashing. It was re-discovered independently by Preneel [3] and Damgård [22]. This hash scheme can be generalised and requires the following:

1. A round function $f()$ that takes a fixed input of $m$ bits and reduces it to $n$ bits.
2. Padding rule.
3. A scalable number of parallel processors.

The difference between the construction by Preneel and the construction by Carter and Wegman is found in the round function used. Carter and Wegman propose the use of a universal hash function (a complexity theoretic construction). Preneel specifies that any secure round function, $f()$, could be used. It is advised that the round function chosen for this scheme should adhere to the conditions imposed on round functions used in the Damgård-Merkle scheme (see Section 5.3).

For the tree hashing scheme to work it is required that the message length, $r$ should be a multiple of the block length $m$. In addition it is required that the number of blocks in the original message should be a multiple of two. These requirements imply a form of padding. The same padding rules as described in Section 5.3 can be applied to this construction.

### A.2.1 Construction

A description of the construction of a tree structured hash function is presented below.

For a message of length $r = 2^q$ with $q \in \mathbb{Z}, q > 0$:

$$\begin{aligned} H_i^1 &= f(X_{2i-1}, X_{2i}) \qquad i \in \{1, 2, 3 \ldots 2^{q-1}\} \\ H_i^j &= f(H_{2i-1}^{j-1}, H_{2i}^{j-1}) \qquad i \in \{1, 2, 3 \ldots 2^{q-1}\} \\ & \qquad\qquad\qquad\qquad\quad j \in \{2, 3 \ldots r-1\} \\ h(X) &= f(H_1^{r-1}, H_2^{r-1}). \end{aligned}$$

A graphical representation of this scheme is presented in Figure A.1.

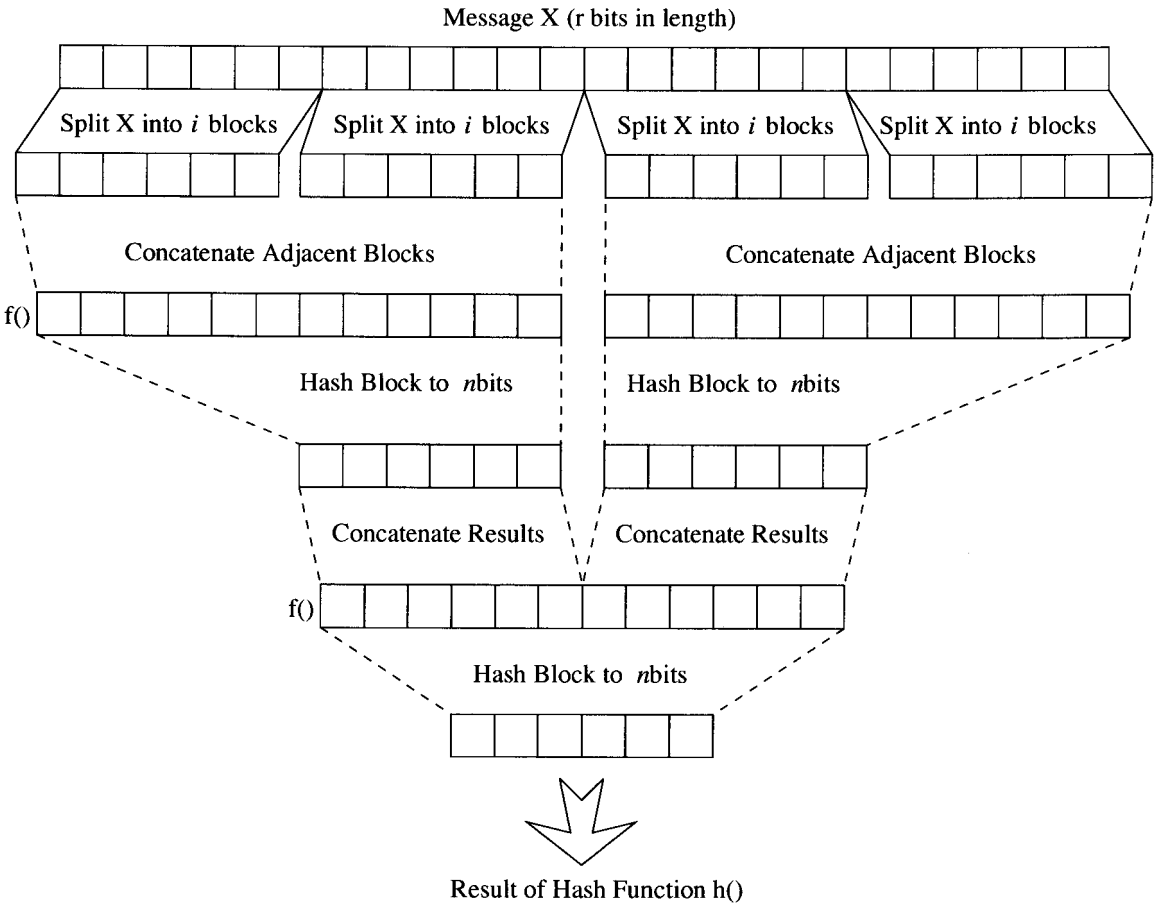


Figure A.1: General Tree Construction for Hash Function

Note that the scheme can be adapted for use as a MAC by making the round function key dependent.

### A.2.2 Practicality

This scheme is faster than the Damgård-Merkle scheme. It is stated in [3] that the hash function, $h()$ can be performed for an $r$ bit input with $\frac{k}{2n}$ processors with:

$$O\left(\frac{n}{r-n} \cdot \log_2\left(\frac{k}{n}\right)\right)$$

evaluations of $f()$. A further advantage of this scheme is the avoidance of chaining and consequently all attacks dependent on the chaining.

The tree hashing scheme has the disadvantage that no known analysis has been performed on this structure. Consequently little is known of its security. Another disadvantage of this scheme is the cost involved. This hash function depends on the use of several processors which can operate in parallel. Since it is required that a hash function should be able to hash messages of arbitrary length, an arbitrary number of processors are required. As the message length $r \rightarrow \infty$ the number of processors, $c \rightarrow \infty$. It can be assumed that the cost of implementing such a hash function would escalate accordingly. An implementation of this scheme is therefore impractical due to the costs involved.

This scheme can be made practical by introducing chaining. This solution should be seen as a hybrid between the Damgård-Merkle scheme and the tree construction scheme. The round function $f()$ in the Damgård-Merkle scheme is effectively replaced by the tree hashing scheme. The hybrid scheme has the disadvantages of reducing the performance and re-introducing attacks dependent on the chaining.

Thus a trade-off between speed, cost and security has to be made when using the tree construction. According to [3] a speedup factor of $c$ is achieved if $c$ processors are used. It is possible to use the tree construction not only in the construction of a hash function, $h()$, but also in the construction of a round function $f()$. It is not known if any practical hash functions are based on this scheme.

## A.3 CASCADING OF HASH FUNCTIONS

The following observations are made in [3] regarding the cascading of hash functions. Let $A||B$ denote concatenation of message $B$ to $A$. If $h_1()$ and $h_2()$ are hash functions that

produces hash values of $n_1$ and $n_2$ bits respectively, and $g()$ is a one way function that yields a $n_3$-bit result, then:

1. $h(X_1, X_2) = g(h_1(X_1)||h_1(X_2))$ is a CRHF if $h_1()$ is a CRHF and $g()$ is a CRF.
2. $h(X_1) = g(h_1(X_1)||h_2(X_1))$ is a CRHF if either $h_1()$ or $h_2()$ is a CRHF and $g()$ is a CRF.
3. $h(X_1) = h_1(X_1)||h_2(X_1)$ is a CRHF if either $h_1()$ or $h_2()$ is a CRHF.

The first construction is equivalent to the tree construction discussed in Section A.2 and results in a hash length of $n_3$. The second construction describes a CRHF that is at least as strong as the strongest of $h_1()$ or $h_2()$, provided $g()$ is a CRF. The resultant hash length for the second construction is $n_3$. The third construction omits the use of a CRF, consequently the hash length equals $n_1 + n_2 > n_3$.

In terms of hash speed, the first construction is more efficient since two message blocks are hashed at a time. In terms of security the last two constructions are more secure since two hash functions are used. If one of the hash functions is insecure, the entire construction does not become insecure. These constructions can be extended to more than two hash functions.

## A.4 ROUND FUNCTION CONSTRUCTIONS

### A.4.1 Block Ciphers

Block ciphers are often used in an iterated construction as a round function. The popularity of block ciphers used as round functions are due to the correlation between the requirements set for hash functions and block ciphers. The use of block ciphers has the advantage that the cost and effort of designing and analysing a new round function is drastically reduced, provided that a trusted block cipher is used. An additional advantage when using a block cipher as a round function is that block ciphers are designed to accommodate a secret key. This is an advantage when constructing a MAC. Three disadvantages should be noted when using a block cipher as a round function. The first disadvantage deals with the functional requirement of speed. Hash functions that contain block ciphers as building blocks are slower than dedicated hash functions. The second disadvantage is the introduction of additional

attacks, based on certain properties of blocks ciphers (see Chapter 3 Section 3.6). A third disadvantage is that a number of hash functions have a block size of 64 bits. It is generally believed that the hash length should equal 128 bits or more, in order to provide protection against birthday attacks (see Chapter 3 Section 3.4).

A distinction is made between round functions for which the length of the chaining variable is equal to the block length, and round functions for which the length of the chaining variable is equal to twice the block length of the block cipher.

Before proceeding consider the following notation. Let $E(K, X_i)$ denote the encryption of message block $X_i$ with key $K$.

### Hash Length Equal to Block Length

Figure A.2 depicts a generic configuration for a block cipher used as a round function in a hash function.

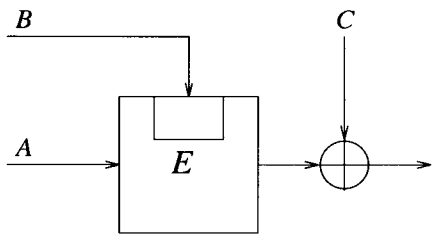


Figure A.2: Generic Round Function from a Block Cipher

Note that three inputs, $A$, $B$, and $C$ are available. Each input can take one of four possible inputs, $X_i$, $H_{i-1}$, $X_i \oplus H_{i-1}$ or a constant. Thus there are $4^3 = 64$ possible configurations of the construction presented in Figure A.2. A general expression for the above construction is given by:

$$\begin{aligned} H_0 &= IV \\ H_i &= E(A, B) \oplus C. \end{aligned}$$

These structures were analysed in [3]. The result of this analysis is that only four of these constructions are considered secure against all attacks. They are defined by the following

expressions [3], [50]:

$$\begin{aligned} H_i &= E(u(H_{i-1}), X_i) \oplus X_i \\ H_i &= E(u(H_{i-1}), (X_i \oplus H_{i-1})) \oplus X_i \oplus H_{i-1} \\ H_i &= E(u(H_{i-1}), X_i) \oplus H_{i-1} \oplus X_i \\ H_i &= E(u(H_{i-1}), (X_i \oplus H_{i-1})) \oplus X_i \end{aligned}$$

The function $u()$ is a mapping from the ciphertext space to the key space. A visual representation of these constructions are presented in Figure A.3.

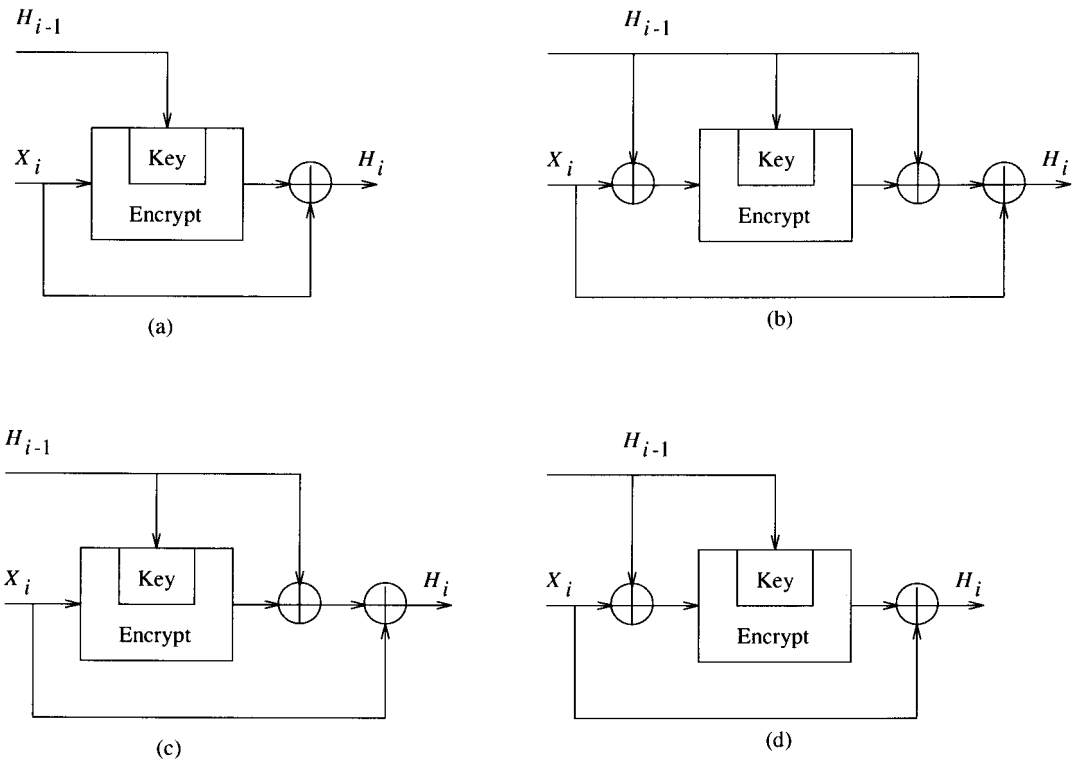


Figure A.3: The Four Secure Round Function Constructions Based on Block Ciphers

Figure A.3(a) is known as the Matyas hash scheme [70]. Figure A.3(c) is known as the Preneel-Miyaguchi hash scheme. The dual of the scheme in Figure A.3(a) is known as the Davies-Meyer scheme and is depicted in Figure A.4.

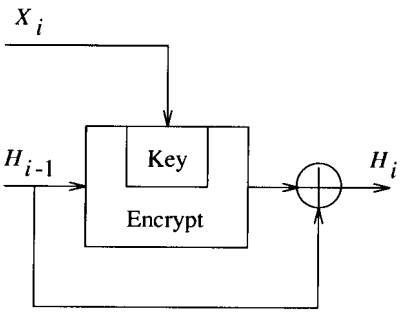


Figure A.4: Davies-Meyer Hash Scheme

The analytical expression for the Davies-Meyer scheme is given by:

$$H_i = E(h(X_i), H_{i-1}) \oplus H_{i-1}.$$

These schemes have been analysed in [3] with regard to:

- Direct attacks.
- Permutation attacks.
- Backward attacks.
- Fixed point attacks.

### Hash Length Equal to Twice the Block Length

Two constructions based on a $n$ bit cipher resulting in a $2 \cdot n$ bit hash result are proposed in [49]. These constructions are based upon the availability of a block cipher with a $n$-bit block size and a $2 \cdot n$ bit key. One such block cipher is IDEA. These constructions are considered variants of the Davies-Meyer scheme mentioned earlier. The first is denoted the tandem Davies-Meyer, and is shown in Figure A.5(a).

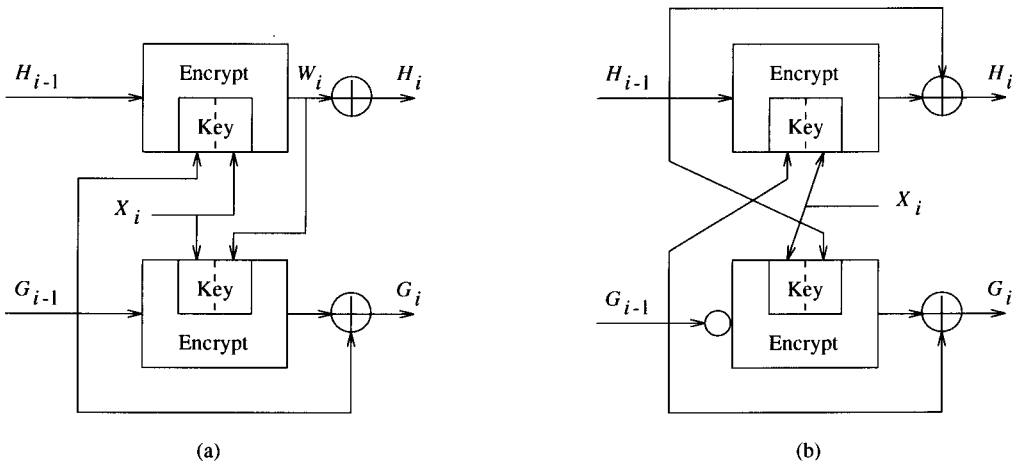


Figure A.5: (a) Tandem Davies-Meyer Scheme (b) Abreast Davies-Meyer Scheme

It is described analytically as follows:

$$\begin{aligned} G_0 &= IV_1 \\ H_0 &= IV_2 \\ W_i &= E(G_{i-1}||X_i, H_{i-1}) \\ G_i &= G_{i-1} \oplus E(X_i||W_{i-1}, G_{i-1}) \\ H_i &= W_i \oplus H_{i-1}. \end{aligned}$$

The Davies-Meyer abreast scheme is also defined in [49] and shown in Figure A.5(b). Analytically the construction is expressed as:

$$\begin{aligned} G_0 &= IV_1 \\ H_0 &= IV_2 \\ G_i &= G_{i-1} \oplus E(X_i||H_{i-1}, (G_{i-1})) \\ H_i &= H_{i-1} \oplus E(G_{i-1}||X_i, (H_{i-1})) \end{aligned}$$

Two additional schemes, which employ block ciphers to construct round functions with a hash length equal to twice the block length, are MDC2 and MDC4. MDC2 is defined as

follows:

$$\begin{aligned} H_0 &= IV_1 \\ H_1 &= IV_2 \\ T1_i &= E(H_{i-1}, X_i) \\ &\quad LT1_i||RT1_i \\ T2_i &= E(G_{i-1}, X_i) \\ &\quad LT2_i||RT2_i \\ H_i &= LT1_i||RT2_i \\ G_i &= LT2_i||RT1_i \end{aligned}$$

with:

$$\begin{aligned} LTx_i &= \text{Lefthand } \frac{n}{2} \text{ bits of } n \text{ bit block.} \\ RTx_i &= \text{Righthand } \frac{n}{2} \text{ bits of } n \text{ bit block.} \\ x &= 1 \text{ or } 2. \end{aligned}$$

MDC4 is defined as the application of two consecutive rounds of MDC2. Thus, the result is that MDC2 is approximately twice as fast as MDC4. It is believed that MDC4 is more secure than MDC2.

Block diagrammatic forms of MDC2 and MDC4 are presented in Figure A.6(a) and Figure A.6(b) respectively.

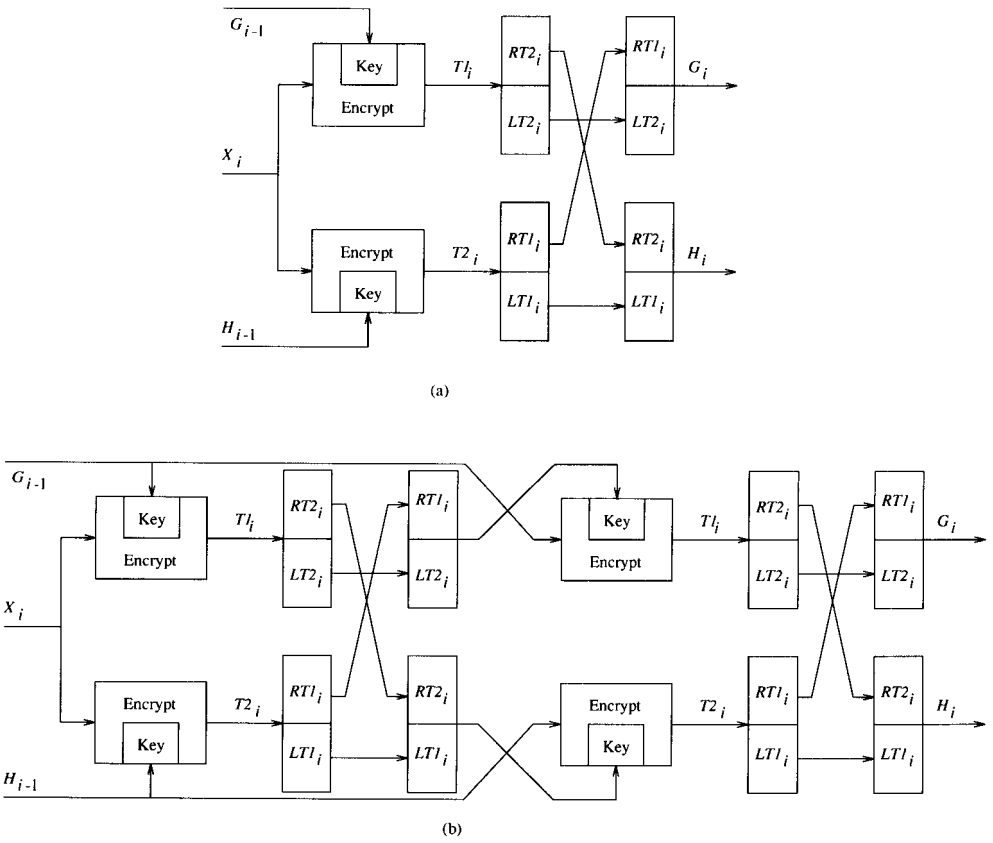


Figure A.6: (a) MDC2 (b) MDC4

Little research has been performed with regard to the security of MDC2 and MDC4.

### MACs Based on Block Ciphers

As shown in Section 5.3 MACs can be based on iterative schemes, providing that the resulting hash value is key dependent. Block ciphers are suitable for constructing MAC round functions since block ciphers are designed to accommodate secret keys. Two constructions based on blocks ciphers are available.

1. Cipher block chaining (CBC).
2. Cipher feedback chaining (CFB).

The block cipher used in CBC mode for a MAC round function construction is described as follows:

$$\begin{aligned} H_0 &= IV \\ H_i &= E(K, X_i \oplus H_{i-1}) \qquad \in \{1, 2, 3, \ldots, j\} \\ H(X) &= H_j. \end{aligned}$$

A graphical representation of this construction for a MAC round function is shown in Figure A.7(a). The block cipher construction used in CFB mode is described as follows:

$$\begin{aligned} H_0 &= IV \\ H_i &= E(K, H_{i-1}) \oplus X_i \qquad \in \{1, 2, 3, \ldots, j\} \\ H(X) &= H_j. \end{aligned}$$

Refer to Figure A.7(b) for a graphical representation of this construction. Note that in the case of CFB, the final result has to be encrypted once again in order to remove the linear dependence of the MAC on the last plaintext block. A third construction was proposed in [71]. It is represented in Figure A.7(c) and is described below:

$$\begin{aligned} H_0 &= IV \\ H_i &= E(K, X_i \oplus H_{i-1}) \oplus X_i \qquad \in \{1, 2, 3, \ldots, j\} \\ H(X) &= H_j. \end{aligned}$$

It is believed that this construction is harder to invert [2] than the previously mentioned constructions. In [3] it is advised that, should encryption of the MAC be required, a different key should be used for encryption purposes.

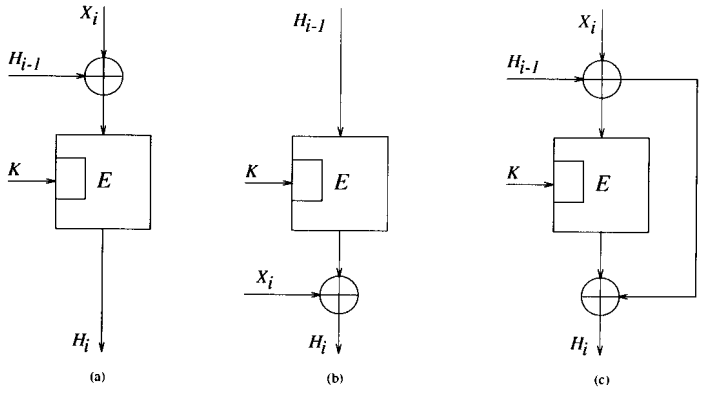


Figure A.7: Block Cipher Based MAC Round Functions

**Suitable Block Ciphers**

A number of block ciphers have been proposed. However, not all block ciphers are suitable for use as round functions in cryptographic hash functions. It should first be noted that the security of the hash function constructed from a block cipher is based on the assumption that the underlying block cipher is secure. Thus block ciphers which are considered insecure should be avoided. In addition certain properties of block ciphers allow the resulting hash function to be susceptible to specific attacks. These properties include:

1. Key collisions.
2. Complementation property.
3. Weak keys.

The manner in which these properties are exploited is considered in Chapter 3, Section 3.5 and 3.6.

### A.4.2 Stream Ciphers

It is conceivable that the round function of a hash function can be based on a stream cipher. This type of construction is hinted at in [69]. It may be possible to adapt the construction presented in [72] to construct a round function for an iterated hash function. Little is known of the security of hash function based on stream ciphers.

## A.5 MAC CONSTRUCTIONS BASED ON MDCS

Traditionally MACs are based on block ciphers (see Section A.4.1). Recently various techniques for constructing hash functions from MDCs were proposed [29], [73], [74]. The preference for the use of MDC based MACs over block cipher based MACs is based on the following factors:

1. Speed of execution.
2. Export restrictions.

Speed of execution is an important functional requirement (see Chapter 4 Section 4.2.4). The matter of export restrictions is a political one. Several countries, most notably the USA, restrict the export of certain cryptographic functions. A large number of block ciphers are covered by these restrictions. Thus, MACs based on block ciphers may not be exported to other countries. It has been proposed that MACs are used in electronic transactions on the Internet [5]. The Internet spans across the globe, and participants from different countries may wish to engage in electronic banking transactions. Thus, MACs based on block ciphers cannot be used for secure Internet Transactions, due to export restrictions. For this reason MACs based on MDCs are preferred over MACs based on block ciphers, since MDCs are not restricted by export controls. Because of these reasons MDC based MACs were adopted in Kerberos, IPSec and SET [5] [29].

It should be remembered that MDCs were not designed to accommodate a key. Thus, when constructing MACs from MDCs, care should be taken in the manner in which the key for the MAC is introduced. The key should be introduced in such a manner that the resulting hash value does not reveal any information of the secret key. This requirement is based on the principles of confusion and diffusion as introduced by Shannon [46].

A number of MAC constructions based on MDCs were proposed. These include:

1. Affix construction.
2. IPSec recommendations
3. NMAC construction.
4. HMAC construction.
5. $MDx$-MAC construction.
6. XOR-MAC construction.

A description of each of these constructions are presented in this section.

### A.5.1 Affix Construction

Three affix constructions are identified. They are:

1. Secret prefix method.
2. Secret suffix method.
3. Envelope method.

These constructions and the security offered by these constructions are considered next.

### Secret Prefix Method

The secret prefix construction was proposed independently in [75] and [76]. This construction requires that the secret key, $K_1$, be prepended to the message $X$. Thus the prefix method can be described as

$$\text{MAC}(X) = h(K_1||X)$$

with:

$$\begin{aligned} h() &= \text{Iterated MDC} \\ || &= \text{Concatenate.} \end{aligned}$$

A graphical representation of this construction is shown in Figure A.8(a). This construction is considered insecure due to the message extension or padding attacks [75] [29]. A variant of the prefix construction with MD5 is used in Kerberos V. This construction is denoted as MD2.5 [77]. Concern is expressed over the security of this construction in [29].

### Secret Suffix Method

This construction is described in [75]. The secret suffix construction appends the secret key $K_2$ to the message $X$ before hashing. The construction is described as follows:

$$\text{MAC}(X) = h(X||K_2)$$

with:

$$\begin{aligned} h() &= \text{Iterated MDC} \\ || &= \text{Concatenate.} \end{aligned}$$

A graphical representation of this construction is shown in Figure A.8(b). A number of attacks on this construction are described in [29]. If off-line attacks are allowed, an internal collision can be found in approximately $O(2^{\frac{n}{2}})$ off-line operations. A second attack is possible if a second pre-image attack on the underlying MDC is possible. A third attack is considered possible if $t$ text-MAC pairs are known. The number of known text-MAC pairs reduces the computational effort to construct a second pre-image from $O(2^n)$ to $O(2^{\frac{n}{t}})$.

**Envelope Method**

The envelope construction is described in [75]. This construction prepends the secret key, $K_1$, and appends the secret key, $K_2$, to the message, $X$, before hashing (see Figure A.8(b)). The construction is described as follows:

$$\text{MAC}(X) = h(K_1||X||K_2)$$

with:

$$\begin{aligned} h() &= \text{Iterated MDC} \\ || &= \text{Concatenate.} \end{aligned}$$

In [75] it is claimed that the effective key length for the envelope construction is equal to the length of $K_1$ ($k_1$ bits) added to the length of $K_2$ ($k_2$ bits). Thus according to [75] $O(2^{k_1+k_2})$ operations are required to establish a collision. This is shown to be incorrect in [29]. In [29] it is demonstrated that the effective key length is less than or equal $k_i + 1$, for whichever is the larger of $k_1$ or $k_2$. This implies that the number of operations required to establish a collision are less than or equal to $O(2^{k_i+1})$ for whichever is the larger, $k_1$ or $k_2$. Thus the security gained by selecting $K_1 \neq K_2$ is less than expected. A divide and conquer attack on the envelope method is described that establishes an internal collision and then searches exhaustively for the two respective keys.

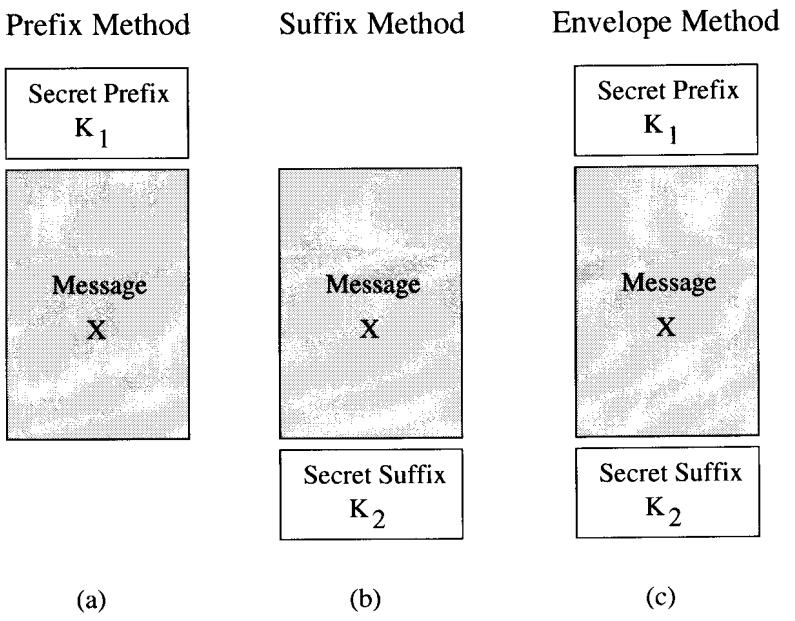


Figure A.8: Affix Constructions

The attacks on the affix constructions are easier than attacks on an ideal MAC. However, provided that the hash function, $h()$, is collision resistant, the attacks on the affix constructions remain computationally infeasible. Key recovery attacks on these constructions are presented in [31].

### A.5.2 IPSec recommendations

In [28] three MAC constructions based on MD5 are presented. These proposals were submitted to the IPSec working group. The constructions could be viewed as variations of the affix methods described in Section A.5.1. The three proposals are summarised below:

1.

$$\text{MAC}(X) = \text{MD5}(K_1||\text{MD5}(K_2||X))$$

with:

$$\begin{aligned} \text{MD5}() &= \text{MD5 hash function} \\ || &= \text{Concatenate} \\ K_1 &= \text{128 bit key} \\ K_2 &= \text{128 bit key} \\ X &= \text{Message.} \end{aligned}$$

2.

$$\text{MAC}(X) = \text{MD5}(K_1||p||X||K_1)$$

with:

$$\begin{aligned} \text{MD5}() &= \text{MD5 hash function} \\ || &= \text{Concatenate} \\ K_1 &= \text{128 bit key} \\ p &= \text{384 padding bits} \\ X &= \text{Message.} \end{aligned}$$

3.

$$\text{MAC}(X) = \text{MD5}(K_1||\text{MD5}(K_1||X))$$

with:

$$\begin{aligned} \text{MD5}() &= \text{MD5 hash function} \\ || &= \text{Concatenate} \\ K_1 &= \text{128 bit key} \\ X &= \text{Message.} \end{aligned}$$

The second proposal is effectively the envelope method with $K_1$ padded to form a 512 bit block. In addition it differs from the envelope method since $K_1 \neq K_2$. Thus, $K_1$ specifies an initial value for the MD5 hash algorithm applied to the message block. Reservation has been expressed on the use of the initial value as a secret key [3]. The key is however also appended to the message for hashing. This should increase the security of the resultant

MAC. This technique is considered susceptible to a divide and conquer attack as described in Section A.5.1.

In [28] it is stated that the chosen message attack requires $2^{64}$ chosen texts. In [29] it shows that this can be reduced to $2^{56.5}$ known text-MAC pairs if it is assumed that the number of trailing blocks, $s$, are $2^{16}$.

### A.5.3 NMAC Construction

NMAC is an acronym for nested message authentication code. It is defined in [73] and [78]. It is a generic construction of the following form:

$$\text{NMAC}(X) = h_{K_2}(h_{K_2}(X))$$

with:

$$\begin{aligned} h_{K_i}() &= \text{Keyed hash function or a MAC} \\ K_1 &= \text{Key 1} \\ K_2 &= \text{Key 2} \\ X &= \text{Message.} \end{aligned}$$

The NMAC construction does not propose a technique for constructing a keyed hash function. The security of this construction is based on the conditions imposed on the compress function of the keyed hash function and the iterated hash function itself.

### A.5.4 HMAC Construction

The HMAC construction is a variant of the NMAC construction for which the $IV$ is fixed. This construction requires no changes to the MDC used for constructing a MAC. The construction involves a single key, $K$, of length $k$ bits. The use of a single key is advantageous with regard to key management and its associated problems. The HMAC construction is defined in [73], [74], [78] as:

$$\text{HMAC}(X) = h(\overline{K} \oplus \texttt{opad} || h(\overline{K} \oplus \texttt{ipad} || X))$$

with:

$h()$ = Hash function

$\overline{K}$ = The key, $K$, padded with $0's$ to form an elementary block

`opad` = The byte `0X36` repeated to form a elementary block

`ipad` = The byte `0X5C` repeated to form a elementary block

$\oplus$ = Bitwise XOR

$||$ = Concatenation

$X$ = Message.

This procedure is summarised in Figure A.9

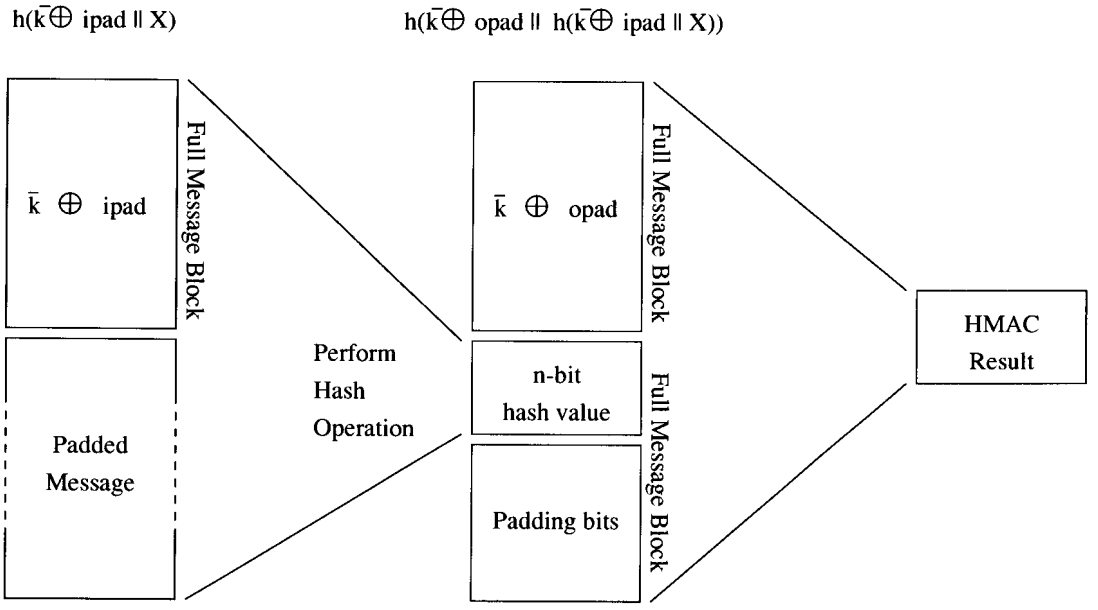


Figure A.9: HMAC Construction

The security of the HMAC construction is based on the security of NMAC. The relation between these two constructions are found in the construction of the two derived keys $K_1$ =`opad` $\oplus\overline{K}$ and $K_2$ =`ipad` $\oplus\overline{K}$. Thus HMAC is a specific instance of NMAC. It is stated in [73] that attacks against HMAC may exist, but that these attacks are not necessarily applicable to NMAC. Note that effectively using the same key, $K$, in both applications of $h()$ does not weaken the construction significantly due to the existence of the divide and conquer attack mentioned in Section A.5.1.

The HMAC construction has become the mandatory construction for use in authentication transforms for Internet security protocols. The HMAC construction is also specified in the SET specification [5]. At present all of the known generic attacks against HMAC are considered infeasible [73], [74], [78].

### A.5.5 $MDx$-MAC Construction

This construction is suggested in [29]. The following design goals were set for this construction:

1. The secret key should be involved at the beginning, at the end, and in every iteration of the hash function.
2. The deviation of the original hash function should be minimal.
3. The performance should be close to that of the hash function.
4. Additional memory requirements should be kept minimal.
5. The approach should be generic, i.e. should apply to any hash function based on the same principles as MD4.

This construction can be used with MD5, RIPEMD or SHA. MD4 is omitted due to the attack described in [17]. In this section $MDx$ refers to one of the three hash functions mentioned above. Let $\overline{MDx}$ refer to an implementation of $MDx$ with both padding and appending length omitted. The resulting construction utilises three 128 bit (16 byte) constants, $T_0$, $T_1$ and $T_2$. These constants are used to construct three additional 768 bit constants $U_0$, $U_1$ and $U_2$. If the key is shorter than 128 bits, the key is expanded to be of a 128 bits length. Once this is accomplished, three sub-keys, $K_0$, $K_1$ and $K_2$, are derived as follows:

$$K_i = \overline{MDx}(K||U_i||K) \qquad i \in \{0, 1, 2\}$$

The constants $U_i$ are required to ensure that the hash is computed over two iterations of the hash function, thus increasing the difficulty of retrieving $K$ from any of the $K_i$, even if two of the sub-keys are known (see Figure A.10). The mapping from $K$ to $K_i$ is not bijective, but the reduction in entropy is believed to negligible [29].

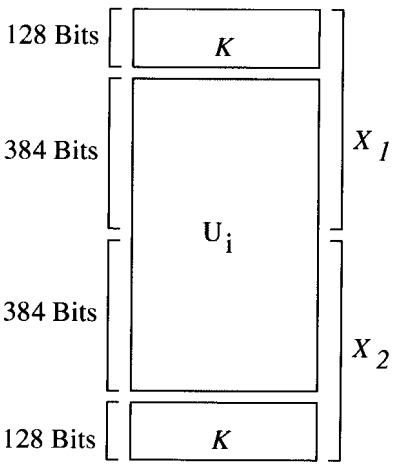


Figure A.10: $MDx$ Key Expansion

Once this step is completed the leftmost 128 bits of the sub-key $K_1$ is split into four 32-bit blocks denoted as $K_1[i]$, with $0 \leq i \leq 3$ (see Figure A.11).

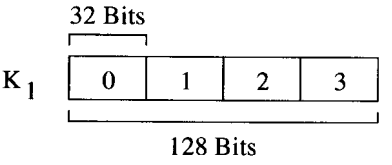


Figure A.11: $MDx$ $K_1$ Sub-key Partitioning

The resulting MAC is now calculated as follows:

1. The initial value, $IV$, of $MDx$ is replaced by $K_0$.
2. $K_1[i \bmod 4]$ is added modulo $2^{32}$ to the constants used in round $i$ of each iteration of $MDx$.
3. Following the last block after normal processing of $MDx$ (i.e. including the padding and addition of message length), append an additional 512 bit block. The additional block is derived from the constants $T_0$, $T_1$, $T_2$ and the sub-key $K_2$ as shown below:

$$K_2||K_2 \oplus T_0||K_2 \oplus T_1||K_2 \oplus T_2.$$

4. The MAC result is the leftmost $m$ bits of the resulting hash value. It is advised that $m = \frac{n}{2}$.

A summary of the above procedure is shown in Figure A.12.

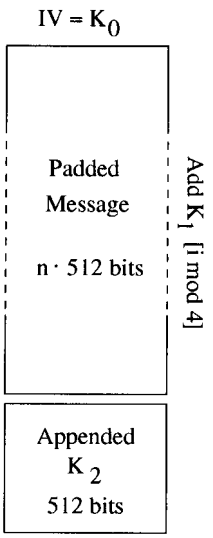


Figure A.12: $MDx$ MAC construction

Let $s$ represent the number of common trailing blocks in two messages. In [29] it is stated that if the MAC length $m = \frac{n}{2}$, a forgery attack requires $O(\frac{2^m}{s+1})$ chosen text-MAC pairs and $O(\frac{2^m}{\sqrt{s+1}})$ known texts. Thus $MDx$-MAC is more secure than the envelope method described in Section A.5.1. It is also stated in [29] that the divide and conquer attack described in Section A.5.1 is not applicable to $MDx$-MAC.

### A.5.6 XOR-MAC Constructions

This construction is described in [79] and [80] and resembles the multiple message hash scheme described in [68] and [69]. This scheme does not show how to construct a MAC from a MDC, but does make use of keyed MDCs in the construction of the MAC. A generic description of this scheme is presented below:

$$\text{XOR-MAC}(X) = F_k(X[1]) \oplus F_k(X[2]) \oplus F_k(X[3]) \oplus \ldots \oplus F_k(X[m])$$

with:

$$\begin{aligned} F_k() &= \text{Keyed pseudorandom function} \\ \oplus &= \text{Bitwise XOR} \\ X &= \text{Message} \\ m &= \text{Number of elementary blocks in message } X \\ X[i] &= \text{Elementary message block i} \quad i \in \{1, 2, 3, \ldots m\}. \end{aligned}$$

Thus the message is divided into elementary block lengths and then processed by a pseudorandom function. The pseudorandom function, PRF, should be keyed. It is suggested in [79] and [80] that the PRF could be either a block cipher, or a keyed hash function. The result of the PRF for each message block is then XOR'ed with the previous result. Once the last XOR is performed, the MAC is calculated. Figure A.13 presents a visual interpretation of this scheme.

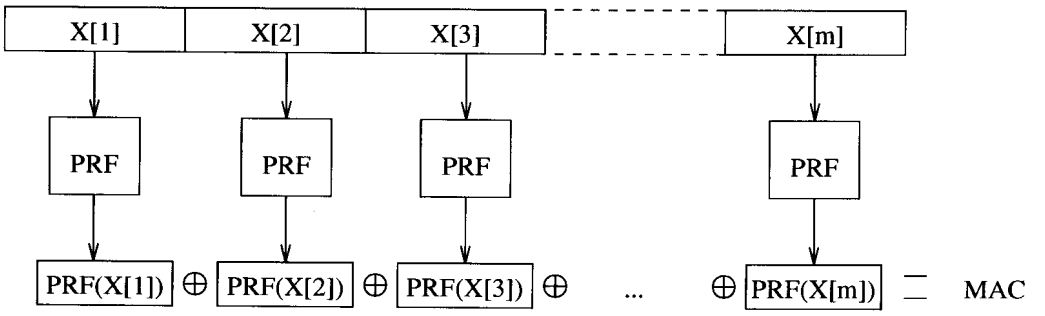


Figure A.13: XOR-MAC construction

Concrete proposals for schemes making use of this construction are presented in [79] and [80], followed by analysis of the security of these proposals. This scheme is highly parallelliseable and has the additional advantages of out of order verification. If only a single block in a message is changed, the output can be updated without recomputing the entire MAC.

## A.6 INTERNATIONAL STANDARDS

A number of hash function constructions are under consideration for standardisation. A summary of these can be found in chapter 3 and chapter 5 of [81]. The multipart standard, ISO/IEC 10118, contains the following proposals.

**ISO/IEC 10118-1:** This part of the standard provides general definitions and background to the remainder of the standard. The iterative hash function construction as defined in Section 5.3 is contained in this part of the proposed standard.

**ISO/IEC 10118-2:** This part of the standard specifies hash functions constructed from block ciphers. Two methods are specified. The first method is the general construction specified in Section A.4.1 with the block length equal to the hash length (see Figure A.2). The second is equal to MDC2 (see Figure A.6(a)).

**ISO/IEC 10118-3:** This part of the standard describes the following three dedicated hash functions, SHS (secure hash standard), RIPEMD-128 and RIPEMD-160.

**ISO/IEC 10118-4:** This part of the standard specifies two hash algorithms based on modular arithmetic, namely MASH-1 and MASH-2.

In addition to the above standard, ISO/IEC 9797 specifies a method for using a key and a $n$-bit block cipher to construct a MAC. The process is summarised as follows:

1. Pad data to form a $n$-bit block.
2. Encipher data in CBC mode.
3. The final ciphertext block is the resulting MAC.

The construction specified in ISO/IEC 9797 corresponds to the construction shown in Figure A.7(a).

## A.7 CONCLUSION

When selecting a hash function construction the construction should be evaluated according to the requirements set in Chapter 4. A trade-off between cost, security and speed has to be made. The generic attacks described in Chapter 3 should be infeasible.

A final matter of interest is the matter of injectivity, surjectivity and bijectivity of the round function used in an iterated scheme. In [30] it is stated that an injective function should never be used in an iterated hash function. In [82] the question is raised whether a bijective round

function allows stronger security claims. This question is answered in part in [29], where it is shown that the known and chosen text attack is not applicable to MACs with bijective round functions. The influence of the bijectivity of the round functions of MDCs and MACs on the security of the entire construction remains unresolved.

# APPENDIX B: SOURCE CODE: IMPLEMENTATION OF MD4

This appendix contains an implementation of the MD4 hash algorithm as described in [10] and [44].

```
/* This header file includes the functions used to implement the MD4 algorithm
 * as described in Crypto 91 by R.Rivest
 *
 * Author: P.R. Kasselman
 * Date: August 20, 1996
 * Filename: md4.h
 * Copyright: Ciphertec cc */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define A0 0x67452301
#define B0 0xefcdab89
#define C0 0x98badcfe
#define D0 0x10325476
#define ROOT2 0x5a827999
#define ROOT3 0x6ed9eba1

#define FS1 3
#define FS2 7
#define FS3 11
#define FS4 19

#define GS1 3
#define GS2 5
#define GS3 9
#define GS4 13

#define HS1 3
#define HS2 9
#define HS3 11
#define HS4 15

int PadBit(int argc, char filename[], unsigned int *PadLen);
void Init(unsigned int *A, unsigned int *B, unsigned int *C, unsigned int *D);
void SaveParms(unsigned int A, unsigned int B, unsigned int C,
       unsigned int D, unsigned int *AA, unsigned int *BB,
       unsigned int *CC, unsigned int *DD);
void ReadArray(int argc, char filename[], unsigned int M[], int n);
unsigned int Rotate(unsigned int X, unsigned int s);
unsigned int FunctionF(unsigned int X, unsigned int Y, unsigned int Z);
unsigned int FunctionG(unsigned int X, unsigned int Y, unsigned int Z);
unsigned int FunctionH(unsigned int X, unsigned int Y, unsigned int Z);
void Round1(unsigned int *A, unsigned int *B, unsigned int *C, unsigned in-
t *D,
    unsigned int X[]);
void Round2(unsigned int *A, unsigned int *B, unsigned int *C, unsigned in-
t *D,
    unsigned int X[]);
```

```
void Round3(unsigned int *A, unsigned int *B, unsigned int *C, unsigned in-
t *D,
    unsigned int X[]);
void Update(unsigned int *A, unsigned int *B, unsigned int *C, unsigned in-
t *D,
    unsigned int AA, unsigned int BB, unsigned int CC,
    unsigned int DD);
void PrintSignature(unsigned int A, unsigned int B, unsigned int C,
    unsigned int D);
void PrintReverse(unsigned int X);
void RestoreFile(char filename[], unsigned int FileLen);

int md4(int argc, char filename[]);

unsigned int RotateRight(unsigned int X, unsigned int s);

/* This routine performs the bit padding as requied for the MD4 algorithm */
int PadBit(int argc, char filename[], unsigned int *PadLen)
{

   unsigned int i,j;
   unsigned int FileLen, FileBits, PadBits, PadBytes, TempInt;
   unsigned char temp;
   FILE *fp;

   FileLen = 0;

   if(argc != 2)
     {
printf("No file specified\n");
exit(1);
     }

   fp = fopen(filename, "r+b");

   if(!fp)
     {
printf("Error opening file\n");
fclose(fp);
exit(1);
     }

   /* Determine the size of the file */

   while(!feof(fp))
     {
fread(&temp, sizeof(unsigned char), 1, fp);
if(!feof(fp))
       {
  FileLen++;
       }
     }

   FileBits = FileLen*8*sizeof(unsigned char);
```

```
    /* Compute the number of bits needed for padding */

    PadBits = abs( (448 - FileBits) % 512 );
    PadBytes = PadBits/8;

    /* Pad bit "1" */
    temp = 0x80;
    fwrite(&temp, sizeof(unsigned char), 1, fp);

    /* Pad zero bits */
    temp = 0;
    for(i=0; i<PadBytes-1; i++)
      {
fwrite(&temp, sizeof(unsigned char), 1, fp);
      }

    /* Append the size of the file
     * (For this implimentation no file larger than 2^32 is expected)*/

    TempInt = FileBits;
    fwrite(&TempInt, sizeof(unsigned int), 1, fp);
    TempInt = 0x00;
    fwrite(&TempInt, sizeof(unsigned int), 1, fp);

    *PadLen = (int)(ceil((double)(FileLen+1) / 64)*64);

    fclose(fp);

    return(FileLen);
}

/* Initialise the Buffers to be processed */

void Init(unsigned int *A, unsigned int *B, unsigned int *C, unsigned int *D)
{
    *A = A0;
    *B = B0;
    *C = C0;
    *D = D0;
}

/* This function updates the holding variables AA, BB, CC, DD */

void SaveParms(unsigned int A, unsigned int B, unsigned int C,
        unsigned int D, unsigned int *AA, unsigned int *BB,
        unsigned int *CC, unsigned int *DD)
{
    *AA = A;
    *BB = B;
    *CC = C;
    *DD = D;
}

/* This function reads the modified data file and updates the M array */
```

```
void ReadArray(int argc, char filename[], unsigned int M[], int n)
{
   unsigned int i;
   unsigned int TempInt;
   FILE *fp;

   if(argc != 2)
     {
printf("No file specified\n");
exit(1);
     }

   fp = fopen(filename, "r+b");

   if(!fp)
     {
printf("Error opening file\n");
fclose(fp);
exit(1);
     }

   /* Read the file */

   for(i=0; i<n; i++)
     {
fread(&TempInt, sizeof(unsigned int), 1, fp);
M[i] = TempInt;
     }

   fclose(fp);
}

void Round1(unsigned int *A, unsigned int *B, unsigned int *C, unsigned in-
t *D,
    unsigned int X[])
{
   *A = Rotate((*A + FunctionF(*B,*C,*D) + X[0]), 3);
   *D = Rotate((*D + FunctionF(*A,*B,*C) + X[1]), 7);
   *C = Rotate((*C + FunctionF(*D,*A,*B) + X[2]), 11);
   *B = Rotate((*B + FunctionF(*C,*D,*A) + X[3]), 19);

   *A = Rotate((*A + FunctionF(*B,*C,*D) + X[4]), 3);
   *D = Rotate((*D + FunctionF(*A,*B,*C) + X[5]), 7);
   *C = Rotate((*C + FunctionF(*D,*A,*B) + X[6]), 11);
   *B = Rotate((*B + FunctionF(*C,*D,*A) + X[7]), 19);

   *A = Rotate((*A + FunctionF(*B,*C,*D) + X[8]), 3);
   *D = Rotate((*D + FunctionF(*A,*B,*C) + X[9]), 7);
   *C = Rotate((*C + FunctionF(*D,*A,*B) + X[10]), 11);
   *B = Rotate((*B + FunctionF(*C,*D,*A) + X[11]), 19);

   *A = Rotate((*A + FunctionF(*B,*C,*D) + X[12]), 3);
   *D = Rotate((*D + FunctionF(*A,*B,*C) + X[13]), 7);
   *C = Rotate((*C + FunctionF(*D,*A,*B) + X[14]), 11);
   *B = Rotate((*B + FunctionF(*C,*D,*A) + X[15]), 19);
```

```
}

void Round2(unsigned int *A, unsigned int *B, unsigned int *C, unsigned in-
t *D,
    unsigned int X[])
{
   *A = Rotate((*A + FunctionG(*B,*C,*D) + X[0] + ROOT2), 3);
   *D = Rotate((*D + FunctionG(*A,*B,*C) + X[4] + ROOT2), 5);
   *C = Rotate((*C + FunctionG(*D,*A,*B) + X[8] + ROOT2), 9);
   *B = Rotate((*B + FunctionG(*C,*D,*A) + X[12] + ROOT2), 13);

   *A = Rotate((*A + FunctionG(*B,*C,*D) + X[1] + ROOT2), 3);
   *D = Rotate((*D + FunctionG(*A,*B,*C) + X[5] + ROOT2), 5);
   *C = Rotate((*C + FunctionG(*D,*A,*B) + X[9] + ROOT2), 9);
   *B = Rotate((*B + FunctionG(*C,*D,*A) + X[13] + ROOT2), 13);

   *A = Rotate((*A + FunctionG(*B,*C,*D) + X[2] + ROOT2), 3);
   *D = Rotate((*D + FunctionG(*A,*B,*C) + X[6] + ROOT2), 5);
   *C = Rotate((*C + FunctionG(*D,*A,*B) + X[10] + ROOT2), 9);
   *B = Rotate((*B + FunctionG(*C,*D,*A) + X[14] + ROOT2), 13);

   *A = Rotate((*A + FunctionG(*B,*C,*D) + X[3] + ROOT2), 3);
   *D = Rotate((*D + FunctionG(*A,*B,*C) + X[7] + ROOT2), 5);
   *C = Rotate((*C + FunctionG(*D,*A,*B) + X[11] + ROOT2), 9);
   *B = Rotate((*B + FunctionG(*C,*D,*A) + X[15] + ROOT2), 13);

}

void Round3(unsigned int *A, unsigned int *B, unsigned int *C, unsigned in-
t *D,
    unsigned int X[])
{
   *A = Rotate((*A + FunctionH(*B,*C,*D) + X[0] + ROOT3), 3);
   *D = Rotate((*D + FunctionH(*A,*B,*C) + X[8] + ROOT3), 9);
   *C = Rotate((*C + FunctionH(*D,*A,*B) + X[4] + ROOT3), 11);
   *B = Rotate((*B + FunctionH(*C,*D,*A) + X[12] + ROOT3), 15);

   *A = Rotate((*A + FunctionH(*B,*C,*D) + X[2] + ROOT3), 3);
   *D = Rotate((*D + FunctionH(*A,*B,*C) + X[10] + ROOT3), 9);
   *C = Rotate((*C + FunctionH(*D,*A,*B) + X[6] + ROOT3), 11);
   *B = Rotate((*B + FunctionH(*C,*D,*A) + X[14] + ROOT3), 15);

   *A = Rotate((*A + FunctionH(*B,*C,*D) + X[1] + ROOT3), 3);
   *D = Rotate((*D + FunctionH(*A,*B,*C) + X[9] + ROOT3), 9);
   *C = Rotate((*C + FunctionH(*D,*A,*B) + X[5] + ROOT3), 11);
   *B = Rotate((*B + FunctionH(*C,*D,*A) + X[13] + ROOT3), 15);

   *A = Rotate((*A + FunctionH(*B,*C,*D) + X[3] + ROOT3), 3);
   *D = Rotate((*D + FunctionH(*A,*B,*C) + X[11] + ROOT3), 9);
   *C = Rotate((*C + FunctionH(*D,*A,*B) + X[7] + ROOT3), 11);
   *B = Rotate((*B + FunctionH(*C,*D,*A) + X[15] + ROOT3), 15);

}
```

```
unsigned int FunctionF(unsigned int X, unsigned int Y, unsigned int Z)
{
    unsigned int W;

    W = ((X&Y) | ((~X)&Z));

    return(W);
}

unsigned int FunctionG(unsigned int X, unsigned int Y, unsigned int Z)
{
    unsigned int W;

    W = ((X&Y) | (X&Z) | (Y&Z));

    return(W);
}

unsigned int FunctionH(unsigned int X, unsigned int Y, unsigned int Z)
{
    unsigned int W;

    W = (X^Y^Z);

    return(W);
}

unsigned int Rotate(unsigned int X, unsigned int s)
{
    unsigned int temp;

    temp = X;
    X = (X << s) | (temp >> (32-s));
    return(X);
}

void Update(unsigned int *A, unsigned int *B, unsigned int *C, unsigned in-
t *D,
     unsigned int AA, unsigned int BB, unsigned int CC,
     unsigned int DD)
{
    *A = *A + AA;
    *B = *B + BB;
    *C = *C + CC;
    *D = *D + DD;
}


void PrintSignature(unsigned int A, unsigned int B, unsigned int C,
     unsigned int D)
{
   printf("Signature: ");
   PrintReverse(A);
   PrintReverse(B);
```

```
    PrintReverse(C);
    PrintReverse(D);
    printf("\n");
}

void PrintReverse(unsigned int X)
{
    int i;

    for(i=0; i<4; i++)
      {
printf("%.2x", X & 0x000000ff);
X = X >> 8;
      }

}

void RestoreFile(char filename[], unsigned int FileLen)
{
    unsigned char TempChar;
    unsigned char *array;
    unsigned int i;
    FILE *fp;

    fp = fopen(filename, "r+b");

    if(!fp)
      {
printf("Error opening file\n");
fclose(fp);
exit(1);
      }

    /* Allocate dynamic memmory */

    array = (unsigned char *)calloc(FileLen, sizeof(unsigned char));

    /* Read the file */

    for(i=0; i<FileLen; i++)
      {
fread(&TempChar, sizeof(unsigned char), 1, fp);
array[i] = TempChar;
      }

    fclose(fp);

    fp = fopen(filename, "wb");

    if(!fp)
      {
printf("Error opening file\n");
fclose(fp);
exit(1);
      }
```

```
    /* Write the file */

    for(i=0; i<FileLen; i++)
      {
TempChar = array[i];
fwrite(&TempChar, sizeof(unsigned char), 1, fp);
      }

    fclose(fp);

    /* Liberate some memory */

    free(array);

}

/* This function uses the above routines to construct a MD4 signature */

/* NB set argc = 2 */

int md4(int argc, char filename[])
{
    unsigned int i,j;
    unsigned int FileLen, PadLen;
    unsigned int A, B, C, D, AA, BB, CC, DD;
    unsigned int *M, *X;

    FileLen = 0;

    FileLen = PadBit(argc, filename, &PadLen);

    M = (unsigned int *)calloc(PadLen/4, sizeof(unsigned int));
    X = (unsigned int *)calloc(16, sizeof(unsigned int));

    ReadArray(argc, filename, M, PadLen/4);

    RestoreFile(filename, FileLen);

    Init(&A, &B, &C, &D);

    for(i=0; i<PadLen/64; i++)
      {

SaveParms(A, B, C, D, &AA, &BB, &CC, &DD);

for(j=0; j<16; j++)
  {
     X[j] = M[i*16+j];
  }

Round1(&A, &B, &C, &D, X);
Round2(&A, &B, &C, &D, X);
Round3(&A, &B, &C, &D, X);
```

```
Update(&A, &B, &C, &D, AA, BB, CC, DD);
     }

   PrintSignature(A,B,C,D);

   /* Liberate some memory */

   free(M);
   free(X);

   return(0);
}

unsigned int RotateRight(unsigned int X, unsigned int s)
{
   unsigned int temp;

   temp = X;
   X = (X >> s) | (temp << (32-s));
   return(X);
}
```

# APPENDIX C: SOURCE CODE: ATTACK ON ALL THREE ROUNDS OF MD4

This is an implementation of the attack on MD4 as described by Dobbertin in [14]. Algorithm 6.3 is used for finding admissible inner almost-collisions in this implementation. This attack yields two messages that hash to the same value in less than one minute.

```
/* This a working version of the full attack on MD4. The alternative algorithm
 * for establishing inner almost-collisions is used in this program.
 *
 * Author: P.R. Kasselman
 * Filename: md4ga5.c
 * Date: 11 October 1996
 * Copyright: Ciphertec (1996) */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include "md4.h"

#define A 0
#define B 1
#define C 2
#define D 3

int main()
{
   unsigned int i,j,k,l, Iteration;
   unsigned int Bi, Ci, U, Ut, V, Vt, W, Wt, Z, Zt;
   unsigned int As, Ds, Bs, Bst, Cs, Cst;
   unsigned int Condition, NewZ, NewW, Final, NextPhase;
   unsigned int DeltaW, DeltaV;
   unsigned int TempInt;
   unsigned int ABCD0[47][4], ABCD1[47][4], Delta19[4];
   unsigned int X1[16], X2[16];
   unsigned int Flag23, Flag27, DispFlag23, DispFlag27;
   unsigned int LastCondition, Collision, CollisionFlag;
   FILE *fp;

   time_t TheTime;

   TheTime = time(NULL);
   srand(TheTime);

   Iteration = 0;

   U = -1;
   Ut = 0;

   V = 0xfffdfffe;
   Vt = V;

   Wt = 0xfdffdfff;
   W = Wt + 0xeffff000;
```

```
    Bi = 0;
    Ci = 0;

    NextPhase = 1;

    while(NextPhase != 0)
      {
Bst = rand();
Bs = Bst + Rotate(1,25);

Cs = rand();
Cst = Cs + Rotate(1,5);

NewW = FunctionF(Vt,Ut,Bi) - FunctionF(V,U,Bi);
DeltaW = Rotate(Wt,21) - Rotate(W,21);

NewZ = 1;
Condition = 1;

while(Condition != -1)
  {
     while(NewZ !=0)
       {
  Zt = rand() | 0x00000000;
  Z = Zt + 0x00001001;

  NewZ = FunctionF(Wt,Vt,Ut) - FunctionF(W,V,U) -
    Rotate(Zt,13) + Rotate(Z,13);
     }

     NewZ = 1;

     Condition = FunctionG(Zt,Wt,Vt) - FunctionG(Z,W,V);
  }

DeltaV = 1;

while(DeltaV != 0)
  {
     As = rand();
     DeltaV = FunctionG(As,Zt,Wt) - FunctionG(As,Z,W);
  }

TempInt = 1;
Final = 1;

while(Final != 0)
  {
     while(TempInt != 0)
       {
  Cs = rand();
  Cst = Cs + Rotate(1,5);

  Bst = rand();
  Bs = Bst + Rotate(1,25);
```

```
   Ds = rand();
   TempInt = FunctionG(Ds,As,Zt) - FunctionG(Ds,As,Z) - W + Wt
     - Rotate(Cst,23) + Rotate(Cs,23);

        }

      TempInt = 1;

      Final = FunctionG(Cst,Ds,As) - FunctionG(Cs,Ds,As) - Z + Zt
        - Rotate(Bst,19) + Rotate(Bs,19) + 1;


   }


NextPhase = FunctionG(Bst,Cst,Ds) - FunctionG(Bs,Cs,Ds);
      }


    if(NextPhase == 0)
      {
printf("An admissable inner collision was found\n");
      }

    ABCD0[11][C] = Ci;
    ABCD1[11][C] = Ci;

    ABCD0[11][B] = Bi;
    ABCD1[11][B] = Bi;

    ABCD0[15][A] = U;
    ABCD1[15][A] = Ut;

    ABCD0[15][D] = V;
    ABCD1[15][D] = Vt;

    ABCD0[15][C] = W;
    ABCD1[15][C] = Wt;

    ABCD0[15][B] = Z;
    ABCD1[15][B] = Zt;

    ABCD0[19][A] = As;
    ABCD1[19][A] = As;

    ABCD0[19][D] = Ds;
    ABCD1[19][D] = Ds;

    ABCD0[19][C] = Cs;
    ABCD1[19][C] = Cst;

    ABCD0[19][B] = Bs;
    ABCD1[19][B] = Bst;
```

```
/* Find the message values that corrosponds to the computed values */

X1[13] = rand();
X2[13] = X1[13];

X1[14] = RotateRight(ABCD0[15][C],11) - ABCD0[11][C] -
  FunctionF(ABCD0[15][D],ABCD0[15][A],ABCD0[11][B]);

X2[14] = X1[14];

X1[15] = RotateRight(ABCD0[15][B],19) - ABCD0[11][B] -
  FunctionF(ABCD0[15][C],ABCD0[15][D],ABCD0[15][A]);

X2[15] = X1[15];

X1[0] = RotateRight(ABCD0[19][A],3) - ABCD0[15][A] -
  FunctionG(ABCD0[15][B],ABCD0[15][C],ABCD0[15][D]) - ROOT2;

X2[0] = X1[0];

X1[4] = RotateRight(ABCD0[19][D],5) - ABCD0[15][D] -
  FunctionG(ABCD0[19][A],ABCD0[15][B],ABCD0[15][C]) - ROOT2;

X2[4] = RotateRight(ABCD1[19][D],5) - ABCD1[15][D] -
  FunctionG(ABCD1[19][A],ABCD1[15][B],ABCD1[15][C]) - ROOT2;


X1[8] = RotateRight(ABCD0[19][C],9) - ABCD0[15][C] -
  FunctionG(ABCD0[19][D],ABCD0[19][A],ABCD0[15][B]) - ROOT2;

X2[8] = RotateRight(ABCD1[19][C],9) - ABCD1[15][C] -
  FunctionG(ABCD1[19][D],ABCD1[19][A],ABCD1[15][B]) - ROOT2;

X1[12] = RotateRight(ABCD0[19][B],13) - ABCD0[15][B] -
  FunctionG(ABCD0[19][C],ABCD0[19][D],ABCD0[19][A]) - ROOT2;

X2[12] = RotateRight(ABCD1[19][B],13) - ABCD1[15][B] -
  FunctionG(ABCD1[19][C],ABCD1[19][D],ABCD1[19][A]) - ROOT2;

ABCD0[11][D] = RotateRight(ABCD0[15][D],7) -
  FunctionF(ABCD0[15][A],ABCD0[11][B],ABCD0[11][C]) - X1[13];

ABCD1[11][D] = ABCD0[11][D];

ABCD0[11][A] = RotateRight(ABCD0[15][A],3) -
  FunctionF(ABCD0[11][B],ABCD0[11][C],ABCD0[11][D]) - X1[12];

ABCD1[11][A] = ABCD0[11][A];

/* Find the message values that will satisfy the initial values */

CollisionFlag = 1;
Flag23 = 0;
DispFlag23 = 0;
```

```
    Flag27 = 0;
    DispFlag27 = 0;

    while(CollisionFlag != 0)
      {
Iteration++;

if(Flag23 == 0)
  {
     X1[1] = rand();
     X2[1] = X1[1];

     X1[5] = rand();
     X2[5] = X1[5];
  } else if (DispFlag23 == 0) {
     printf("X1 and X5 are fixed\n");
     DispFlag23 = 1;
  }


if(Flag27 == 0)
  {
     X1[2] = rand();
     X2[2] = X1[2];
  } else if (DispFlag27 == 0) {
     printf("X2 is fixed\n");
     DispFlag27 = 1;
  }

X1[3] = rand();
X2[3] = X1[3];

ABCD0[0][A] = A0;
ABCD0[0][B] = B0;
ABCD0[0][C] = C0;
ABCD0[0][D] = D0;

ABCD0[3][A] = Rotate((ABCD0[0][A] +
      FunctionF(ABCD0[0][B],ABCD0[0][C],ABCD0[0][D]) +
      X1[0]), 3);

ABCD0[3][D] = Rotate((ABCD0[0][D] +
      FunctionF(ABCD0[3][A],ABCD0[0][B],ABCD0[0][C]) +
      X1[1]), 7);

ABCD0[3][C] = Rotate((ABCD0[0][C] +
      FunctionF(ABCD0[3][D],ABCD0[3][A],ABCD0[0][B])
      + X1[2]), 11);

ABCD0[3][B] = Rotate((ABCD0[0][B] +
      FunctionF(ABCD0[3][C],ABCD0[3][D],ABCD0[3][A])
      + X1[3]), 19);

ABCD0[7][A] = Rotate((ABCD0[3][A] +
      FunctionF(ABCD0[3][B],ABCD0[3][C],ABCD0[3][D]) +
```

```
        X1[4]), 3);

ABCD0[7][D] = Rotate((ABCD0[3][D] +
        FunctionF(ABCD0[7][A],ABCD0[3][B],ABCD0[3][C]) +
        X1[5]), 7);

ABCD0[7][B] = -1;

ABCD0[7][C] = RotateRight(ABCD0[11][A],3) - ABCD0[7][A] - X1[8];

X1[6] = ( RotateRight(ABCD0[7][C], 11) - ABCD0[3][C] -
 FunctionF(ABCD0[7][D],ABCD0[7][A],ABCD0[3][B]));

X2[6] = X1[6];

X1[7] = ( RotateRight(ABCD0[7][B], 19) - ABCD0[3][B] -
 FunctionF(ABCD0[7][C],ABCD0[7][D],ABCD0[7][A]) );

X2[7] = X1[7];

TempInt = Rotate((ABCD0[7][A] +
  FunctionF(ABCD0[7][B],ABCD0[7][C],ABCD0[7][D]) +
  X1[8]), 3);

X1[9] = ( RotateRight(ABCD0[11][D], 7) - ABCD0[7][D] -
 FunctionF(ABCD0[11][A],ABCD0[7][B],ABCD0[7][C]));

X2[9] = X1[9];


X1[10] = (RotateRight(ABCD0[11][C], 11) - ABCD0[7][C] -
  FunctionF(ABCD0[11][D],ABCD0[11][A],ABCD0[7][B]));

X2[10] = X1[10];

X1[11] = (RotateRight(ABCD0[11][B], 11) - ABCD0[7][B] -
  FunctionF(ABCD0[11][C],ABCD0[11][D],ABCD0[11][A]));

X2[11] = X1[11];

ABCD0[23][A] = Rotate((ABCD0[19][A] +
       FunctionG(ABCD0[19][B],ABCD0[19][C],ABCD0[19][D])
        + X1[1] + ROOT2), GS1);

ABCD1[23][A] = Rotate((ABCD1[19][A] +
       FunctionG(ABCD1[19][B],ABCD1[19][C],ABCD1[19][D])
        + X2[1] + ROOT2), GS1);

Collision = ABCD0[23][A] - ABCD1[23][A];

if(Collision == 0)
  {

     ABCD0[23][D] = Rotate((ABCD0[19][D] +
      FunctionG(ABCD0[23][A],ABCD0[19][B],ABCD0[19][C])
```

```
        + X1[5] + ROOT2), GS2);

      ABCD1[23][D] = Rotate((ABCD1[19][D] +
       FunctionG(ABCD1[23][A],ABCD1[19][B],ABCD1[19][C])
        + X2[5] + ROOT2), GS2);

      Collision = ABCD0[23][D] - ABCD1[23][D];

  }

if(Collision == 0)
  {
     ABCD0[23][C] = Rotate((ABCD0[19][C] +
      FunctionG(ABCD0[23][D],ABCD0[23][A],ABCD0[19][B])
      + X1[9] + ROOT2), GS3);

     ABCD1[23][C] = Rotate((ABCD1[19][C] +
      FunctionG(ABCD1[23][D],ABCD1[23][A],ABCD1[19][B])
      + X2[9] + ROOT2), GS3);

     Collision = ABCD0[23][C] - ABCD1[23][C];
  }

if(Collision == -1*Rotate(1,14))
  {
     ABCD0[23][B] = Rotate((ABCD0[19][B] +
    FunctionG(ABCD0[23][C],ABCD0[23][D],ABCD0[23][A])
      + X1[13] + ROOT2), GS4);

     ABCD1[23][B] = Rotate((ABCD1[19][B] +
      FunctionG(ABCD1[23][C],ABCD1[23][D],ABCD1[23][A])
      + X2[13] + ROOT2), GS4);


     Collision = ABCD0[23][B] - ABCD1[23][B];

  }

/* Is next iteration satisfied */

if(Collision == Rotate(1,6))
  {
     Flag23 = 1;

     ABCD0[27][A] = Rotate((ABCD0[23][A] +
    FunctionG(ABCD0[23][B],ABCD0[23][C],ABCD0[23][D])
      + X1[2] + ROOT2), GS1);

     ABCD1[27][A] = Rotate((ABCD1[23][A] +
    FunctionG(ABCD1[23][B],ABCD1[23][C],ABCD1[23][D])
      + X2[2] + ROOT2), GS1);

     Collision = ABCD0[27][A] - ABCD1[27][A];
  }
```

```
if(Collision == 0)
  {

      ABCD0[27][D] = Rotate((ABCD0[23][D] +
       FunctionG(ABCD0[27][A],ABCD0[23][B],ABCD0[23][C])
        + X1[6] + ROOT2), GS2);

      ABCD1[27][D] = Rotate((ABCD1[23][D] +
       FunctionG(ABCD1[27][A],ABCD1[23][B],ABCD1[23][C])
        + X2[6] + ROOT2), GS2);

      Collision = ABCD0[27][D] - ABCD1[27][D];

  }

if(Collision == 0)
  {
      ABCD0[27][C] = Rotate((ABCD0[23][C] +
       FunctionG(ABCD0[27][D],ABCD0[27][A],ABCD0[23][B])
       + X1[10] + ROOT2), GS3);

      ABCD1[27][C] = Rotate((ABCD1[23][C] +
       FunctionG(ABCD1[27][D],ABCD1[27][A],ABCD1[23][B])
       + X2[10] + ROOT2), GS3);

      Collision = ABCD0[27][C] - ABCD1[27][C];
  }

if(Collision == -1*Rotate(1,23))
  {
      ABCD0[27][B] = Rotate((ABCD0[23][B] +
     FunctionG(ABCD0[27][C],ABCD0[27][D],ABCD0[27][A])
       + X1[14] + ROOT2), GS4);

      ABCD1[27][B] = Rotate((ABCD1[23][B] +
       FunctionG(ABCD1[27][C],ABCD1[27][D],ABCD1[27][A])
       + X2[14] + ROOT2), GS4);


      Collision = ABCD0[27][B] - ABCD1[27][B];

  }

/* Calculate steps 28-31 */

if(Collision == Rotate(1,19))
  {

      Flag23 = 1;

      ABCD0[31][A] = Rotate((ABCD0[27][A] +
    FunctionG(ABCD0[27][B],ABCD0[27][C],ABCD0[27][D])
       + X1[3] + ROOT2), GS1);

      ABCD1[31][A] = Rotate((ABCD1[27][A] +
```

```
      FunctionG(ABCD1[27][B],ABCD1[27][C],ABCD1[27][D])
         + X2[3] + ROOT2), GS1);

       Collision = ABCD0[31][A] - ABCD1[31][A];
   }

 if(Collision == 0)
   {

      ABCD0[31][D] = Rotate((ABCD0[27][D] +
       FunctionG(ABCD0[31][A],ABCD0[27][B],ABCD0[27][C])
        + X1[7] + ROOT2), GS2);

      ABCD1[31][D] = Rotate((ABCD1[27][D] +
       FunctionG(ABCD1[31][A],ABCD1[27][B],ABCD1[27][C])
        + X2[7] + ROOT2), GS2);

      Collision = ABCD0[31][D] - ABCD1[31][D];

   }

 if(Collision == 0)
   {
      ABCD0[31][C] = Rotate((ABCD0[27][C] +
       FunctionG(ABCD0[31][D],ABCD0[31][A],ABCD0[27][B])
       + X1[11] + ROOT2), GS3);

      ABCD1[31][C] = Rotate((ABCD1[27][C] +
       FunctionG(ABCD1[31][D],ABCD1[31][A],ABCD1[27][B])
       + X2[11] + ROOT2), GS3);

      Collision = ABCD0[31][C] - ABCD1[31][C];
   }

 if(Collision == -1)
   {
      ABCD0[31][B] = Rotate((ABCD0[27][B] +
     FunctionG(ABCD0[31][C],ABCD0[31][D],ABCD0[31][A])
       + X1[15] + ROOT2), GS4);

      ABCD1[31][B] = Rotate((ABCD1[27][B] +
       FunctionG(ABCD1[31][C],ABCD1[31][D],ABCD1[31][A])
       + X2[15] + ROOT2), GS4);


      Collision = ABCD0[31][B] - ABCD1[31][B];

   }

/* Calculate steps 32-35 */

 if(Collision == 1)
   {

      ABCD0[35][A] = Rotate((ABCD0[31][A] +
```

```
      FunctionH(ABCD0[31][B],ABCD0[31][C],ABCD0[31][D])
         + X1[0] + ROOT3), HS1);

       ABCD1[35][A] = Rotate((ABCD1[31][A] +
      FunctionH(ABCD1[31][B],ABCD1[31][C],ABCD1[31][D])
         + X2[0] + ROOT3), HS1);

       Collision = ABCD0[35][A] - ABCD1[35][A];

   }

if(Collision == 0)
   {

       ABCD0[35][D] = Rotate((ABCD0[31][D] +
        FunctionH(ABCD0[35][A],ABCD0[31][B],ABCD0[31][C])
        + X1[8] + ROOT3), HS2);

       ABCD1[35][D] = Rotate((ABCD1[31][D] +
        FunctionH(ABCD1[35][A],ABCD1[31][B],ABCD1[31][C])
        + X2[8] + ROOT3), HS2);

       Collision = ABCD0[35][D] - ABCD1[35][D];

   }

if(Collision == 0)
   {
       ABCD0[35][C] = Rotate((ABCD0[31][C] +
        FunctionH(ABCD0[35][D],ABCD0[35][A],ABCD0[31][B])
        + X1[4] + ROOT3), HS3);

       ABCD1[35][C] = Rotate((ABCD1[31][C] +
        FunctionH(ABCD1[35][D],ABCD1[35][A],ABCD1[31][B])
        + X2[4] + ROOT3), HS3);

       Collision = ABCD0[35][C] - ABCD1[35][C];
   }

if(Collision == 0)
   {
       ABCD0[35][B] = Rotate((ABCD0[31][B] +
      FunctionH(ABCD0[35][C],ABCD0[35][D],ABCD0[35][A])
        + X1[12] + ROOT3), HS4);

       ABCD1[35][B] = Rotate((ABCD1[31][B] +
        FunctionH(ABCD1[35][C],ABCD1[35][D],ABCD1[35][A])
        + X2[12] + ROOT3), HS4);


       Collision = ABCD0[35][B] - ABCD1[35][B];

       if(Collision == 0)
         {
   CollisionFlag = 0;
```

```
        }

    }


      }

    printf("Iterations: %u\n", Iteration);

    printf("\tX1\t\t X2\n");
    for(i=0; i<16; i++)
      {
printf("(%u)\t%.8X\t %.8X\n", i, X1[i], X2[i]);
      }

    /* Write results to file */

    fp = fopen("X1.dat", "wb");

    if(!fp)
      {
printf("Error opening file\n");
fclose(fp);
exit(1);
      }

    /* Write the first message to file */

    for(i=0; i<16; i++)
      {
TempInt = X1[i];
fwrite(&TempInt, sizeof(unsigned int), 1, fp);
      }

    fclose(fp);

    fp = fopen("X2.dat", "wb");

    if(!fp)
      {
printf("Error opening file\n");
fclose(fp);
exit(1);
      }

    /* Write the second message to file */

    for(i=0; i<16; i++)
      {
TempInt = X2[i];
fwrite(&TempInt, sizeof(unsigned int), 1, fp);
      }

    fclose(fp);
```

```
    /* Test to see if a collision has occured */

    md4(2,"X1.dat");
    md4(2,"X2.dat");


    return(0);
}
```

# APPENDIX D: IMPLEMENTATION: MD5

This Appendix contains the C source code of an implementation of MD5.

```
#define A0 0x67452301
#define B0 0xefcdab89
#define C0 0x98badcfe
#define D0 0x10325476

#define FS1 7
#define FS2 12
#define FS3 17
#define FS4 22

#define GS1 5
#define GS2 9
#define GS3 14
#define GS4 20

#define HS1 4
#define HS2 11
#define HS3 16
#define HS4 23

#define IS1 6
#define IS2 10
#define IS3 15
#define IS4 21

int T[64] = {0xd76aa478, 0xe8c7b756, 0x242070db, 0xc1bdceee, 0xf57c0faf,
     0x4787c62a, 0xa8304613, 0xfd469501, 0x698098d8, 0x8b44f7af, 0xffff5bb1,
     0x895cd7be, 0x6b901122, 0xfd987193, 0xa679438e, 0x49b40821, 0xf61e2562,
     0xc040b340, 0x265e5a51, 0xe9b6c7aa, 0xd62f105d, 0x02441453, 0xd8a1e681,
     0xe7d3fbc8, 0x21e1cde6, 0xc33707d6, 0xf4d50d87, 0x455a14ed, 0xa9e3e905,
     0xfcefa3f8, 0x676f02d9, 0x8d2a4c8a, 0xfffa3942, 0x8771f681, 0x6d9d6122,
     0xfde5380c, 0xa4beea44, 0x4bdecfa9, 0xf6bb4b60, 0xbebfbc70, 0x289b7ec6,
     0xeaa127fa, 0xd4ef3085, 0x04881d05, 0xd9d4d039, 0xe6db99e5, 0x1fa27cf8,
     0xc4ac5665, 0xf4292244, 0x432aff97, 0xab9423a7, 0xfc93a039, 0x655b59c3,
     0x8f0ccc92, 0xffeff47d, 0x85845dd1, 0x6fa87e4f, 0xfe2ce6e0, 0xa3014314,
     0x4e0811a1, 0xf7537e82, 0xbd3af235, 0x2ad7d2bb, 0xeb86d391};

char *PadBit(char *Message, unsigned int *Length);
void char_2_int_array(char *Message, unsigned int *MessageInt, unsigned in-
t Length);
unsigned int *AppendLength(unsigned int *MessageInt,
   unsigned int *AppendLength, unsigned int OrgLen);
unsigned int MD5_F(unsigned int X, unsigned int Y, unsigned int Z);
unsigned int MD5_G(unsigned int X, unsigned int Y, unsigned int Z);
unsigned int MD5_H(unsigned int X, unsigned int Y, unsigned int Z);
unsigned int MD5_I(unsigned int X, unsigned int Y, unsigned int Z);
void InitMD5Buf(unsigned int *A, unsigned int *B,
unsigned int *C, unsigned int *D);
void SaveMD5Parms(unsigned int A, unsigned int B, unsigned int C,
  unsigned int D, unsigned int *AA, unsigned int *BB,
  unsigned int *CC, unsigned int *DD);
void Round1(unsigned int *A, unsigned int *B, unsigned int *C, unsigned in-
t *D,
    unsigned int *X);
```

```
void Round2(unsigned int *A, unsigned int *B, unsigned int *C, unsigned in-
t *D,
    unsigned int *X);
void Round3(unsigned int *A, unsigned int *B, unsigned int *C, unsigned in-
t *D,
    unsigned int *X);
void Round4(unsigned int *A, unsigned int *B, unsigned int *C, unsigned in-
t *D,
    unsigned int *X);
void Update(unsigned int *A, unsigned int *B, unsigned int *C, unsigned in-
t *D,
    unsigned int AA, unsigned int BB, unsigned int CC,
    unsigned int DD);
unsigned int RotateLeft(unsigned int X, unsigned int s);
void PrintSignature(unsigned int *A);
void PrintReverse(unsigned int X);
unsigned int Reverse(unsigned int X);
void MD5(char *Message, unsigned int Length, unsigned int *Hash);

void MD5(char *Message, unsigned int Length, unsigned int *Hash)
{
   int i,j;
   int PaddedSize, AppendSize;
   unsigned int *MessageInt, *X;
   unsigned int A, B, C, D, AA, BB, CC, DD;

   PaddedSize = Length;

   X =(unsigned int *)calloc(16, sizeof(unsigned int));
   if(X == NULL)
     {
printf("Error Allocating Memory\n");
exit(1);
     }

   /* Determine the number of Padding bytes required */
   Message = PadBit(Message, &PaddedSize);

   MessageInt =(unsigned int *)calloc(PaddedSize/(sizeof(unsigned int)), size-
of(unsigned int));

   char_2_int_array(Message, MessageInt, PaddedSize);
   AppendSize = PaddedSize;

   for(i=0; i<PaddedSize/(sizeof(unsigned int)); i++)
     {
MessageInt[i] = Reverse(MessageInt[i]);
     }

   MessageInt = AppendLength(MessageInt, &AppendSize, Length);

   InitMD5Buf(&A,&B,&C,&D);

   for(i=0; i<(AppendSize*8)/512; i++)
     {
```

```
SaveMD5Parms(A, B, C, D, &AA, &BB, &CC, &DD);

for(j=0; j<16; j++)
  {
     X[j] = MessageInt[i*16+j];
  }

Round1(&A, &B, &C, &D, X);
Round2(&A, &B, &C, &D, X);
Round3(&A, &B, &C, &D, X);
Round4(&A, &B, &C, &D, X);
Update(&A, &B, &C, &D, AA, BB, CC, DD);
     }

   Hash[0] = A;
   Hash[1] = B;
   Hash[2] = C;
   Hash[3] = D;

   free(X);
   free(MessageInt);
}

/* This routine performs the bit padding as requied for the MD4 algorithm */
char *PadBit(char *Message, unsigned int *Length)
{
   char *TempPtr;
   unsigned int i,j;
   unsigned int FileLen, FileBits, PadBits, PadBytes;
   unsigned char temp;

   FileBits = *Length*8*sizeof(unsigned char);

   /* Compute the number of bits needed for padding */
   PadBits = abs( (448 - FileBits) % 512 );
   if(PadBits == 0)
     {
PadBits = 512*(FileBits/512) + 512;
     }

   PadBytes = PadBits/8;

   TempPtr = (char *)realloc(Message, (*Length+PadBytes));
   if(TempPtr == NULL)
     {
printf("Error Reallocating Memory\n");
exit(1);
     }

   /* Pad bit "1" */

   TempPtr[*Length] = 0x80;
   /* Pad zero bits */
   for(i=1; i<PadBytes; i++)
     {
```

```
TempPtr[*Length+i] = 0x00;
     }

   Message = (char *)realloc(TempPtr, (*Length+PadBytes));
   if(Message == NULL)
     {
printf("Error Reallocating Memory\n");
exit(1);
     }

   *Length = (PadBytes+(*Length));

   return(Message);
}

unsigned int *AppendLength(unsigned int *MessageInt,
   unsigned int *AppendLength, unsigned int OrgLen)
{
   int i;
   unsigned int *TempPtr;

   *AppendLength = *AppendLength+2*sizeof(unsigned int);

   TempPtr = (unsigned int *)realloc(MessageInt, *AppendLength);
   if(TempPtr == NULL)
     {
printf("Error Reallocating Memory\n");
exit(1);
     }

   TempPtr[*AppendLength/sizeof(unsigned int)-2] = OrgLen*8;
   TempPtr[*AppendLength/sizeof(unsigned int)-1] = 0;

   MessageInt = (unsigned int *)realloc(TempPtr, *AppendLength);
   if(MessageInt == NULL)
     {
printf("Error Reallocating Memory\n");
exit(1);
     }

   return(MessageInt);
}

void char_2_int_array(char *Message, unsigned int *MessageInt,
      unsigned int Length)
{
   int i, j;

   for(i=0; i<(Length/sizeof(unsigned int)); i++)
     {
for(j=0; j<sizeof(unsigned int); j++)
  {
     MessageInt[i] = (MessageInt[i] << 8*sizeof(char)) |
       (Message[i*sizeof(unsigned int) + j] & 0x000000ff);
  }
```

```
    }

}

unsigned int MD5_F(unsigned int X, unsigned int Y, unsigned int Z)
{
   return((X&Y) | (~X&Z));
}

unsigned int MD5_G(unsigned int X, unsigned int Y, unsigned int Z)
{
   return((X & Z) | (Y & (~Z)));
}

unsigned int MD5_H(unsigned int X, unsigned int Y, unsigned int Z)
{
   return(X ^ Y ^ Z);
}

unsigned int MD5_I(unsigned int X, unsigned int Y, unsigned int Z)
{
   return(Y ^ (X | (~Z)));
}

void InitMD5Buf(unsigned int *A, unsigned int *B,
unsigned int *C, unsigned int *D)
{
   *A = A0;
   *B = B0;
   *C = C0;
   *D = D0;
}

void SaveMD5Parms(unsigned int A, unsigned int B, unsigned int C,
       unsigned int D, unsigned int *AA, unsigned int *BB,
       unsigned int *CC, unsigned int *DD)
{
   *AA = A;
   *BB = B;
   *CC = C;
   *DD = D;
}

void Round1(unsigned int *A, unsigned int *B, unsigned int *C, unsigned in-
t *D,
    unsigned int *X)
{
   *A = *B + RotateLeft((*A + MD5_F(*B,*C,*D) + X[0] + T[0]), FS1);
   *D = *A + RotateLeft((*D + MD5_F(*A,*B,*C) + X[1] + T[1]), FS2);
   *C = *D + RotateLeft((*C + MD5_F(*D,*A,*B) + X[2] + T[2]), FS3);
   *B = *C + RotateLeft((*B + MD5_F(*C,*D,*A) + X[3] + T[3]), FS4);

   *A = *B + RotateLeft((*A + MD5_F(*B,*C,*D) + X[4] + T[4]), FS1);
   *D = *A + RotateLeft((*D + MD5_F(*A,*B,*C) + X[5] + T[5]), FS2);
   *C = *D + RotateLeft((*C + MD5_F(*D,*A,*B) + X[6] + T[6]), FS3);
```

```
    *B = *C + RotateLeft((*B + MD5_F(*C,*D,*A) + X[7] + T[7]), FS4);

    *A = *B + RotateLeft((*A + MD5_F(*B,*C,*D) + X[8] + T[8]), FS1);
    *D = *A + RotateLeft((*D + MD5_F(*A,*B,*C) + X[9] + T[9]), FS2);
    *C = *D + RotateLeft((*C + MD5_F(*D,*A,*B) + X[10] + T[10]), FS3);
    *B = *C + RotateLeft((*B + MD5_F(*C,*D,*A) + X[11] + T[11]), FS4);

    *A = *B + RotateLeft((*A + MD5_F(*B,*C,*D) + X[12] + T[12]), FS1);
    *D = *A + RotateLeft((*D + MD5_F(*A,*B,*C) + X[13] + T[13]), FS2);
    *C = *D + RotateLeft((*C + MD5_F(*D,*A,*B) + X[14] + T[14]), FS3);
    *B = *C + RotateLeft((*B + MD5_F(*C,*D,*A) + X[15] + T[15]), FS4);
}

void Round2(unsigned int *A, unsigned int *B, unsigned int *C, unsigned in-
t *D,
     unsigned int *X)
{
    *A = *B + RotateLeft((*A + MD5_G(*B,*C,*D) + X[1] + T[16]), GS1);
    *D = *A + RotateLeft((*D + MD5_G(*A,*B,*C) + X[6] + T[17]), GS2);
    *C = *D + RotateLeft((*C + MD5_G(*D,*A,*B) + X[11] + T[18]), GS3);
    *B = *C + RotateLeft((*B + MD5_G(*C,*D,*A) + X[0] + T[19]), GS4);

    *A = *B + RotateLeft((*A + MD5_G(*B,*C,*D) + X[5] + T[20]), GS1);
    *D = *A + RotateLeft((*D + MD5_G(*A,*B,*C) + X[10] + T[21]), GS2);
    *C = *D + RotateLeft((*C + MD5_G(*D,*A,*B) + X[15] + T[22]), GS3);
    *B = *C + RotateLeft((*B + MD5_G(*C,*D,*A) + X[4] + T[23]), GS4);

    *A = *B + RotateLeft((*A + MD5_G(*B,*C,*D) + X[9] + T[24]), GS1);
    *D = *A + RotateLeft((*D + MD5_G(*A,*B,*C) + X[14] + T[25]), GS2);
    *C = *D + RotateLeft((*C + MD5_G(*D,*A,*B) + X[3] + T[26]), GS3);
    *B = *C + RotateLeft((*B + MD5_G(*C,*D,*A) + X[8] + T[27]), GS4);

    *A = *B + RotateLeft((*A + MD5_G(*B,*C,*D) + X[13] + T[28]), GS1);
    *D = *A + RotateLeft((*D + MD5_G(*A,*B,*C) + X[2] + T[29]), GS2);
    *C = *D + RotateLeft((*C + MD5_G(*D,*A,*B) + X[7] + T[30]), GS3);
    *B = *C + RotateLeft((*B + MD5_G(*C,*D,*A) + X[12] + T[31]), GS4);
}

void Round3(unsigned int *A, unsigned int *B, unsigned int *C, unsigned in-
t *D,
     unsigned int *X)
{
    *A = *B + RotateLeft((*A + MD5_H(*B,*C,*D) + X[5] + T[32]), HS1);
    *D = *A + RotateLeft((*D + MD5_H(*A,*B,*C) + X[8] + T[33]), HS2);
    *C = *D + RotateLeft((*C + MD5_H(*D,*A,*B) + X[11] + T[34]), HS3);
    *B = *C + RotateLeft((*B + MD5_H(*C,*D,*A) + X[14] + T[35]), HS4);

    *A = *B + RotateLeft((*A + MD5_H(*B,*C,*D) + X[1] + T[36]), HS1);
    *D = *A + RotateLeft((*D + MD5_H(*A,*B,*C) + X[4] + T[37]), HS2);
    *C = *D + RotateLeft((*C + MD5_H(*D,*A,*B) + X[7] + T[38]), HS3);
    *B = *C + RotateLeft((*B + MD5_H(*C,*D,*A) + X[10] + T[39]), HS4);

    *A = *B + RotateLeft((*A + MD5_H(*B,*C,*D) + X[13] + T[40]), HS1);
    *D = *A + RotateLeft((*D + MD5_H(*A,*B,*C) + X[0] + T[41]), HS2);
    *C = *D + RotateLeft((*C + MD5_H(*D,*A,*B) + X[3] + T[42]), HS3);
```

```
    *B = *C + RotateLeft((*B + MD5_H(*C,*D,*A) + X[6] + T[43]), HS4);

    *A = *B + RotateLeft((*A + MD5_H(*B,*C,*D) + X[9] + T[44]), HS1);
    *D = *A + RotateLeft((*D + MD5_H(*A,*B,*C) + X[12] + T[45]), HS2);
    *C = *D + RotateLeft((*C + MD5_H(*D,*A,*B) + X[15] + T[46]), HS3);
    *B = *C + RotateLeft((*B + MD5_H(*C,*D,*A) + X[2] + T[47]), HS4);
}

void Round4(unsigned int *A, unsigned int *B, unsigned int *C, unsigned in-
t *D,
     unsigned int *X)
{
    *A = *B + RotateLeft((*A + MD5_I(*B,*C,*D) + X[0] + T[48]), IS1);
    *D = *A + RotateLeft((*D + MD5_I(*A,*B,*C) + X[7] + T[49]), IS2);
    *C = *D + RotateLeft((*C + MD5_I(*D,*A,*B) + X[14] + T[50]), IS3);
    *B = *C + RotateLeft((*B + MD5_I(*C,*D,*A) + X[5] + T[51]), IS4);

    *A = *B + RotateLeft((*A + MD5_I(*B,*C,*D) + X[12] + T[52]), IS1);
    *D = *A + RotateLeft((*D + MD5_I(*A,*B,*C) + X[3] + T[53]), IS2);
    *C = *D + RotateLeft((*C + MD5_I(*D,*A,*B) + X[10] + T[54]), IS3);
    *B = *C + RotateLeft((*B + MD5_I(*C,*D,*A) + X[1] + T[55]), IS4);

    *A = *B + RotateLeft((*A + MD5_I(*B,*C,*D) + X[8] + T[56]), IS1);
    *D = *A + RotateLeft((*D + MD5_I(*A,*B,*C) + X[15] + T[57]), IS2);
    *C = *D + RotateLeft((*C + MD5_I(*D,*A,*B) + X[6] + T[58]), IS3);
    *B = *C + RotateLeft((*B + MD5_I(*C,*D,*A) + X[13] + T[59]), IS4);

    *A = *B + RotateLeft((*A + MD5_I(*B,*C,*D) + X[4] + T[60]), IS1);
    *D = *A + RotateLeft((*D + MD5_I(*A,*B,*C) + X[11] + T[61]), IS2);
    *C = *D + RotateLeft((*C + MD5_I(*D,*A,*B) + X[2] + T[62]), IS3);
    *B = *C + RotateLeft((*B + MD5_I(*C,*D,*A) + X[9] + T[63]), IS4);

    /*printf("%8.8X %8.8X %8.8X %8.8X\n", *A, *B, *C, *D);*/
}

void Update(unsigned int *A, unsigned int *B, unsigned int *C, unsigned in-
t *D,
     unsigned int AA, unsigned int BB, unsigned int CC,
     unsigned int DD)
{
    *A = *A + AA;
    *B = *B + BB;
    *C = *C + CC;
    *D = *D + DD;
}

unsigned int RotateLeft(unsigned int X, unsigned int s)
{
    unsigned int temp;

    temp = X;
    X = (X << s) | (temp >> (32-s));
    return(X);
}
```

```
void PrintSignature(unsigned int *A)
{
   int i;

   for(i=0; i<4; i++)
     {
PrintReverse(A[i]);
     }
}

void PrintReverse(unsigned int X)
{
   int i;

   for(i=0; i<4; i++)
     {
printf("%.2x", X & 0x000000ff);
X = X >> 8;
     }

}

unsigned int Reverse(unsigned int X)
{
   int i;
   unsigned int Y;

   Y=0;
   for(i=0; i<4; i++)
     {
Y = (Y<<8) | X & 0x000000ff;
X = X >> 8;
     }
   return(Y);
}

#undef A0
#undef B0
#undef C0
#undef D0

#undef FS1
#undef FS2
#undef FS3
#undef FS4

#undef GS1
#undef GS2
#undef GS3
#undef GS4

#undef HS1
#undef HS2
#undef HS3
#undef HS4
```

```
#undef IS1
#undef IS2
#undef IS3
#undef IS4
```