RIPE integrity primitives Part I
Final report of RACE 1040

RIPE Consortium

Computer Science/Department of Algorithmics and Architecture

Report CS-R9324 April 1993

CWI is the National Research Institute for Mathematics and Computer Science. CWI is part of the Stichting Mathematisch Centrum (SMC), the Dutch foundation for promotion of mathematics and computer science and their applications. SMC is sponsored by the Netherlands Organization for Scientific Research (NWO). CWI is a member of ERCIM, the European Research Consortium for Informatics and Mathematics.

P.O. Box 4079, 1009 AB Amsterdam (NL)
Kruislaan 413, 1098 SJ Amsterdam (NL)
Telephone +31 20 592 9333
Telefax +31 20 592 4199

RIPE Integrity Primitives
Part I

Final Report of RACE Integrity Primitives Evaluation (R1040)

B. den Boer, *Philips Crypto B.V., Eindhoven (NL)*
J.P. Boly, *PTT Research, Leidschendam (NL)*
A. Bosselaers, *ESAT Lab, K.U. Leuven (B)*
J. Brandt, *Aarhus Universitet, Århus (DK)*
D. Chaum, *CWI, Amsterdam (NL)*
I. Damgård, *Aarhus Universitet, Århus (DK)*
M. Dichtl, *Siemens AG, München (D)*
W. Fumy, *Siemens AG, Erlangen (D)*
M. van der Ham, *CWI, Amsterdam (NL)*
C.J.A. Jansen, *Philips Crypto B.V., Eindhoven (NL)*
P. Landrock, *Aarhus Universitet, Århus (DK)*
B. Preneel, *ESAT Lab, K.U. Leuven (B)*
G. Roelofsen, *PTT Research, Leidschendam (NL)*
P. de Rooij, *PTT Research, Leidschendam (NL)*
J. Vandewalle, *ESAT Lab, K.U. Leuven (B)*

Abstract

This is a manual intended for those seeking to secure information systems by applying modern cryptography. It represents the successful attainment of goals by RIPE (RACE Integrity Primitives Evaluation), a 350 man-month project funded by the Commission of the European Communities. The recommended portfolio of integrity primitives, which is the main product of the project, forms the heart of this volume.

By integrity, we mean the kinds of security that can be achieved through cryptography, apart from keeping messages secret. Thus included are ways to ensure that stored or communicated data is not illicitly modified, that parties exchanging messages are actually present, and that "signed" electronic messages can be recognised as authentic by anyone.

Of particular concern to the project were the high-speed requirements of broad-band communication. But the project also aimed for completeness in its recommendations. As a result, the portfolio contains primitives, i.e. building blocks, that can meet most of today's perceived needs for integrity.

AMS Subject Classification (1991): 94A60
CR Subject Classification (1991): D.4.6
Keywords & Phrases: Integrity Primitives, Security Services, Integrity Mechanisms, Data Origin Authentication, Entity Authentication, Access Control, Data Integrity, Non-repudiation, Signature, Key Exchange.
Note: The work described in this report is the result of a research project carried out during the period 1989 to June 1992. While the project received support under the EC RACE programme the results should not be interpreted as given a view on the Community policy in this area.

Report CS-R9324
ISSN 0169-118X
CWI
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

Contents

Part I

Introduction and Background

Contents

1 Introduction

This is a manual intended for those seeking to secure information systems by applying modern cryptography. It represents the successful attainment of goals by RIPE (RACE Integrity Primitives Evaluation), a 350 man-month project funded by the Commission of the European Communities. The recommended portfolio of integrity primitives, which is the main product of the project, forms the heart of this volume.

By integrity, we mean the kinds of security that can be achieved through cryptography, apart from keeping messages secret. Thus, included are ways to ensure that stored or communicated data is not illicitly modified, that parties exchanging messages are actually present, and that "signed" electronic messages can be recognized as authentic by anyone.

Of particular concern to the project were the high-speed requirements of broad-band communication. But the project also aimed for completeness in its recommendations. As a result, the portfolio contains primitives, i.e. building blocks, that can meet most of today's perceived needs for integrity.

1.1 Perspective on the project results

Six leading European research groups in the field made up the RIPE consortium; however, much of the input was drawn from experts around the world who responded to two widely circulated calls for submissions. These responses were then extensively evaluated according to a work plan that involved multiple stages of review for each submission and independent verification of decisions. We were fortunate that those submissions remaining, after some adjustments, gave essentially the same breadth of coverage as the original set.

Most of the project's effort is regrettably not directly reflected in this report. The majority of submissions had to be, after substantial efforts in many cases, rejected because of weaknesses uncovered during the project.

In the cryptographic world today there are few absolute guarantees, however. Primitives with provable security such as the so called one-time pad are rare and of limited applicabillity. A few others have security that can be reduced to certain established assumptions, as with many minimum disclosure protocols. But security of the overwhelming majority of primitives is based simply on the lack of any known successful attack. In this respect our portfolio contains a representative sampling, which means that ongoing monitoring will continue to be necessary for most of the primitives we recommend.

The project was of course confronted with the question of what kinds of primitives are really needed. Unable to find an acceptable answer elsewhere, we developed a taxonomy of primitives which we believe is well suited to currently perceived needs for integrity. But as integrity techniques continues to be more widely applied, the need for more sophisiticated capabilities can be expected to grow.

1.2 Relationship to standardization

Standardization of integrity primitives has proven to be rather difficult in practice. Though substantial efforts have been made over the last decade, the specificity of the results is far less than what we propose here. And specificity is key to cryptographic standards, since compatibility and interoperability of systems are major requirements and are often the main motivation for standardization in the first place.

Part of the reason our efforts have gone further is that we could apply substantial research effort which usually is not available to support standardization activities. Also, we have had the benefit of relevant standardization proposals at various stages.

We expect that parts of our portfolio will be serious candidates for international standardization in ISO. As a European project, however, we are particularly focused on standardization activities such as ETSI and CEN, as well as de facto standardization that may result from adoption within CEC DG XIII funded programs such as RACE II.

1.3 Organization of this report

The remainder of Part I describes the background, structure, and actual experience of the RIPE project itself.

Part II is intended as a survey of the kinds of problems generally encountered in information security and the particular kinds of solutions offered by integrity primitives. The approach is structured by our taxonomy of primitives. For the newcomer to the field, references to introductory literature are provided.

Each chapter of Part III covers a different recommended integrity primitive. Although the description of individual primitives is self- contained, they all follow a common format. They each have a section on definitions, in the style of standardization documents; a description of the primitive itself; recommended ways to use it; indications of the claimed properties as well as the extent to which they have been verified; performance estimates, and software implementation guidelines and test values.

2 The RIPE Project

2.1 Integrated Broad-band Communication for Europe

The European Community plans to set up a unified European market of about 300 million customers by 1993. In view of this market Integrated Broad-band Communication (IBC) has been planned for commercial use in 1996. This communication network will provide high-speed channels and will support a broad spectrum of services. In order to pave the way towards commercial use of IBC, the Commission of the European Communities has launched the RACE program (Research and Development in Advanced Communications Technologies in Europe) [RAC88]. Under this program pre-competitive and pre-normative work is going on. It is clear that the majority of

the services offered as well as the management of the network are crucially dependent on the use of cryptographic techniques for security.

Within RACE, the RIPE project (RACE Integrity Primitives Evaluation) had the goal of putting forward a portfolio of techniques to meet the anticipated security requirements of IBC. (See [FVCJ92] for RIPE seen from an IBC perspective.)

The members of the RIPE project are: the Center for Mathematics and Computer Science, Amsterdam (prime contractor); Siemens AG; Philips Crypto BV; Royal PTT Nederland NV, PTT Research; Katholieke Universiteit Leuven; and Arhus Universitet.

2.2 An open call for integrity primitives

The project's motivation was the unique opportunity to attain consensus on openly available integrity primitives. In order to achieve wide acceptance for a collection of algorithms, the RIPE consortium decided to disseminate an open call.

The scope of the project and the evaluation procedure were fixed after having reached consensus with the main parties involved. The scope includes any digital integrity primitive, except for those offering data confidentiality. (It should be noted, however, that in some documents, [ISO83] e.g., integrity is used in a very much narrower sense than we use it.)

In response to the first call—which was circulated in a mailing of around 1250 brochures, announced by presentations at Eurocrypt'89 and Crypto'89 [VCFJ89], published in the Journal of Cryptology and the IACR Newsletter—fifteen submissions were received. Most common types of primitives were represented, but three additional primitives were invited for more comprehensive coverage. In the end, many well known primitives were submitted as well as proprietary ones from major suppliers.

From the eighteen submissions, ten came from academic submitters and other eight from industry. The division over different countries was as follows: West Germany 5; U.S.A. 4; Denmark 3; Canada and Japan 2; Belgium and Australia 1. In October 1989, many of the submitters attended special meetings aimed at clarifying the submissions.

2.3 Evaluation results

The evaluation was carried out following a carefully designed procedure. The submissions were evaluated with respect to three aspects: functionality, modes of use, and performance. The evaluation comprised computer simulation, statistical verification and analysis of mathematical structures, particularly to verify the integrity properties. Because of the limited resources and time period, it was decided that if any flaw was identified, the submitter would not be allowed to patch the flaw, thus preventing moving targets.

Five submissions already had to be rejected in a preliminary screening. After the main phase of the evaluation, and after taking into account deficiencies implied by work done in the cryptographic community, seven submissions remained.

These submissions showed significant potential, but each required modification and/or further specification by the submitters. Five of the seven had minor functional

problems. In most cases, it was clear how the problems could be avoided. Further evaluation was postponed, in accordance with the policy of not allowing modifications by submitters during evaluation. Permission was obtained from six submitters to publish the problems we had uncovered.

2.4 Second call for integrity primitives

At the inception of the project, it was already foreseen that some first round submissions would require adjustment, but a full second call was planned. It was circulated in essentially the same ways as the first call[PCFJ91], and resulted in 14 submissions all told. About 7 of these could be considered improved versions of submissions from the first call.

Of the fourteen submissions, nine came from academic submitters and the other 5 from industry. The division over different countries was as follows: Germany 6; U.S.A. 3; Belgium 2; Canada, Denmark, and the Netherlands 1. In June 1991 meetings with submitters were again held.

Essentially the same evaluation techniques were used as for the first call. The final seven primitives resulting comprise Part III of this report. Five of them were actual submissions, and two were adapted from the literature.

3 Conclusion

The RIPE project carried out its planned acquisition and evaluation of integrity primitives. It has been able to put forward a comprehensive yet carefully evaluated and specified portfolio of integrity primitives.

References

[FVCJ92] W. Fumy, J. Vandewalle, D. Chaum, C.J.A. Jansen, P. Landrock and G. Roelofsen, "Integrity Primitives for IBC," *Proceedings of IWACA '92-International Workshop for Advanced Communication Architecture*, Munich, Germany, 1992.

[ISO83] ISO International Standard 7498, Open Systems Interconnection: Basic Reference Model, 1983.

[PCFJ91] B. Preneel, D. Chaum, W. Fumy, C.J.A. Jansen, P. Landrock and G. Roelofsen, "Race Integrity Primitives Evaluation (RIPE): A Status Report," in: *Advances in Cryptology - EUROCRYPT'91*, D.W. Davies ed., Lecture Notes in Computer Science no. 547, Springer-Verlag, Berlin-Heidelberg-New York, pp. 547- 551, 1992.

[VCFJ89] J. Vandewalle, D. Chaum, W. Fumy, C.J.A. Jansen, P. Landrock and G. Roelofsen, "A European call for cryptographic algorithms: RIPE (RACE

Integrity Primitives Evaluation)," in: Advances of Cryptology - EUROCRYPT'89, J.-J. Quisquater and J. Vandewalle eds., Lecture Notes in Computer Science no. 434, Springer-Verlag, Berlin-Heidelberg-New York, pp. 267- 271, 1990.

[RAC88] *RACE Workplan*, Commission of the European Communities, 1988, Rue de la Loi 200, B-1049, Brussels, Belgium.

Part II

Integrity Concepts

Contents

1 Introduction

Information integrity is essential for many envisaged applications of broadband communication. In former days, the protection of information was mainly an issue of physical security and trust. The protection of authenticity relied on the difficulty of forging certain documents and/or signatures. In the electronic age, letters, contracts and other documents are replaced by sequences of bits, but the demand for authenticity and integrity existing in the 'old' world has in no way been diminished. On the contrary, with open and untrusted networks, it is relatively easy to commit all kinds of frauds unless precautions are taken [DaPr89].

There are several types of security measures. Security measures based on cryptographic techniques can provide efficient and flexible logical protection of information, which yields crucial advantages for most applications. To achieve a high level of communications security in a network, cryptographic mechanisms are to be employed. Encipherment e.g. supports data confidentiality, whereas cryptographic integrity check values can be used to protect data integrity. The use of cryptographic methods can prevent wiretapping, masquerading, and modification attacks, and does additionally allow for some access control policy to be implemented [PoKl89], [VoKe85].

The security measures employed for a specific system of course have to meet the user requirements. The set of those requirements is growing with the number of electronic communications facilities. However, a substantial part can be met by a fixed set of security services as identified in [ISO88].

The aim of this chapter is to establish a relationship between security services, security mechanisms and integrity primitives. Speaking in general terms, at the user end security services are offered. These security services are provided by security protocols or mechanisms, which are built up from basic building blocks called security primitives.

The RIPE project has selected a set of integrity primitives, the ingredients needed to solve a wide range of integrity problems. These techniques are concerned with protecting against non-trusted or potentially malicious disruption of the integrity in information systems. Some illustrative example uses of integrity primitives are as follows:

- *Accessible password files*: a publicly accessible 'password' list can allow anyone to check whether a particular password is allowed by the list, but nobody to discover what any of the passwords actually are.

- *Fingerprinted data*: a small fingerprint can be produced of an arbitrarily large amount of data when it is stored, so that when the data is later retrieved, the fingerprint can be checked and no modification of the data will escape detection.

- *Authenticated messages*: a message can be communicated through a hostile environment, in such a way that the intended receiver can be sure that it has not been modified and was sent by the intended sender.

- *Identification protocols*: one party can verify that a certain other party is really at the other end of the line during a conversation.
- *Public key digital signatures*: a message can be digitally signed so that anyone can verify its signature and even convince a judge that it really was signed by a particular party.

The following section details the concept of security services and describes how security mechanisms can be used to provide those services. A taxonomy of integrity primitives is given in Section 3. This allows security mechanisms to be built up from primitives as described in Section 4. General references for further reading are [Bra88, DaPr89, FuRi88, MeMa82].

2 Security Services

Security measures for information systems can be realized in various ways. In particular, there are many options for the integration of security services into a communications architecture. A system *Security Architecture* constitutes an overall security blueprint for a system. It describes security services and their interrelationships, and it shows how the security services map onto the system architecture.

In order to extend the field of application of the Basic Reference Model for Open System Interconnection [ISO83], ISO (the *International Organization for Standardization*) has identified a set of security services and possibilities of integrating them into the seven layers of the OSI architecture. The *OSI Security Architecture* [ISO88] recognizes five primary security services: authentication, access control, confidentiality, integrity, and non-repudiation. These services form the basic building blocks for other security architectures as well.

- *Data Origin Authentication* provides corroboration that the claimed origin of the data received is indeed the real origin of the data, while *Entity Authentication* is the verification that the communicating entities are the ones claimed.
- *Access Control* provides protection against unauthorized use of resources (e.g. network services).
- *Confidentiality* provides protection of information from unauthorized disclosure.
- *Integrity* ensures that data is accurately transmitted from source to destination. The system must be able to counter equipment failure as well as actions by persons or processes that are not authorized to alter the data.
- *Non-Repudiation* protects against denial by one of the entities involved in a communication of having participated in all or part of the communication. Non-repudiation with proof of origin e.g. protects against any attempt by the sender to repudiate having sent a message, while non-repudiation with proof of delivery protects against any attempt by the recipient to falsely deny having received a message.

Security services describe natural user requirements. The relevance of a particular service depends on a number of factors. In ISO 7498-2 there is a detailed discussion on which services are relevant for which layers. But even if this is clarified other factors may be of importance. An example is whether users are communicating in connection oriented or connectionless mode. In connectionless mode data is transmitted without a connection being established explicitly. As each packet is sent on its own, connectionless data integrity is strongly related to data origin authentication. Therefore mechanisms are required which are able to provide data origin authentication and integrity. However, there is no protection against replay or deletion. Connection oriented mode applies to the situation in which a connection between two entities is established before user data is transmitted via the connection. In addition to the connectionless case one can think of mechanisms that prevent against insertion, replay, and deletion [ISO88].

According to ISO 7498-2, types of *Security Mechanisms* that can be used to provide the security services identified include encipherment, digital signature, access control, data integrity, authentication exchange, traffic padding, routing control, and notarization. The specification of suitable mechanisms is not within the scope of a security architecture. This is for instance dealt with in the ISO subcommittee ISO/IEC/JTC1/SC27 ("Security Techniques"). In the following we briefly discuss which type of cryptographic mechanism suits which security service.

- *Data origin authentication* implies that the originator of the information performs a transformation of the data out of which the receiver can unambiguously verify its origin. Of course the receiver needs to know who claims to have sent the data, because otherwise he has to try every possible sender. This information can be sent in the clear as it does not reduce the security of a system. An intruder can modify this additional information thus obstructing the authentication process, even if the information were not in the clear but without gaining anything. Under the assumption that the claimed sender can be derived from the received message, the receiver has to be able to find out if the claimed sender is really the one who sent the data. This requires the message to be linked to the identity of the sender. It has to be assured that neither changing the originator's identity nor changing the message will result in a successful data origin authentication. The latter condition implies that data origin authentication is based on a data integrity mechanism.

- *Entity Authentication* requires that the entity in question proves to know or to have something which makes the interrogator believe that he deals with the specific entity. Entity authentication can be considered as the sum of data origin authentication and a guarantee that it is not a replay of a previous transfer. The latter can be prevented by taking care that never the integrity of the same message has to be assured (e.g. by incorporating a date/time stamp into the data block) or that the mechanisms to provide entity authentication are never applied in exactly the same way (e.g. by changing the keys).

- *Access Control* can be realized in many different ways. One example is a process

where a subject interrogates an Access Control List (ACL) stating the access rights of each entity to each object. For this, it is necessary that the object managing the ACL can verify the authenticity of the inquirer. Thus, the access control mechanism utilizes an authentication mechanism.

- *Data Confidentiality* can e.g. be realized by a data routing mechanism that takes care that confidential data is not passing malicious entities, or by an encipherment mechanism where the corresponding decipherment operation can only be performed by authorized entities. Data confidentiality is excluded from the scope of RIPE.

- *Data Integrity* shall enable the receiving entity to detect any modification, insertion, deletion or replay of data (see DataOrigin Authentication).

- *Non-Repudiation* can take at least four forms: non-repudiation with proof of origin, receipt, delivery and submission. *Non-repudiation with proof of origin* enables the receiver of data to prove that the sender of the data really sent the data. This service is a specific combination of data integrity and entity or data origin authentication. Such a service is not only useful for the receiver but also important for the sender who is protected from being accused of sending data he did not send. Because a third party has to be able to make a distinction between sender and receiver, non-repudiation services suggest the use of asymmetric mechanisms. In an asymetric cryptographic mechanism it is computationally infeasible to deduce the enciphering key from the deciphering key and/or vice-versa. In a symmetric cryptographic algorithm those keys are equal, or can easily be deduced from one another. Computationally infeasible refers to a computation that is theoretically achievable, but not feasible in terms of the time taken to perform it with the current or predicted power of technology. The most common mechanisms are digital signatures. It should be noted,however, that non-repudiation services can also be based on symmetric authentication mechanisms. *Non-repudiation with proof of receipt* enables the sender of data to prove that its receiver really received the data. *Non-repudiation with proof of submission* enables the sender to prove that he submitted specific data. *Non-repudiation with proof of delivery* finally enables to prove that specific data was delivered (e.g. to a mailbox). Until recently, delivery and receipt were one service since a need to have both functions had not been identified.

3 Integrity Primitives

Security primitives are the basic building blocks of security services and mechanisms. Integrity primitives can be widely applied to protect information systems. This section introduces different types of primitives together with various kinds of uses that can be made of them. Figure 1 gives a taxonomy of security primitives. Integrity primitives are shown in boldface (note that in some documents, e.g. [ISO88] the term integrity has a much more restricted meaning). The given list of integrity primitives is not

considered to be complete. Especially, integrity primitives where there is no practical significance foreseen (e.g. claw-free permutations, randomized mappings) have not been sought for within the RIPE project.

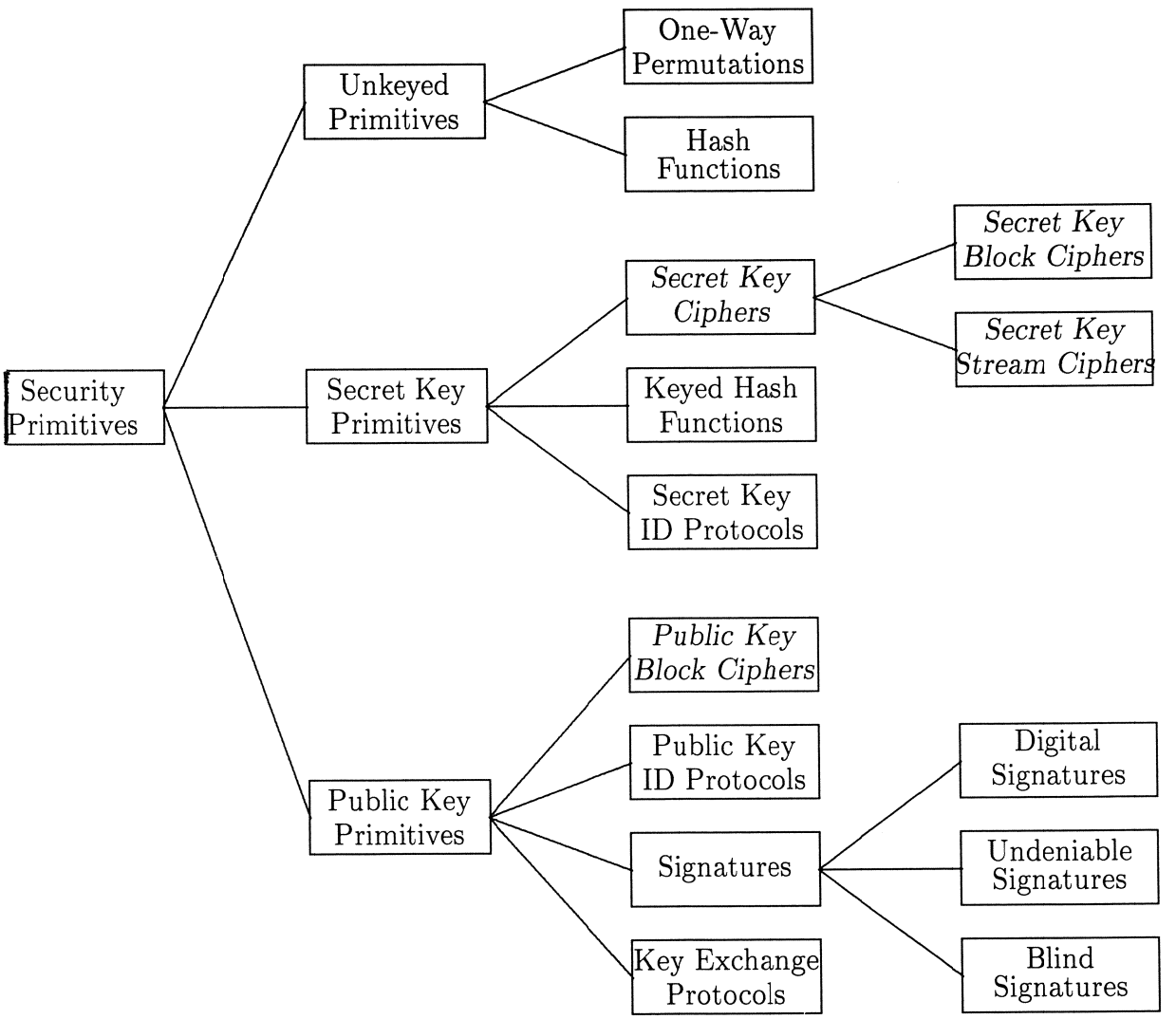

Figure 1: Taxonomy of Security Primitives.

Integrity primitives are of course ultimately intended to be useful to people. But actually computerized equipment will often be performing the necessary computation – perhaps even automatically. It will thus be convenient to abstract from the distinction between a person, a person directly controlling a computerized piece of equipment, or a computer autonomously playing a similar role. Accordingly, the active agents using the integrity primitives will simply be referred to as 'parties'.

The subsections – unkeyed, secret key, and public key – correspond to the second level of the taxonomy in the figure. The essential idea is that secret keys are known to

more than one party, but not to an adversary; and public keys are known to all parties including any adversary.

3.1 Unkeyed Integrity Primitives

There are two types of security primitives that are not parametrized by a cryptographic key ("unkeyed security primitives"). Both are essential for a broad spectrum of security services:

- A *One-Way Function* is a mapping for which it is in general infeasible given an image to find a function argument that results in that image. A *One-Way Permutation* is an bijective one-way function. One-way functions can be used to protect password files. Instead of storing passwords themselves in a file, they are transformed by a one-way permutation before being stored. The applied transform can be public, so that anyone can verify that a particular password is in the file. But due to the properties of the one-way function nobody can discover what any of the other passwords actually are. For a one-way permutation the result of applying the function naturally has to be as large as its input. In many practical applications, however, potential inputs are large, and producing a relatively small output is desirable.

- A *Hash Function* derives from any input an output of fixed length in such a way that it is practically infeasible to find two different messages that result in the same hash value (collision resistancy). Thus, a hash function is a one-way function that maps an arbitrary string to a result of a fixed length in bits. By a hash function in general many inputs will be mapped to the same output. A hash function can be used in a way similar to the way one-way permutations were used for passwords above. This would have the advantage that instead of passwords of a limited size, arbitrarily long 'passphrases' could be used. In fact, just by looking at the password file, no clue would be given as to the length of the corresponding password or passphrase. Another important field of application for hash functions are signature mechanisms that ask for a cryptographic 'footprint' of the data to be signed rather than applying the costly signature primitive to large amounts of data (such a signature mechanism often is called *Digital Signature with Appendix*). For a hash function based on a secret key block cipher a standard is under development [ISO92c].

3.2 Secret Key Integrity Primitives.

If not only the integrity of information is to be protected, but the information is also to be linked to an originator, a key has to be involved in the security mechanism, assuming that some coupling between a person and his or her key is established. There are two classes of security primitives that are parametrized by a cryptographic key: secret key primitives and public key primitives. Secret key primitives are based on the fact that the parties involved share a (secret) key. This key would be established

by the parties in advance, for instance by one party creating the key at random and supplying it to the other party without letting anyone else learn it. By secret key techniques, parties can protect themselves against outsider attacks. If protection from insider fraud is required, a third party has to be able to distinguish between entities and the capabilities of the entities have to be different. This effect can be achieved with public key primitives. For both classes the basic security primitives parametrized by a key are block ciphers.

- A *Block Cipher* is an invertible function parametrized by a key that maps message blocks of a fixed block length into ciphertext blocks of the same length. Block ciphers are keyed one-way permutations. There are secret key block ciphers (e.g. the Data Encryption Standard, DES [ANS81], [NBS77]) and public key block ciphers (e.g. the Rivest- Shamir-Adleman algorithm, RSA [RSA78]). The evaluation of block ciphers used to provide confidentiality is excluded from the scope of RIPE. But, given a block cipher there are several ways to construct other integrity primitives, such as one-way functions, keyed hash functions, or identification protocols. Such a construction is denoted as a 'mode-of- use' of the keyed one-way permutation. Mode of use usually refers to the manner in which a block cipher may be operated (e.g. electronic code book, cipher block chaining, etc.). Note that a block cipher must be a one-way function with respect to its key input, but it is not with respect to its data input.

Keyed hash functions play an important role for the authentication of messages, while identification protocols are to authenticate entities (e.g. persons, or devices).

- A *Keyed Hash Function* is a hash function parametrized by a (secret) key. Such a hash function can be used to add controlled redundancy to information and to thus protect the integrity of the information. Before a message is sent, the secret key is used to compute a keyed hash value from it. This value is sent along with the message and can be checked by the legitimate recipient using the common secret key. Since the hash function is parametrized by a secret key it furthermore allows to link the added redundancy to the originator of the information and therefore to provide both data integrity and data origin authentication. For that reason, keyed hash functions often are called *Message Authentication Codes* (MACs). Of course these same techniques may be applied to messages that are stored before they are ultimately received. A specific way to use a block cipher for the calculation of a MAC has been standardized [ISO89] (see also [ANS86], [ISO87]). Another important application for keyed hash functions are identification schemes.

- *Secret Key Identification Protocols* are based on the assumption that the parties involved share a common key prior to the application of the protocol. In a 2-way authentication scheme the basic ingredients of the protocol are a challenge from one entity which is answered by a token, that is a set of data items formed for an authentication exchange, from the other entity (e.g. a dynamic password scheme). In a 1-way authentication protocol a token from one entity to another

is simply transmitted. A *token* is a set of data items formed for an authentication exchange and transmitted from one entity to another. In this case the challenge answered is implicitly given by a time-stamp or a sequence number. For secret key identification protocols based on a secret key block cipher a standard is under development [ISO92a].

3.3 Public Key Integrity Primitives

As indicated above, the use of secret key primitives cannot solve disputes between sender and receiver, because they share the same secret information. The idea of public key primitives is that an entity is associated with a pair of related keys. One of the keys is called the 'private key' and is kept secret by its owner; the other key of the pair, called the 'public key', is made known in a way that ensures that it is genuine. Public keys might, in principle, be published in a newspaper. An essential part of the concept is that it must be infeasible to compute or simulate possession of the private key using only the public key, which is reminiscent of the one-way property described above.

Public key techniques are in some sense more powerful than the unkeyed and secret key techniques already described, but they also are more costly, i.e. in general their computational complexity is considerably higher. With public key techniques, the prearrangement of each participant publishing a public key allows any pair of participants to ensure the integrity of their communication. This flexibility would require one pair of keys for each possible pair of participants using only secret key techniques, which rapidly becomes impractical as the number of participants grows. An even more fundamental advantage is that every party can verify that a message must have been sent by the party owning a particular secret key. This allows the recipient of such a 'signed' message to provide the signature for verification by any other participant, even a judge.

- *Public Key Identification Protocols* are based on the fact that each entity owns a public key pair. A party that has published its public key can identify itself to any other party. Suppose a challenging party has obtained the public key of the identifying party. Then the challenging party can form a random challenge, encode it with the public key of the identifying party, and send the result to the identifying party. The identifying party can then use its private key to compute the inverse operation to recover the original challenge value, which can be returned to the challenging party. Thus the challenging party is convinced that the identifying party is the only one that could have made the computation. Public keys are typically certified by a *Certification Authority*, i.e. they are rendered unforgeable by a third party. A common procedure is to sign the public key of an entity together with some additional information used to authenticate that entity using the private key of the certification authority. The set of data items assigned to an entity and used to authenticate that entity is called its *credentials*. A set of credentials together with the certification authority's signature of those credentials is called a *certificate*. For public key identification protocols

also a standard is under development [ISO92b]. As for secret key identification protocols, the timeliness of the tokens can be established via timestamps or sequence numbers, or by answering explicit challenges as described above. In any case, public key identification protocols are based on public key block ciphers.

- A *Digital Signature* is to serve as the electronic equivalent of a written signature. It allows to 'sign' a message so that anyone can verify the signature, i.e. prove that a message has been created by a specific entity. Since the receiver must not be able to construct the signed message, the process of signing requires secret information. The validation of the signature, however, only requires access to public information. With an *Undeniable Signature* collaboration of the signing entity is necessary for that proof. *Blind Signatures* have been introduced to provide anonymity to the signer (e.g. in the context of electronic money) and are based on commutative one-way permutations. For digital signatures that provide message recovery the additional specification of a special transformation introducing redundancy is necessary, while digital signatures with appendix require a collision-resistant hash function. One specific way to calculate a digital signature with message recovery has been standardized [ISO91].

- *Key Exchange Protocols* are to establish a secret key between the communicating parties. In many cases identification protocols can easily be modified to provide additionally for key exchange. An example for a key establishment protocol is given in [DiHe79]. Standards for key exchange protocols are currently under development.

4 Integrity Mechanisms

In this Section we will establish a relationship between integrity mechanisms and integrity primitives.

- *Data Origin Authentication* and *Data Integrity* mechanisms make use of unkeyed hash functions (Manipulation Detection Code, MDC), or of keyed hash functions (Message Authentication Code, MAC). In the case of an MDC the integrity of the hash value has to be protected, e.g. by encipherment or by an integrity channel (e.g. a phone when people know each other's voice). With a MAC protection of authenticity is independent of confidentiality but requires its own key. With an MDC authentication is separated from encipherment but compromised when the cipher is broken.

- *Entity Authentication* mechanisms are based on secret key or public key encipherment schemes, or on keyed hash functions. In the symmetric case the parties have to share the same secret information. If this cannot be assumed they each have to share some secret information with an on-line trusted center. In the asymmetric case the identities of the entities can be certified by an off-line trusted center that issues credentials. With zero knowledge (minimum knowledge) authentication, the asymmetric protocol between prover and verifier is designed such that the verifier learns nothing but the validity of the assertion (gets only minimum additional knowledge).

  Authentication in general is based on proving/verifying specific knowledge or possession of something (e.g. a secret key). Such a proof typically requires a protocol with a number of interactions (passes). Two-pass authentication for instance involves a challenge (e.g. a random number) from one entity and a response from the other entity which convinces the challenging entity (e.g. that the challenged entity has a specific secret key). One-pass authentication makes use of the same principle but employs an implicit challenge (e.g. a time stamp or a sequence number).

- *Signature* mechanisms are based on asymmetric block ciphers. Signature mechanisms with message recovery need the additional specification of a special transformation introducing the required redundancy. Signature mechanisms with imprint/appendix as well as signature mechanisms based on zero knowledge proofs require a collision-resistent hash function. Undeniable signatures are based on commutative one-way functions, while blind signatures are based on commutative one-way permutations.

- *Encipherment* mechanisms are based on block ciphers in various modes of use.

- *Key Exchange* mechanisms are based on commutative one-way functions or on zero-knowledge techniques and will typically be integrated into entity authentication mechanisms.

References

[ANS81] ANSI X3.92-1981, *Data Encryption Algorithm*, 1981.

[ANS86] ANSI X9.9-1986, *Financial Institution Message Authentication*, 1986.

[Bra88] G. Brassard, *Modern Cryptography*, Lecture Notes in Computer Science no. 325, Springer-Verlag, Berlin-Heidelberg-New York, 1988.

[DaPr89] D.W. Davies and W.L. Price, *Security for Computer Networks, 2nd ed.*, Wiley & Sons, New York, 1989.

[DiHe79] W. Diffie, M.E. Hellman, "New Directions in Cryptography," *IEEE Transactions on Information Theory*, vol. 22, pp. 644-654, 1976.

[FuRi88] W. Fumy and H.P. Riess, Kryptographie, Oldenbourg, Munich, 1988.

[ISO83] ISO International Standard 7498, *Open Systems Interconnection: Basic Reference Model*, 1983.

[ISO87] ISO International Standard 8731-1, *Banking - Approved Algorithms for Message Authentication Part 1: DEA*, 1987.

[ISO88] ISO International Standard 7498-2, *Open Systems Interconnection Reference Model - Part 2: Security Architecture*, 1988.

[ISO89] ISO/IEC International Standard 9797, *Data Integrity Mechanism Using a Cryptographic Check Function Employing a Block Cipher Algorithm*, 1989.

[ISO91] ISO/IEC International Standard 9796, *Digital Signature Scheme Giving Message Recovery*, 1991.

[ISO92a] ISO/IEC Committee Draft 9798-2, *Entity Authentication Mechanisms - Part 2: Entity Authentication Using Symmetric Techniques*, 1992.

[ISO92b] ISO/IEC Draft International Standard 9798-3, *Entity Authentication Mechanisms - Part 3: Entity Authentication Using a Public-Key Algorithm*, 1992.

[ISO92c] ISO/IEC Committee Draft 10118-2, *Hash Functions - Part 2: Hash Functions Using a Symmetric Block Cipher Algorithm*, 1992.

[MeMa82] C.H. Meyer and S.M. Matyas, *Cryptography: A New Dimension in Computer Data Security*, Wiley & Sons, New York, 1982.

[NBS77] National Bureau of Standards, *Data Encryption Standard*, Federal Information Processing Standard, Publication 46, US Department of Commerce, January 1977.

[PoKl89] G.J. Popek and C.S. Kline, "Encryption and secure computer networks," *ACM Computing Surveys*, vol. 11, pp. 331-356, 1979.

[RSA78] R.L. Rivest, A. Shamir and L. Adleman, "A Method for obtaining digital signature and public key cryptosystems," *Communications of the ACM*, vol. 21, pp. 120-126, 1978.

[VoKe85] V.L. Voydock and S.T. Kent, "Security in high level network protocols," *IEEE Communications Magazine*, pp. 12-24, July 1985.

Part III

Recommended Integrity Primitives

Chapter 1

Introduction to Part III

Contents

This part of the RIPE final report contains full descriptions, detailed analysis as well as guidelines for use of the RIPE primitives. Each chapter is devoted to one of the recommended primitives.

The chapters are all organized in the same manner. Each starts with a short introduction (Section 1). In order to avoid ambiguities in the descriptions, notation and relevant definitions are then given (Section 2). After this, the primitive is described (Section 3), followed by the recommended modes of use (Section 4). Next, the security aspects are discussed including claimed properties and a presentation of the main results of the algebraic evaluation (Section 5). Finally, hardware and software implementation are considered and some guidelines for software implementation are given (Section 6). The appendices to each chapter contain sample software implementations in the programming language C and test values for the primitives to allow verification.

The chapters in this part are organized according to the taxonomy introduced in Chapter 3 of Part II. The first two chapters contain unkeyed integrity primitives, the next three secret key integrity primitives and the final two public key primitives. Here we first briefly indicate how each of the recommended primitives made it into the portfolio. Figure 1 illustrates how the RIPE primitives fit in the taxonomy tree introduced in Part II of this report.

1 Unkeyed integrity primitives

We present two hash functions, MDC-4 and RIPEMD. The former is especially suited for hardware implementation, the latter for software implementation.

MDC-4 (MDC stands for Manipulation Detection Code), invented by D. Coppersmith, S. Pilpel, C.H. Meyer, S.M. Matyas, M.M. Hyden, J. Oseas, B. Brachtl, and M. Schilling is a hash function based on a symmetric block cipher and is recommended for use with a DES-engine.

RIPEMD (MD in this case stands for Message Digest) is based on the publicly known hash function MD4, which was submitted to RIPE by RSA Inc. However, due to partial attacks on the first two out of three rounds (found by R. Merkle) and also on the last two rounds (discovered by the RIPE-team), it was concluded that applying the design principle of MDC-4 to MD4 would make it a sufficiently secure unkeyed hash function, which would be very fast in software. The use of MD4 was preferred over its follow-up, MD5, which has been discovered to have weak collisions (found by B. den Boer and A. Bosselaers).

2 Secret key integrity primitives

Next we present three primitives, two of which are keyed hash functions (RIPEMAC and IBC-Hash), whereas the third is a secret key identification protocol.

The design of RIPEMAC (MAC stands for Message Authentication Code) is inspired by the standardized "CBC-MAC" function (see, e.g., ISO/IEC 9797). Like CBC-MAC, RIPEMAC is based on the block chaining mode of a block-cipher. It has

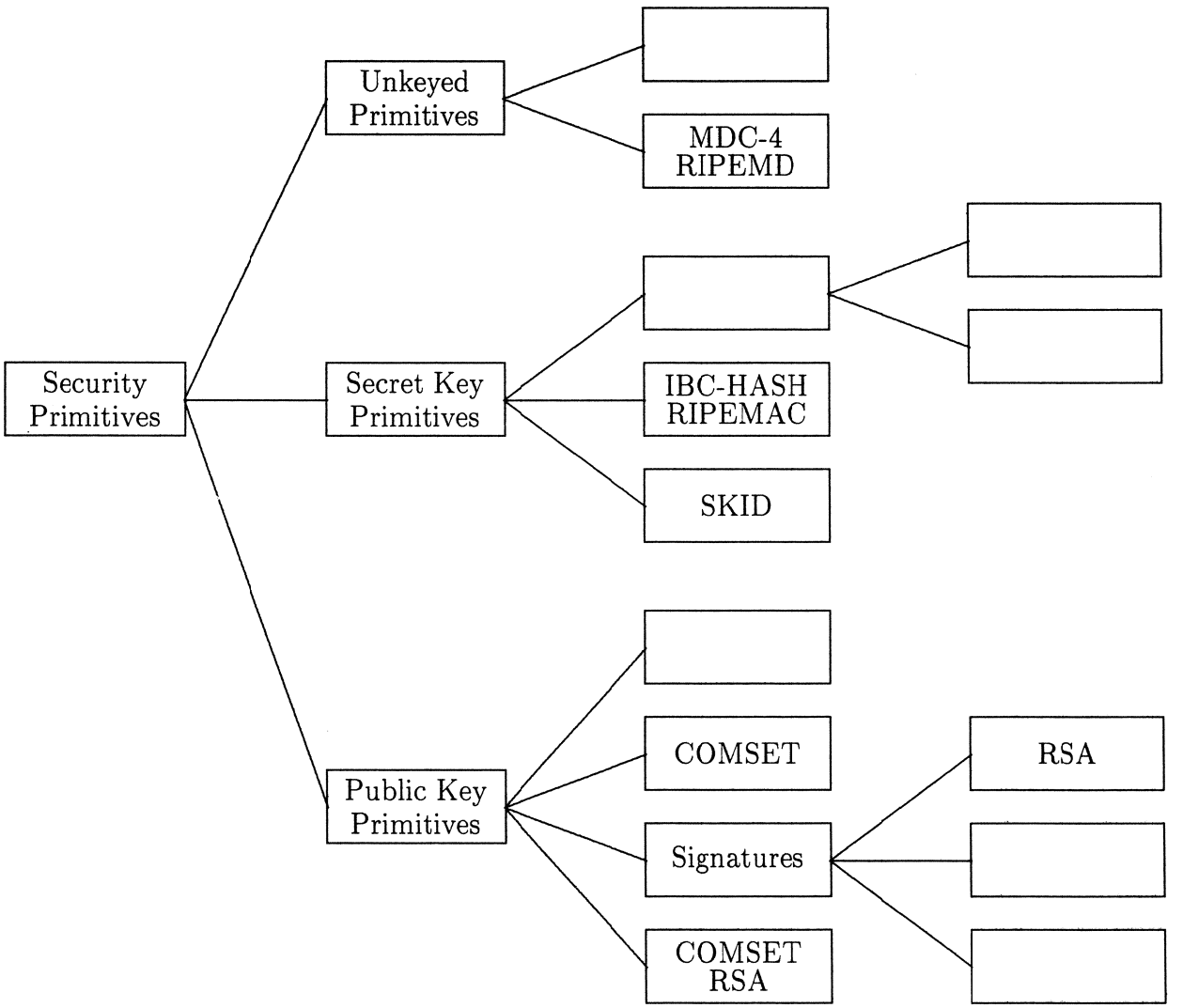

Figure 1: Mapping of the RIPE primitives on the taxonomy tree.

the additional feature that it cannot be directly used for concealment. Two variants of RIPEMAC are given, which are based on single and triple encryption—corresponding to increasing levels of security.

The second keyed hash function, IBC-hash, has been designed by members of the RIPE-team, David Chaum, Maarten van der Ham and Bert den Boer. IBC-Hash has the distinguished feature of being both provably secure and rather fast in both hardware and software.

The primitive named SKID actually contains two secret key identification protocols that can be used to provide entity authentication. The two variants are for unidirectional and bidirectional authentication. Both are based on a keyed one-way function rather than on a block cipher. Their design emphasizes the integrity oriented approach of the project. These protocols are expected to fill an important gap in ISO-standardization and to be incorporated in future versions of the proposed set of entity authentication standards specified in ISO/IEC 9798. The SKID primitives are not the result of a submission but have been selected and enhanced by Markus Dichtl and Walter Fumy.

3 Public key primitives

Finally, the RIPE portfolio contains two public key integrity primitives. One of them is the well-known RSA algorithm, while the other one is a public key identification protocol based on the Rabin scheme.

The penultimate chapter is based on the RSA public key scheme, which has been submitted to RIPE by RSA Inc. It includes an evaluation of the ISO/IEC 9796 standard for a digital signature with message recovery. This standard was developed by an ISO editorial group and has now been independently evaluated for the first time. The digital signature based on ISO/IEC 9796 is thus one recommended mode of use for RSA, where the input typically would be the result of one of the RIPE unkeyed integrity primitives applied to a message. Another recommended mode of use provides key forwarding, which would find its application typically in batch settings, such as EDIFACT.

The last chapter presents a public key identification protocol, COMSET, which has been designed and submitted by a research group at Aarhus University, Jørgen Brandt, Ivan Damgård, Peter Landrock and Torben Pedersen. Originally, it was developed for DANSECT, a secure system developed by the Danish Telecommunication Companies. COMSET is a zero-knowledge mutual identification scheme, which in addition allows for secret key exchange. One of the foreseen applications of COMSET is interactive EDI.

These chapters are followed by two useful appendices, one which is supposed to serve as a guide to RSA key generation, and the other of which introduces the basic principles of the modular arithmetic computations needed.

Chapter 2

MDC-4

Contents

1 Introduction

This chapter describes the integrity primitive MDC-4 [MeSc88, Mey89, Mat91]; we observer that it has been patented, see [CPMM90].

MDC-4 is a so-called *modification detection code* (MDC) or *hash function* that compresses messages of arbitrary length to a 128-bit output block, the *hashcode* of the message. It is claimed that it is computationally infeasible to produce two messages having the same hashcode, or to produce any messages having a given prespecified target hashcode. Hash functions with these properties are used in message authentication applications such as the protection of the integrity and the origin of data stored or transmitted using secret-key or public-key techniques (see Part II of this report).

MDC-4 processes the data in blocks of 64 bits. The name MDC-4 is derived from the four DES [NBS77] encryptions that are required to process a single message block. The algorithm itself specifies no padding rule. The padding algorithm used is that of RIPEMD (see Chapter 3), which is the same as that of MD4 [Riv90].

Since the MDC-4 algorithm uses four DES encryptions per message block, its design is oriented towards implementations using hardware realizations of the DES. The performance of a pure software implementation will suffer noticeably from the low software performance of the DES, especially since each DES application involves a change of the key.

The structure of this chapter is as follows. In order to avoid any ambiguities in the description of the primitive, the notation and definitions in this chapter are fixed in Section 2. Section 3 contains a description of the primitive and in Section 4 the possible modes of use of the primitive are considered. The security aspects of the primitive are discussed in Section 5. These include the claimed properties and the results of algebraic evaluations of the primitive. Finally, in Section 6 the performance aspects of MDC-4 are considered, and Section 7 gives some guidelines for software implementation.

This chapter has two appendices. Appendix A contains a straightforward software implementation of MDC-4 in the programming language C and in Appendix B test values for the primitive are given.

2 Definitions and Notation

2.1 Introduction

In order to obtain a clear description of the primitive, the notation and definitions used in this chapter are fully described in this section. These include the representation of the numbers in the description, and the operations, functions and constants used by the primitive.

2.2 General

The symbol ":=" is used for the assignment of a value or a meaning to a variable or symbol. That is, $a := b$ either means that the variable a gets the value of the variable b, or it means that a is defined as "b". It will be obvious from the context which meaning is intended.

The equality-sign "=" is used for equality only. That is, it indicates that the two entities on either side are equal.

Note that in C-source code, '=' denotes assignment, while comparison is denoted by '=='.

An ellipsis ("...") denotes an implicit enumeration. For example, "$i = 0, 1, \ldots, n$" is meant to represent the sentence "for $i = 0$, $i = 1$, and so on, up to $i = n$".

2.3 Representation of the Numbers

In this chapter a *byte* is defined as an 8-bit quantity and a *word* as a 64-bit quantity. A byte is considered to be a nonnegative integer. That is, it can take on the values 0 through $2^8 - 1 = 255$. Likewise, a word is considered to be a nonnegative integer, hence it takes on the values 0 through $2^{64} - 1 = 18\,446\,744\,073\,709\,551\,615$. The value of a word can be given in decimal as well as in hexadecimal form. In the latter case the number is written as '0x' followed immediately by at most 16 hexadecimal digits, the most significant one first. For example, the hexadecimal representation of the 64-bit number 5 931 894 172 722 287 186 is 0x5252525252525252.

A sequence of $8n$ bits $b_0, b_1, \ldots, b_{8n-1}$ is interpreted as a sequence of n bytes in the following way. Each group of 8 consecutive bits is considered as a byte, the first bit of such a group being the most significant bit of that byte. Hence,

$$B_i := b_{8i}2^7 + b_{8i+1}2^6 + \cdots + b_{8i+7}, \quad i = 0, 1, \ldots, n-1. \tag{1}$$

A sequence of $8l$ bytes $B_0, B_1, \ldots, B_{8l-1}$ is interpreted as a sequence of words $W_0, W_1, \ldots, W_{l-1}$ in the following way. Each group of 8 consecutive bytes is considered as a word, the first byte of such a word being the most significant byte of that word. Hence,

$$W_i := B_{8i}(256)^7 + B_{8i+1}(256)^6 + \cdots + B_{8i+6}(256) + B_{8i+7}, \quad i = 0, 1, \ldots, l-1. \tag{2}$$

In accordance with the notations above, the bits of a word W are denoted as

$$W = (w_0, w_1, \ldots, w_{63}), \tag{3}$$

where

$$W = \sum_{i=0}^{63} w_i 2^{63-i}. \tag{4}$$

In this chapter words are always denoted by uppercase letters and the bits of this word by the corresponding lowercase letter with indices as in Equation (3). Likewise, bytes are indicated by uppercase letters and the bits that constitute this byte by lowercase letters with indices as in Equation (1). The ordering of bytes in a word is given by Equation (2).

2.4 Definitions and Basic Operations

- A *string* is a sequence of bits. If X is a string consisting of n bits, then those bits are denoted from left to right by x_0, x_1, ..., x_{n-2}, x_{n-1}.
- For a string X the *length* of X is denoted as $|X|$. That is, $|X|$ is the number of bits in the string X. If $|X| = n$, then X is said to be an n-bit string.
- Strings of length 32 will also be considered as words according to the representation defined by (4), and vice versa. Hence, if X is a string of length 32, then the corresponding word is equal to

  $$X = \sum_{i=0}^{6} 3x_i 2^{63-i}.$$

  Note that the same symbol is used for both the string and the corresponding word. It will be clear from the context which representation is intended.
- For an integer N, the *length* of N is defined as the length of the shortest binary representation of N. This is the representation with most significant bit equal to 1. (All "leading zeros" are removed.) The length of N is denoted as $|N|$.
- For two strings X and Y of length $|X| = |Y| = n$, the $2n$-bit string $W = X||Y$ is defined as the *concatenation* of the strings X and Y. That is, according to Equation (3)

  $$\begin{aligned} w_i &:= x_i \qquad i = 0, 1, \ldots, n-1 \\ w_{i+n} &:= y_i \qquad i = 0, 1, \ldots, n-1. \end{aligned}$$

- For a nonnegative integer A and a positive integer B, the numbers A div B and A mod B are defined as the nonnegative integers Q, respectively R, such that

  $$A = QB + R \quad \text{and} \quad 0 \leq R < B.$$

  That is, A mod B is the *remainder*, and A div B is the *quotient* of an integer division of A by B.

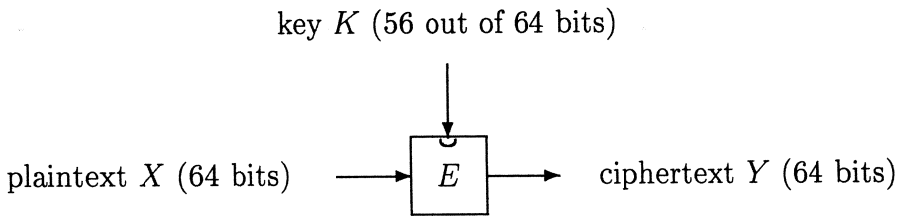

Figure 1: The basic DES encryption operation.

- For two words X and Y, the words $U = X \oplus Y$ is defined as the bitwise *XOR* of X and Y. Hence, according to Equation (3):

$$u_i := (x_i + y_i) \bmod 2 \quad (i = 0, 1, \ldots, 63).$$

2.5 Functions used by MDC-4

MDC-4 uses two basic functions $E_1^{\oplus}(\cdot)$ and $E_2^{\oplus}(\cdot)$ that each map two words onto a single word. They are both one-way functions based on the Data Encryption Standard (DES) [NBS77] in which certain key bits are kept constant. A DES encryption operation $E(\cdot)$ will be graphically represented as shown in Figure 1 and mathematically written as

$$Y = E(K, X),$$

where K is the key and X is the 64-bit plaintext block to be encrypted. The key K is represented as a word $(k_0, k_1, \ldots, k_{63})$, but in the DES encryption operation the key bits $k_7, k_{15}, k_{23}, k_{31}, k_{39}, k_{47}, k_{55}$ and k_{63} are ignored.

A DES based one-way function $E^{\oplus}(\cdot)$ is obtained by combining the plaintext and the DES ciphertext with an XOR-operation:

$$Y = E^{\oplus}(K, X) := E(K, X) \oplus X$$

This primitive is depicted in Figure 2, together with a shorthand.

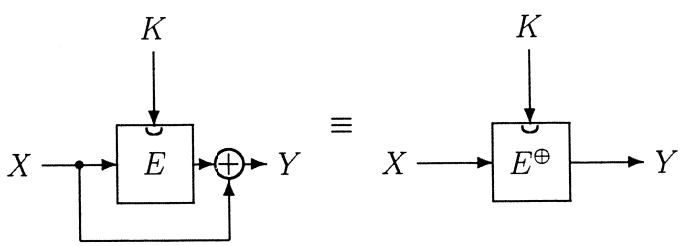

Figure 2: The DES based one-way function $E^{\oplus}$, together with a shorthand.

The one-way functions $E_1^{\oplus}(\cdot)$ and $E_2^{\oplus}(\cdot)$ used in MDC-4 are defined in terms of this DES based one-way function $E^{\oplus}(\cdot)$ as follows. Let K' be equal to K except that bits

k'_1 and k'_2 are set equal to respectively 1 and 0. Similarly let K'' be equal to K except that bits k''_1 and k''_2 are set equal to respectively 0 and 1.

$$\begin{aligned} E_1^{\oplus}(K,X) &:= E^{\oplus}(K',X) \\ E_2^{\oplus}(K,X) &:= E^{\oplus}(K'',X) \end{aligned}$$

They will be graphically represented like the DES based one-way function $E^{\oplus}(\cdot)$, with $E^{\oplus}$ replaced by $E_1^{\oplus}$ or $E_2^{\oplus}$.

In the description of the MDC-4 scheme, the one-way functions $E_1^{\oplus}(\cdot)$ and $E_2^{\oplus}(\cdot)$ can be substituted by two other one-way functions. However, the security of such a scheme would have to be re-evaluated, as it heavily depends on the properties of these new one-way functions. This chapter only considers the DES-based one-way function.

3 Description of the Primitive

3.1 Outline of MDC-4

MDC-4 is a hash function that maps a message M of arbitrary length onto a 128-bit block MDC-4(M). The basis of MDC-4 is the compression function compress. This function compresses a three word input to a two word output. This function is used in the following way.

First, the message M is expanded to an appropriate length and represented as a sequence of words X. Then, starting with a two-word initial vector, the sequence X is compressed by repeatedly appending a message word and compressing the resulting three words to two words by applying compress until the message is exhausted. Thus, a two-word result is obtained. Below this is explained in detail.

Let $M = (m_0, m_1, \ldots, m_{n-1})$ be a message of n bits long. The hash value MDC-4(M) of M is computed in two steps.

expansion: M is expanded to a sequence X consisting of N words X_0, X_1, ..., X_{N-1}, where $N = (n \text{ div } 64)+2$. That is, the message is expanded such that it becomes a multiple of 64 bits. Padding is done even if the original message length is a multiple of 64 bits.

compression: Define the words $H1_0$ and $H2_0$ as

$$\begin{aligned} H1_0 &:= \text{0x5252525252525252} \\ H2_0 &:= \text{0x2525252525252525}. \end{aligned}$$

For $i = 0, 1, \ldots, N-1$, the words $H1_{i+1}$ and $H2_{i+1}$ are computed from the words $H1_i$ and $H2_i$ and the message word X_i as follows:

$$(H1_{i+1}, H2_{i+1}) := \text{compress}((H1_i, H2_i), X_i).$$

The hash value MDC-4(M) is equal to the 128-bit string $H1_N \parallel H2_N$, where the interpretation of the words $H1_N$ and $H2_N$ in terms of bitstrings is given by Equation 3.

3.2 Expanding the Message

Let $N = ((n+64) \text{ div } 64)+1$. The n-bit message $M = (m_0, m_1, \ldots, m_{n-1})$ is expanded to the N-word message $X = (X_0, X_1, \ldots, X_{N-1})$ in the following three steps.

1. Append a single "1" and $k = 63 - (n \bmod 64)$ zero bits to the message M:

$$\begin{aligned} m_n &:= 1, \\ m_{n+1} &:= m_{n+2} := \cdots := m_{n+k} := 0. \end{aligned}$$

This step expands the message M to a message that is a multiple of 8 bytes. Note that padding is done even if the length of M is already a multiple of 8 bytes.

2. Transform this $(n+k+1)$-bit extended message into the $\frac{n+k+1}{64} = N-1$ words $X_0, X_1, \ldots, X_{N-2}$ according to the conventions defined in Section 2.3. Hence

$$X_i := \sum_{j=0}^{63} m_{64i+j} 2^{63-j} \qquad i = 0, 1, \ldots, N-2.$$

3. Complete the expansion by appending the length n of the original message:

$$X_{N-1} := n \bmod 2^{64}.$$

3.3 The Compression Function compress

For the 2-word number $(H1_i, H2_i)$ and the message word X_i the 2-word function value

$$(H1_{i+1}, H2_{i+1}) := \mathsf{compress}((H1_i, H2_i), X_i)$$

is computed as follows (see also Figure 3).

$$\begin{aligned}
T1_i = LT1_i \parallel RT1_i &:= E_1^{\oplus}(H1_i, X_i) \\
T2_i = LT2_i \parallel RT2_i &:= E_2^{\oplus}(H2_i, X_i) \\
U1_i &:= LT1_i \parallel RT2_i \\
U2_i &:= LT2_i \parallel RT1_i \\
V1_i = LV1_i \parallel RV1_i &:= E_1^{\oplus}(U1_i, H2_i) \\
V2_i = LV2_i \parallel RV2_i &:= E_2^{\oplus}(U2_i, H1_i) \\
H1_{i+1} &:= LV1_i \parallel RV2_i \\
H2_{i+1} &:= LV2_i \parallel RV1_i
\end{aligned}$$

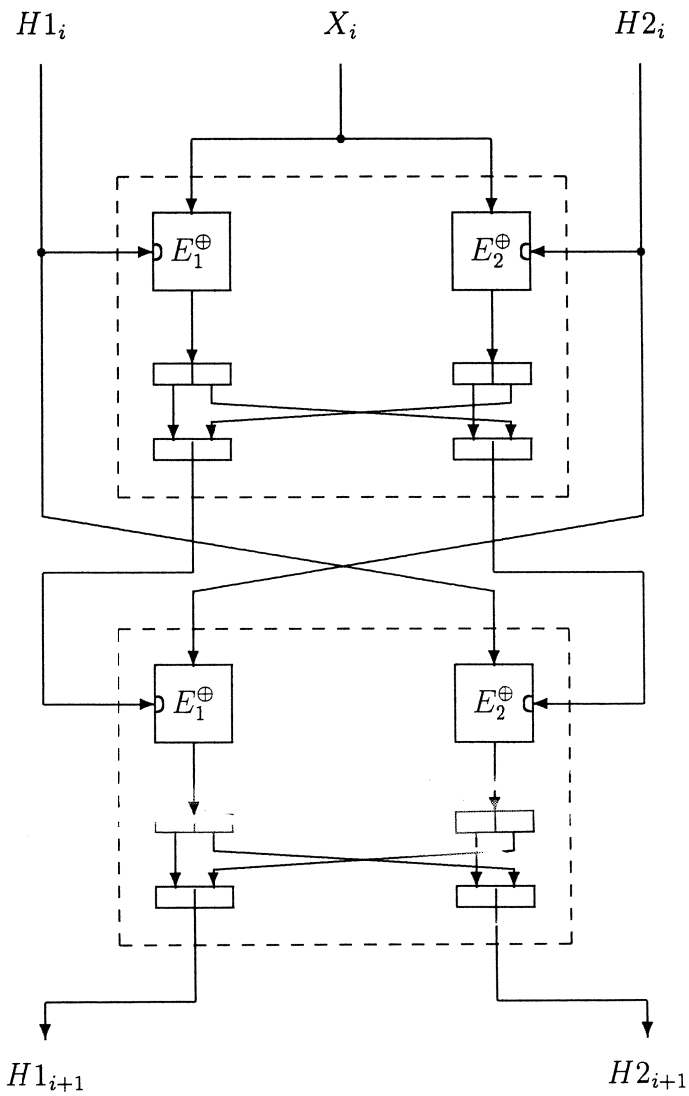

Figure 3: Outline of the compression function compress.

4 Use of the Primitive

The primitive MDC-4 (and any other hash function) can be used in the following ways.

- Firstly, it can be used in a data integrity mechanism, namely for the verification of the authenticity of a message.
- Secondly, it can be used in a signature mechanism, namely to compress a message before signing it.
- Thirdly, it can be used for the generation of publicly accessible password or passphrase files.

Those modes of use are explained below.

Use in data integrity mechanisms The primitive MDC-4 can be used for the verification of the authenticity of a message. This is done as follows, and under the following constraints.

After receipt of a message and the corresponding hashcode, one (re)calculates the hashcode of the message. If the calculated hashcode is equal to the value originally received, it is reasonable to assume that the original hashcode is computed from the same message. This holds, as computation of another message with the same hashcode is claimed to be infeasible (see Section 5.1). Therefore, if it can be guaranteed that the hashcode is unmodified, it is reasonable to assume that the message is authentic (unmodified).

So, in order to verify the authenticity of a message, only the hashcode must be protected against modification. For the message itself this is not required. The protection of the hashcode may be achieved by public key methods, such as digital signatures, by secret key methods, such as encryption, or through physical or logical means. For more details, see Part II of this report. An example of a public key primitive that achieves this is RSA described in Chapter 7 of this report.

Use in signature mechanisms A digital signature enables the receiver of a message to prove to a third party that the message has been created by a specific entity.

If the message is long, it is much more efficient to sign the hashcode of a message than to sign the message itself. Because of the collision resistance and one-way property, it is infeasible to find two messages with the same hashcode or to find a message resulting in a given hashcode (see Section 5.1). As a consequence, it is infeasible to find two messages with the same signature, or to find a message resulting in a given signature as well.

For a digital signature primitive we refer to Chapter 7 of this report.

Use for readable password files Because of the collision-resistance and the one-way property, MDC-4 can be used for the generation of publicly accessible (readable) password or passphrase files. This is done by storing MDC-4(*password*) in this file,

instead of the password itself. Because of the one-way property, it is infeasible to derive a valid password from an entry in the file. Because of the collision resistance, this holds even if a valid password is known.

5 Security Evaluation

5.1 Claimed Properties

MDC-4 is conjectured to be a one-way collision resistant hash function. That is, it is conjectured to satisfy the following two conditions:

1. It is computationally infeasible to compute two distinct messages M_1 and M_2 that have the same hashcode (message digest), i.e., MDC-4(M_1) = MDC-4(M_2). A hash function that satisfies this property is called *collision resistant.*

2. It is computationally infeasible, given a message and the corresponding hashcode, to compute a different message with the same hashcode. That is, given a message M, it is infeasible to compute a message $M' \neq M$ such that MDC-4(M') = MDC-4(M). A hash function that satisfies this property is called *one-way.*

By "computationally infeasible" we mean to express the practical impossibility of computing something with the technology that is currently available or can be foreseen to become available in the near future.

It is hard to give a bound beyond which a computation is infeasible, but currently, a computation requiring 2^{60} (or 10^{18}) operations is computationally infeasible. On the other hand, a computation taking 2^{40} (about 10^{12}) operations is hard, but not impossible.

In view of the past development of computing power and the relatively new possibility of using huge amounts of computing power of for example idle workstations in a network, one should cater for a substantial increase in the bound for computational infeasibility on the longer term.

Since a proof for the conjectured strength of MDC-4 cannot be given, the rest of this section is dedicated to the algebraic evaluation of the algorithm.

5.2 Algebraic Evaluation

5.2.1 Justification of the Focus on compress

A theorem by Damgård [Dam89] states that a family of hash functions satisfying certain requirements is collision resistant and one-way if the compression function is collision resistant. This implies that a member of such a family with sufficiently large input block size, is collision resistant and one-way with high probability. As MDC-4 can easily be embedded in such a family, Damgård's theorem provides evidence for the fact that MDC-4 is collision resistant and one-way if compress is collision resistant. This justifies the fact that this section focuses on the properties of compress rather than MDC-4.

5.2.2 Collision Resistance and One-wayness of compress

The collision resistance of compress heavily depends on the properties of the DES, or, more precisely, on those of the $E^{\oplus}$-operation. Since its introduction in 1977, DES has

been studied extensively, but to date no significant weaknesses have been found. The most effective known attack on DES is the so-called *differential cryptanalysis* attack of Biham and Shamir [BiSh91a, BiSh91b].

In this section we first consider some general attacks on compress. After that the resistance of compress against differential cryptanalysis is investigated and finally, the infeasibility of brute force attacks on compress is discussed.

An obvious attack on compress is to make use of so-called 'weak keys' of DES [MoSi87] or to try to force $H1$ and $H2$ to the same value. Such attacks do not work because of the following reasons.

- As $H1$ and $H2$ are used as keys in DES, the value of the DES-keys cannot easily be forced.
- Because bits 1 and 2 of the keys are forced to different values, it is impossible to have identical DES-keys in $E_1^{\oplus}$ and $E_2^{\oplus}$ or to have weak keys in one of the applications of DES.

Furthermore, because of the extra XOR in the $E^{\oplus}$-operations, it is hard to 'work backwards' through MDC-4. If this was possible, brute force attacks could be made considerably faster than mentioned below. Finally, it is worth mentioning that the switching of the half blocks increases the dependency between the halves.

Due to the internal structure of compress, a collision for the upper part, i.e., two message blocks X_i and X_i' that yield the same value for $T1_i$ and $T2_i$ for a given value of $H1_i$ and $H2_i$, is also a collision for compress. Below we investigate whether this property and differential cryptanalysis of DES can be used to find collisions in compress.

Differential cryptanalysis is a probabilistic attack on DES-like algorithms based on the relation between the XOR of two different inputs and the XOR's of the respective intermediate results and outputs. Note that two inputs such that their XOR is preserved by DES yield the same output of the first $E^{\oplus}$-operation, as the XOR's cancel out. Hence, differential cryptanalysis in principle is applicable to compress, and might yield two blocks compressing to the same result. Furthermore, the key is fixed and known (or even chosen by the cryptanalyst). Since only one pair of input blocks is needed, this might be chosen to optimize the probability of success.

However, since the keys used in the first $E_1^{\oplus}$ respectively $E_2^{\oplus}$-operation are different by definition, an XOR of a pair of inputs has to be preserved by DES for both keys simultaneously. Essentially, the probability of success is the product of the probabilities for both keys $H1$ and $H2$ separately. This roughly squares the number of steps of an attack. Furthermore, there are some unsolved technical problems, the main one being the fact that known attacks work for an odd number of rounds of DES only. This implies that without a major breakthrough, a differential attack will be far slower than a brute force attack. The current estimate for the complexity of a differential attack is about 2^{90} steps.

Given that no internal weakness in the $E^{\oplus}$ operations can be used to find collisions for the function compress, only 'brute force' attacks on compress and hence on the hash function seem possible.

To date, the best attack to find a pair of message blocks that compress to the same hashcode, given a value for $(H1, H2)$ is a so-called 'birthday attack', see for example [QuDe89]. Such an attack will require in the order of 2^{54} steps. Here each step essentially consists of an application of compress (i.e., two applications of DES, including key scheduling). Although 2^{54} operations is for the moment computationally infeasible, it can be expected that this becomes feasible during the next decade.

The best attack that finds an input or a second input to compress that yields a prespecified output is a 'brute force' attack. This implies that no better attack on MDC-4 can be found. Hence the best attack that finds a preimage that compresses to a prespecified hashcode, given a value for $(H1, H2)$, is a pure brute force attack. This would take in the order of 2^{127} evaluations of compress. Of course, this number is also limited by the effective size of the message space, hence this space should not be too small.

Finding a different message that compresses to the same preimage as a given message also requires in the order of 2^{127} evaluations of compress. If we allow n different hashcodes, the complexity of this attack would decrease by a factor of n, but a storage in the order of n input/hashcode pairs is necessary.

We can therefore conclude that brute force attacks to find a preimage are computationally infeasible.

So, currently there seems to be no reason to doubt the security of MDC-4. However, a birthay attack to find collisions for compress might become feasible in the future.

6 Performance Evaluation

6.1 Software Implementations

The figures for a high performant software implementation of MDC-4 are given in Table 1. They use the ideas introduced in Section 7 to improve the DES performance as well as the performance of the compression function compress. Both a C and a 80386 Assembly language implementation are considered. The C version has the advantage of being portable (and has been ported, see Appendix A). It is in the configuration described below not much slower than the Assembly language implementation. However, as explained in Section 7, this is not necessarily the case for other configurations. Both versions use the same tables totalling 224K of memory, of which 96K is used to implement the DES encryption operation and 128K for the keyschedule operation. The code of the compression function takes in addition to that about 7K. The figures of Table 1 are for an IBM-compatible 33 MHz 80386DX based PC with 64K cache memory using WATCOM C/386 9.0 in combination with the DOS/4GW DOS extender. Hence all code runs in protected mode.

	C	Assembly
MDC-4	65 Kbit/s	89 Kbit/s

Table 1: Software performance of MDC-4 on a 33 MHz 80386DX based PC with a 64K memory cache using WATCOM C/386 9.0 in combination with the DOS/4GW DOS extender. Both versions use about 230K of memory.

6.2 Hardware Implementations

The DES algorithm has been designed for hardware implementations. Hence high performance is only attainable in hardware. With current submicron CMOS technology and a clock of 25 MHz a data rate of 90 Mbit/s has been achieved [VHVM88, VHVM91, Cry89, Pij92]. A faster clock of 40 MHz would allow for data rates of up to 150 Mbit/s. However at such speeds the critical path does not run through the DES module, but is situated in the I/O interface. The actual data rates will therefore be lower, but 50 to 60 Mbits/s is achievable.

If used in an MDC-4 implementation, the throughput of the existing chips will be significantly reduced, as for every application of the DES a new key has to be loaded. Moreover this new key is only available at the end of the previous step, which makes it impossible to load the key in advance. It nevertheless is expected that in dedicated hardware the speed mentioned above is attainable. Parallelism can be used to run both halves of MDC-4 simultaneously.

7 Guidelines for Software Implementation

The implementation in the C language given in Appendix A can be used as a guideline for software implementations. It also provides the test values given in Appendix B.

The speed of software implementations will be very low due to the fact that DES, including the (slow) key scheduling, has to be performed four times for each 8-byte message block. Since the the program spends almost all its time in the calls to DES, the speed of a software implementation will be determined by the efficiency of a DES encryption; optimizations other than in DES will not provide significant improvement of the speed.

The software implementation of DES, let alone an efficient implementation, is beyond the scope of this document. Only some guidelines for such an implementation are given. No claim on the completeness or relative importance is made, however.

Note that for each 64-bit block of the expanded message, four applications of DES are needed, each time with a different key. This is where most of the computation time is spent: even a highly optimized key-scheduling will be twice as slow as the encryption operation itself.

The key to an efficient software implementation of the DES will be the use of equivalent representations of the algorithm, see [DDFG83, DDGH84, FeKa89]. This allows a suitable reordering of the bits, the combination of several steps into one (e.g., the P-permutations and the expansion E) and the use of tables instead of the (bit)operations it is described in. Of course there is a time-memory trade-off: the more memory is used for tables, the more instructions can be replaced by table look-ups, and the faster the code will be. Moreover, combination of small tables into larger ones will reduce the number of look-ups, and hence further increase the speed.

However, one must be careful with this analysis. The speed of a computer is (for our purposes) determined by two things: the speed of the central processing unit (CPU) and the speed by which memory can be accessed. On many computers nowadays the speed of the CPU has become so enormous with respect to the speed of memory access, that a program with extensive memory access gets significantly slowed down. This means that a program with more instructions but fewer memory access might be faster than a program with less instructions but more memory access. A way around this problem is the use of a (small) amount of very fast (but very expensive) memory, so called cache memory. This way programs with extensive memory access, but which fit in cache memory are significantly faster than programs that only partially can use the benefit of this cache, because the amount of memory they need is larger than the size of the cache. Hence a program that is perfect for one computer (in the sense that it has minimal execution time) is therefore not necessarily optimal for another configuration. That is, there is no such thing as a single program being optimal for every configuration.

References

[BiSh91a] E. Biham and A. Shamir, "Differential Cryptanalysis of DES-like Cryptosystems," *Journal of Cryptology*, Vol. 4, no. 1, pp. 3-72, 1991.

[BiSh91b] E. Biham and A. Shamir, *Differential Cryptanalysis of the full 16-round DES*, Technical Report # 708, Technion - Israel Institute of Technology, Department of Computer Science, December 1991.

[CPMM90] D. Coppersmith, S. Pilpel, C.H. Meyer, S.M. Matyas, M.M. Hyden, J. Oseas, B. Brachtl, and M. Schilling, *Data Authentication Using Modification Detection Codes Based on a Public One Way Encryption Function*, U.S. Patent No. 4,908,861, March 13, 1990.

[Cry89] Cryptech, *CRY12C102 DES chip*, 1989.

[Dam89] I. B. 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, Berlin-Heidelberg-New York, pp. 416-427, 1990.

[DDFG83] M. Davio, Y. Desmedt, M. Fosseprez, R. Govaerts, J. Hulsbosch, P. Neutjens, P. Piret, J.-J. Quisquater, J. Vandewalle and P. Wouters, "Analytical Characteristics of the DES," in: *Advances in Cryptology - CRYPTO'83*, D. Chaum ed., Plenum Press, New York-London, pp. 171-202, 1984.

[DDGH84] M. Davio, Y. Desmedt, J. Goubert, F. Hoornaert and J.-J. Quisquater, "Efficient hardware and software implementations of the DES," in: *Advances in Cryptology - CRYPTO'84*, G.R. Blakely and D. Chaum eds., Lecture Notes in Computer Science Vol. 196, Springer-Verlag, Berlin-Heidelberg-New York, pp. 144-146, 1985.

[FeKa89] D. C. Feldmeier and P. R. Karn, "UNIX password security – Ten years later," in: *Advances in Cryptology - CRYPTO'89*, G. Brassard ed., Lecture Notes in Computer Science Vol. 435, Springer-Verlag, Berlin-Heidelberg-New York, pp. 44-63, 1990.

[Mat91] S.M. Matyas, "Key handling with control vectors," *IBM Systems Journal*, Vol. 30, no. 2, pp. 151-174, 1991.

[Mey89] C.H. Meyer, "Cryptography - A state of the art review," *Proceedings of COMPEURO'89, Proceedings VLSI and Computer Peripherals*, 3rd Annual European Computer Conference, Hamburg, Germany, pp. 150-154, May 8-12, 1989.

[MeSc88] C.H. Meyer and M. Schilling, "Secure Program Load with Manipulation Detection Code," *Proceedings of SECURICOM'88*, Paris, France, pp. 111-130, 1988.

[MoSi87] J.H. Moore and G.J. Simmons, "Cycle structure of the DES for keys having palindromic (or antipalindromic) sequences of round keys," *IEEE Transactions on Software Engineering*, vol. 13, pp. 262-273, 1987.

[NBS77] National Bureau of Standards, *Data Encryption Standard* , Federal Information Processing Standard, Publication 46, US Department of Commerce, January 1977.

[Pij92] Pijnenburg micro-electronics & software: *PCC100 Data Encryption Device*, 1992.

[QuDe89] J.-J. Quisquater and J.-P. Delescaille, "How easy is collision search? Applications to DES," in: *Advances in Cryptology - EUROCRYPT'89*, J.-J.Quisquater and J. Vandewalle eds., Lecture Notes in Computer Science, no. 434, Springer Verlag, Berlin-Heidelberg-New York, pp. 429-434, 1990.

[Riv90] R. L. Rivest, "The MD4 Message Digest Algorithm," in: *Advances in Cryptology - CRYPTO'90*, A.J. Menezes and S.A. Vanstone eds.,

Lecture Notes in Computer Science, no. 537, Springer Verlag, Berlin-Heidelberg-New York, pp. 303-311, 1991.

[VHVM88] I. Verbauwhede, J. Hoornaert, J. Vandewalle and H. De Man, "Security and performance optimization of a new DES data encryption chip," *IEEE Journal on Solid-State Circuits*, vol. 3, pp. 647-656, 1988.

[VHVM91] I. Verbauwhede, J. Hoornaert, J. Vandewalle and H. De Man, "ASIC Cryptographic Processor based on DES," *Proceedings of the EuroAsic'91 Conference*, Paris, France, May 1991.

A C Implementation of the Primitive

This appendix provides a ANSI C-implementation of the primitive MDC-4 and an example program that uses MDC-4 to hash messages. This program can be used for testing purposes as well, as it can provide the test values of Appendix B.

Use of this implementation The file `mdc2.c` contains the functions `MDCinit`, `compress` and `MDCfinish`. `MDCinit` performs initialization of $H1$ and $H2$, `compress` applies compress, `MDCfinish` pads the message, appends the length and compresses the resulting blocks. The header file `mdc2.h` contains the function prototypes and some macros.

The test program `hashtest` can be compiled as follows. First, compile `mdc2.c`, `hashtest.c` and a file containing an implementation of DES with an (ANSI) C compiler. Next, link those three files.

This implementation has been tested on a wide variety of environments, so it should be portable or at worst easy to port. The testing environments include MS-DOS (4.2 and 5.0 on both 80286 and 80386 processors) with several compilers, VAX/VMS, RISC ULTRIX, ConvexOs and Macintosh.

Note that VAX/VMS does not allow the `run` command to pass arguments to an executable. A patch for this is given in the comment of the `main()` function in `hashtest.c`.

A.1 The Header File

```
/*********************************************************************/
/*  file: mdc4.h                                                     */
/*                                                                   */
/*  description: header file for MDC-4, a sample C-implementation    */
/*                                                                   */
/*  copyright (C)                                                    */
/*           Centre for Mathematics and Computer Science, Amsterdam  */
/*           Siemens AG                                              */
/*           Philips Crypto BV                                       */
/*           PTT Research, the Netherlands                           */
/*           Katholieke Universiteit Leuven                          */
/*           Aarhus University                                       */
/*  1992, All Rights Reserved                                        */
/*                                                                   */
/*  date:    06/25/92                                                */
/*  version: 1.0                                                     */
/*                                                                   */
/*********************************************************************/

#ifndef  MDC4H              /* make sure this file is read only once */
#define  MDC4H

/*********************************************************************/

/* typedef 8, 16 and 32 bit types, resp.  */
/* adapt these, if necessary,
   for your operating system and compiler */
typedef    unsigned char        byte;
typedef    unsigned short       word;
typedef    unsigned long        dword;


/*********************************************************************/

/* function prototypes */

byte *DES(byte *input, byte *key);
/*
 * returns output of DES with given input and key.
 * source is not provided by RIPE!
 */

void MDCinit(byte *H1, byte *H2);
/*
 *  initializes H1 and H2
 */

void compress(byte *H1, byte *H2, byte *X);
/*
 *  the compression function.
 *  transforms H1, H2 using message bytes X[0] through X[7]
```

```
 */

void MDCfinish(byte *H1, byte *H2, byte *strptr,
               dword lswlen, dword mswlen);
/*
 *  pads, appends length and compresses the last block(s)
 *  note: length in bits == 8 * (lswlen + 2^32 mswlen).
 *  note: there are (lswlen mod 8) unprocessed bytes left in strptr.
 */

#endif  /* MDC4H */

/************************ end of file mdc4.h ************************/
```

A.2 C Source Code for MDC-4

```
/*********************************************************************/
/*  file: mdc4.c                                                     */
/*                                                                   */
/*  description: A sample C-implementation of the MDC-4              */
/*           hash-function.  The function DES() is external;         */
/*           it is not provided by RIPE.                             */
/*                                                                   */
/*  copyright (C)                                                    */
/*           Centre for Mathematics and Computer Science, Amsterdam */
/*           Siemens AG                                              */
/*           Philips Crypto BV                                       */
/*           PTT Research, the Netherlands                           */
/*           Katholieke Universiteit Leuven                          */
/*           Aarhus University                                       */
/*  1992, All Rights Reserved                                        */
/*                                                                   */
/*  date:    06/25/92                                                */
/*  version: 1.0                                                     */
/*                                                                   */
/*********************************************************************/


/*  header files */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "mdc4.h"

/*********************************************************************/

void MDCinit(byte *H1, byte *H2)
{
   int   i;

   for (i=0; i<8; i++) {
      H1[i] = 0x52;
      H2[i] = 0x25;
   }

   return;
}

/*********************************************************************/

static void swap(byte *x, byte *y)
{
   byte tmp;

   tmp = *x;
   *x = *y;
   *y = tmp;
```

```
   return;
}

/*****************************************************************************/

static void E1xor(byte *input, byte *key, byte *output)
{
   byte        *desres;
   byte         oldbits;
   int          i;

   /* store value of the bits k_1 and k_2 */
   oldbits = key[0] & 96;
   /* set those bits to 1 resp. 0 */
   key[0] ^= oldbits ^ 64;

   /* one-way function: */
   desres = DES(input, key);
   for (i=0; i<8; i++) {
      output[i] = input[i] ^ desres[i];
   }

   if (key != output) {
      /* reset the two bits k_1 and k_2 */
      key[0] ^= oldbits ^ 64;
   }

   return;
}

/*****************************************************************************/

static void E2xor(byte *input, byte *key, byte *output)
{
   byte        *desres;
   byte         oldbits;
   int          i;

   /* store value of the bits k_1 and k_2 */
   oldbits = key[0] & 96;
   /* set those bits to 0 resp. 1 */
   key[0] ^= oldbits ^ 32;

   /* one-way function: */
   desres = DES(input, key);
   for (i=0; i<8; i++) {
      output[i] = input[i] ^ desres[i];
   }

   if (key != output) {
      /* reset the two bits k_1 and k_2 */
      key[0] ^= oldbits ^ 32;
```

```
   }

   return;
}

/*********************************************************************/

void compress(byte *H1, byte *H2, byte *X)
{
   byte  temp1[8], temp2[8];
   int   i;

   /* apply E1xor and E2xor to the halves */
   E1xor(X, H1, temp1);
   E2xor(X, H2, temp2);

   /* interchange right halves */
   for (i=4; i<8; i++) {
      swap(temp1+i, temp2+i);
   }

   /* apply E1xor and E2xor to the halves */
   E1xor(X, temp1, H2);
   E2xor(X, temp2, H1);
   /* note: H1 and H2 are interchanged, so: */

   /* interchange _left_ halves */
   for (i=0; i<4; i++) {
      swap(H1+i, H2+i);
   }

   return;
}

/*********************************************************************/

void MDCfinish(byte *H1, byte *H2, byte *input,
               dword lswlen, dword mswlen)
{
   int          i;                              /* counter         */
   byte         X[8];                           /* message bytes   */

   /* get lswlen mod 8 unprocessed bytes */
   for (i=0; i<(lswlen&7); i++) {
      X[i] = input[i];
   }

   /* append the bit m_n == 1 */
   X[lswlen&7] = 0x80;
   /* ... and zero bits: */
   for (i=(lswlen&7)+1; i<8; i++) {
      X[i] = 0;
   }
```

```
   compress(H1, H2, X);

   /* append length in bits; MSB to LSB */
   X[0] = mswlen >> 21;
   X[1] = mswlen >> 13;
   X[2] = mswlen >> 5;
   X[3] = (mswlen << 3) ^ (lswlen >> 29);
   X[4] = lswlen >> 21;
   X[5] = lswlen >> 13;
   X[6] = lswlen >> 5;
   X[7] = lswlen << 3;

   compress(H1, H2, X);

   return;
}

/************************* end of file mdc4.c **********************/
```

A.3 An Example Program

This section gives the listing of an example program. By means of command line options, several different tests can be performed (see the top of the file). Test values can be found in Appendix B. The file `test.bin` should contain the string `abc` and nothing else. (Make sure no newline of linefeed is appended by your editor.)

```
/*************************************************************************/
/*  file: hashtest.c                                                     */
/*                                                                       */
/*  description: test file for MDC-4                                     */
/*           DES() is an external function, not provided by RIPE.        */
/*                                                                       */
/*  command line arguments:                                              */
/*           filename  -- compute hash code of file read binary          */
/*           -sstring  -- print string & hashcode                        */
/*           -t        -- perform time trial                             */
/*           -a        -- execute standard test suite, ASCII input       */
/*                        and binary input from file test.bin            */
/*           -x        -- execute standard test suite, hexadecimal       */
/*                        input read from file test.hex                  */
/*                                                                       */
/*  copyright (C)                                                        */
/*           Centre for Mathematics and Computer Science, Amsterdam      */
/*           Siemens AG                                                  */
/*           Philips Crypto BV                                           */
/*           PTT Research, the Netherlands                               */
/*           Katholieke Universiteit Leuven                              */
/*           Aarhus University                                           */
/*  1992, All Rights Reserved                                            */
/*                                                                       */
/*  date:    06/25/92                                                    */
/*  version: 1.0                                                         */
/*                                                                       */
/*************************************************************************/


#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <string.h>
#include "mdc4.h"

#ifndef CLK_TCK
#define CLK_TCK CLOCKS_PER_SEC   /* this is a hack for c89 compiler */
#endif                           /* (DEC C for ULTRIX on RISC v1.0) */

#define TEST_BLOCK_SIZE 100
#define TEST_BLOCKS 1000

/* number of test bytes = TEST_BLOCK_SIZE * TEST_BLOCKS */
static long TEST_BYTES = (long)TEST_BLOCK_SIZE * (long)TEST_BLOCKS;
```

```
/*****************************************************************/

byte *MDC4(byte *message)
/*
 * returns MDC-4(message)
 * message should be a string terminated by '\0'
 */
{
   static byte   H1[16];       /* contains H1_i and H2_i            */
   static byte  *H2 = H1 + 8;  /* points to right half              */
   dword         length;       /* length in bytes of message        */
   dword         nbytes;       /* # of bytes not yet processed      */
   byte         *strptr;       /* points to the current mess. block */

   /* initialize */
   MDCinit(H1, H2);
   strptr = message;                /* strptr points to first block */
   length = (dword)strlen((char *)message);
   nbytes = length;

   /* process message in 8-byte chunks */
   while (nbytes > 7) {
      compress(H1, H2, strptr);
      strptr += 8;
      nbytes -= 8;                        /* 8 bytes less to process */
   }                                      /* length mod 8 bytes left */

   /* finish: */
   MDCfinish(H1, H2, strptr, length, 0);

   return (byte *)H1;
}

/*****************************************************************/

byte *MDC4binary(char *fname)
/*
 * returns MDC-4(message in file fname)
 * fname is read as binary data.
 */
{
   FILE         *mf;                /* pointer to file <fname>      */
   static byte   H1[16];            /* contains H1_i and H2_i       */
   static byte  *H2 = H1 + 8;       /* points to right half         */
   byte          data[1024];        /* contains current mess. block */
   dword         nbytes;            /* length of this block         */
   word          i;                 /* counter                      */
   dword         length[2];         /* length in bytes of message   */
   dword         offset;            /* # of bytes unprocessed at    */
              /* call of MDCfinish */

   /* initialize */
   if ((mf = fopen(fname, "rb")) == NULL) {
```

```
      fprintf(stderr, "\nMDC4binary: cannot open \"%s\".\n", fname);
      exit(1);
   }
   MDCinit(H1, H2);
   length[0] = 0;
   length[1] = 0;

   while ((nbytes = fread(data, 1, 1024, mf)) != 0) {
      /* process all complete blocks */
      for (i=0; i<(nbytes>>3); i++) {
         compress(H1, H2, data+(i<<3));
      }

      /* update length[] */
      if (length[0] + nbytes < length[0])
         length[1]++;                  /* overflow to msw of length */
      length[0] += nbytes;
   }

   /* finish: */
   offset = length[0] & 0x3F8;   /* extracts bits 3 to 10 inclusive */
   MDCfinish(H1, H2, data+offset, length[0], length[1]);

   fclose(mf);

   return (byte *)H1;
}

/*************************************************************************/


byte *MDC4hex(char *fname)
/*
 * returns MDC-4(message in file fname)
 * fname should contain the message in hex format;
 * first number of bytes, then the bytes in hexadecimal.
 */
{
   FILE         *mf;               /* pointer to file <fname>      */
   static byte   H1[16];           /* contains H1_i and H2_i       */
   static byte  *H2 = H1 + 8;      /* points to right half         */
   byte          data[8];          /* contains current mess. block */
   dword         nbytes;           /* length of the message        */
   word          i, j;             /* counters                     */
   int           val;              /* temp for reading from file   */

   /* initialize */
   if ((mf = fopen(fname, "r")) == NULL) {
      fprintf(stderr, "\nMDC4hex: cannot open file \"%s\".\n",
              fname);
      exit(1);
   }
   MDCinit(H1, H2);
```

```
   fscanf(mf, "%x", &val);
   nbytes = val;
   i = 0;
   while (nbytes - i > 7) {
      /* read and process complete block */
      for (j=0; j<8; j++) {
         fscanf(mf, "%x", &val);
         data[j] = (byte)val;
      }
      compress(H1, H2, data);
      i += 8;
   }

   /* read last nbytes-i bytes: */
   j = 0;
   while (i<nbytes) {
      fscanf(mf, "%x", &val);
      data[j++] = (byte)val;
      i++;
   }

   /* finish */
   MDCfinish(H1, H2, data, nbytes, 0);

   fclose(mf);

   return (byte *)H1;
}

/*************************************************************************/

void speedtest(void)
/*
 * A time trial routine, to measure the speed of MDC-4.
 * Measures processor time required to process TEST_BLOCKS times
 *  a message of TEST_BLOCK_SIZE characters.
 */
{
   clock_t       t0, t1;
   static byte   H1[16];            /* contains H1_i and H2_i       */
   static byte  *H2 = H1 + 8;       /* points to right half         */
   byte          data[TEST_BLOCK_SIZE];
   word          i, j;

  /* initialize test data */
  for (i=0; i<TEST_BLOCK_SIZE; i++)
     data[i] = (byte)i;

  printf ("MDC4 time trial. Processing %ld characters...\n", TEST_BYTES);

  /* start timer */
  t0 = clock();
```

```
  /* process data */
  MDCinit(H1, H2);
  for (i=0; i<TEST_BLOCKS; i++) {
     for (j=0; j<TEST_BLOCK_SIZE; j+=8) {
        compress(H1, H2, data+(j>>3));
     }
  }
  MDCfinish(H1, H2, data, TEST_BYTES, 0);

  /* stop timer, get time difference */
  t1 = clock();
  printf("\nTest input processed in %g seconds.\n",
         ((double)(t1-t0)/(double)CLK_TCK));
  printf("Characters processed per second: %g\n",
         (double)CLK_TCK*TEST_BYTES/((double)t1-t0));

  printf("\nhashcode: ");
  for (i=0; i<16; i++)
     printf("%02x", H1[i]);

  return;
}

/*********************************************************************/

void testascii (void)
/*
 *   standard test suite, ASCII input
 */
{
   int      i;
   byte    *hashcode;

   printf("\nMDC4 test suite results (ASCII):\n");

   hashcode = MDC4((byte *)"");
   printf("\n\nmessage: \"\"  (empty string)\nhashcode: ");
   for (i=0; i<16; i++)
      printf("%02x", hashcode[i]);

   hashcode = MDC4((byte *)"a");
   printf("\n\nmessage: \"a\"\nhashcode: ");
   for (i=0; i<16; i++)
      printf("%02x", hashcode[i]);

   hashcode = MDC4((byte *)"abc");
   printf("\n\nmessage: \"abc\"\nhashcode: ");
   for (i=0; i<16; i++)
      printf("%02x", hashcode[i]);

   hashcode = MDC4((byte *)"message digest");
   printf("\n\nmessage: \"message digest\"\nhashcode: ");
```

```
   for (i=0; i<16; i++)
      printf("%02x", hashcode[i]);

   hashcode = MDC4((byte *)"abcdefghijklmnopqrstuvwxyz");
   printf("\n\nmessage: \"abcdefghijklmnopqrstuvwxyz\"\nhashcode: ");
   for (i=0; i<16; i++)
      printf("%02x", hashcode[i]);

   hashcode = MDC4((byte *)
      "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789");
   printf(
      "\n\nmessage: A...Za...z0...9\nhashcode: ");
   for (i=0; i<16; i++)
      printf("%02x", hashcode[i]);

   hashcode = MDC4((byte *)"12345678901234567890123456789012345678901\
2345678901234567890123456789012345678901");
   printf("\n\nmessage: 8 times \"1234567890\"\nhashcode: ");
   for (i=0; i<16; i++)
      printf("%02x", hashcode[i]);

   /* Contents of binary created file test.bin are "abc" */
   printf("\n\nmessagefile (binary): test.bin\nhashcode: ");
   hashcode = MDC4binary("test.bin");
   for (i=0; i<16; i++)
      printf("%02x", hashcode[i]);

   return;
}

/*************************************************************************/

void testhex (void)
/*
 *   standard test suite, hex input, read from files
 */
{
   int i;
   byte *hashcode;

   printf("\nMDC4 test suite results (hex):\n");

   hashcode = MDC4hex("test1.hex");
   printf("\n\nfile test1.hex; hashcode: ");
   for (i=0; i<16; i++)
      printf("%02x", hashcode[i]);

   hashcode = MDC4hex("test2.hex");
   printf("\n\nfile test2.hex; hashcode: ");
   for (i=0; i<16; i++)
      printf("%02x", hashcode[i]);

   hashcode = MDC4hex("test3.hex");
```

```
    printf("\n\nfile test3.hex; hashcode: ");
    for (i=0; i<16; i++)
       printf("%02x", hashcode[i]);

    hashcode = MDC4hex("test4.hex");
    printf("\n\nfile test4.hex; hashcode: ");
    for (i=0; i<16; i++)
       printf("%02x", hashcode[i]);

    hashcode = MDC4hex("test5.hex");
    printf("\n\nfile test5.hex; hashcode: ");
    for (i=0; i<16; i++)
       printf("%02x", hashcode[i]);

    hashcode = MDC4hex("test6.hex");
    printf("\n\nfile test6.hex; hashcode: ");
    for (i=0; i<16; i++)
       printf("%02x", hashcode[i]);

    hashcode = MDC4hex("test7.hex");
    printf("\n\nfile test7.hex; hashcode: ");
    for (i=0; i<16; i++)
       printf("%02x", hashcode[i]);

    return;
}

/*************************************************************************/

main (int argc, char *argv[])
/*
 *  main program. calls one or more of the test routines depending
 *  on command line arguments. see the header of this file.
 *
 *  (For VAX/VMS, do: HASHTEST :== $<pathname>HASHTEST.EXE
 *   at the command prompt (or in login.com) first.
 *   (The run command does not allow command line args.)
 *   The <pathname> must include device, e.g., "DSKD:".)
 */
{
  int   i, j;
  byte *hashcode;

   if (argc == 1) {
      fprintf(stderr, "hashtest: no command line arguments supplied.\n");
      exit(1);
   }
   else {
      for (i = 1; i < argc; i++) {
         if (argv[i][0] == '-' && argv[i][1] == 's') {
            printf("\n\nmessage: %s", argv[i]+2);
            hashcode = MDC4((byte *)argv[i] + 2);
            printf("\nhashcode: ");
```

```
            for (j=0; j<16; j++)
               printf("%02x", hashcode[j]);
         }
         else if (strcmp (argv[i], "-t") == 0)
            speedtest ();
         else if (strcmp (argv[i], "-a") == 0)
            testascii ();
         else if (strcmp (argv[i], "-x") == 0)
            testhex ();
         else {
            hashcode = MDC4binary (argv[i]);
            printf("\n\nmessagefile (binary): %s", argv[i]);
            printf("\nhashcode: ");
            for (j=0; j<16; j++)
               printf("%02x", hashcode[j]);
         }
      }
   }
   printf("\n");

   return 0;
}

/********************** end of file hashtest.c **********************/
```

B Test Values

Below, the files `test1.hex` up to `test7.hex` as used by the test `hashtest -x` are listed. The format of those files is as follows. All numbers are hexadecimal; all numbers except possibly the first one are one byte long. The first number represents the number of bytes in the message to be hashed; it is followed by (at least) this number of bytes. For example, `test3.hex` represents the message consisting of the three bytes 0x61, 0x62 and 0x63. This is the string "abc" in ASCII.

`test1.hex`:

```
0
```

`test2.hex`:

```
1
61
```

`test3.hex`:

```
3
61 62 63
```

`test4.hex`:

```
e
6d 65 73 73 61 67 65 20 64 69 67 65 73 74
```

`test5.hex`:

```
1a
61 62 63 64 65 66 67 68 69 6a 6b 6c 6d 6e 6f 70
71 72 73 74 75 76 77 78 79 7a
```

`test6.hex`:

```
3e
41 42 43 44 45 46 47 48 49 4a 4b 4c 4d 4e 4f 50
51 52 53 54 55 56 57 58 59 5a 61 62 63 64 65 66
67 68 69 6a 6b 6c 6d 6e 6f 70 71 72 73 74 75 76
77 78 79 7a 30 31 32 33 34 35 36 37 38 39
```

`test7.hex`:

```
50
31 32 33 34 35 36 37 38 39 30
31 32 33 34 35 36 37 38 39 30
31 32 33 34 35 36 37 38 39 30
31 32 33 34 35 36 37 38 39 30
31 32 33 34 35 36 37 38 39 30
31 32 33 34 35 36 37 38 39 30
31 32 33 34 35 36 37 38 39 30
31 32 33 34 35 36 37 38 39 30
```

The following test values were obtained by running **hashtest -x**. If ASCII encoding is used, **hashtest -a** should provide the same answers, followed by the result of hashing "abc" again. The latter only holds if the file **test.bin** exists and contains nothing but this string.

```
MDC4 test suite results (hex):


file test1.hex; hashcode: 14131f5dadbc2cc4e4d4c8dcef91d462

file test2.hex; hashcode: f15cc877d7f3c929330f896222410f01

file test3.hex; hashcode: e0abd233a1be9650713609decc1ef62b

file test4.hex; hashcode: 971add04ab1c607dcfa8c855aac15a16

file test5.hex; hashcode: 600cf23bd5eef96472825a34e6bd8236

file test6.hex; hashcode: 7514607cc53c00fb1ef10aeab90cdadf

file test7.hex; hashcode: dee26cddf118817974b89028c7de5999
```

Chapter 3

RIPEMD

Contents

1 Introduction

This chapter describes the integrity primitive RIPEMD. This is a so-called *message-digest algorithm* or *hash function* that compresses messages of arbitrary length to a 128-bit output block, that is called the *fingerprint*, *hashcode*, *hash value* or *message-digest* of the message. It is conjectured that it is computationally infeasible to produce two messages having the same hashcode, or to produce any messages having a given prespecified target hashcode. Hash functions with these properties are used in message authentication applications such as the protection of the integrity and the origin of data stored or transmitted using secret-key or public-key techniques (see Section 4 in this chapter and Part II of this report).

RIPEMD is an extension of the MD4 message-digest algorithm [Riv90]. MD4 consists of two parts: first, the message is expanded slightly to obtain an appropriate length for further processing. Next, a compression function is used iteratively to compress this expanded message to 128 bits. RIPEMD has the same structure, but its compression function consists of two parallel copies of MD4's compression function, identical but for some internal constants. The results of both copies are combined to yield the output of RIPEMD's compression function. The reason for this more complicated design lies in the fact that successful attacks exist on two round versions of MD4 [BoBo91, Mer90]. Because of these attacks, the designers of MD4 proposed a modified version, which is called MD5 [RiDu91]. However, there are indications that this redesign has introduced new weaknesses in the algorithm [Boe92]. The RIPEMD extension of MD4 is also from [Boe92].

The design of the RIPEMD algorithm is such that it is very suitable for software implementation on 32-bit machines. Since no substitution tables are used, the algorithm can be coded quite compactly. The large input block sizes however, do not favor compact hardware realizations.

The structure of this chapter is as follows. In order to avoid any ambiguities in the description of the primitive, the notation and definitions used in this chapter are fixed in Section 2. Section 3 contains a description of the primitive and in Section 4 the possible modes of use of the primitive are considered. The security aspects of the primitive are discussed in Section 5. These include the claimed properties and the results of statistical and algebraic evaluations of the primitive. Finally, in Section 6 the performance aspects of RIPEMD are considered, and Section 7 gives some guidelines for software implementation.

This chapter has two appendices. Appendix A contains a straightforward software implementation of RIPEMD in the programming language C and in Appendix B test values for the primitive are given.

2 Definitions and Notation

2.1 Introduction

In order to obtain a clear description of the primitive, the notation and definitions used in this chapter are fully described in this section. These include the representation of the numbers in the description, and the operations, functions and constants used by the primitive.

2.2 General

The symbol ":=" is used for the assignment of a value or a meaning to a variable or symbol. That is, $a := b$ either means that the variable a gets the value of the variable b, or it means that a is defined as "b". It will be obvious from the context which meaning is intended.

The equality-sign "=" is used for equality only. That is, it indicates that the two entities on either side are equal.

Note that in C-source code, '=' denotes assignment, while comparison is denoted by '=='.

An ellipsis ("...") denotes an implicit enumeration. For example, "$i = 0, 1, \ldots, n$" is meant to represent the sentence "for $i = 0$, $i = 1$, and so on, up to $i = n$".

In pseudo-codes, comment is indicated as in the C-language, viz. by enclosing it between "/*" and "*/".

2.3 Representation of Numbers

In this chapter a *byte* is defined as an 8-bit quantity and a *word* as a 32-bit quantity. A byte is considered to be a nonnegative integer. That is, it can take on the values 0 through $2^8 - 1 = 255$. Likewise, a word is considered to be a nonnegative integer, hence it takes on the values 0 through $2^{32} - 1 = 4294967295$. The value of a word can be given in decimal as well as in hexadecimal form. In the latter case the number is written as '0x' followed immediately by at most 8 hexadecimal digits, the most significant one first. For example, the hexadecimal representation of the 32-bit number 4023233417 is `0xefcdab89`.

A sequence of $8n$ bits $b_0, b_1, \ldots, b_{8n-1}$ is interpreted as a sequence of n bytes in the following way. Each group of 8 consecutive bits is considered as a byte, the first bit of such a group being the most significant bit of that byte. Hence,

$$B_i := b_{8i}2^7 + b_{8i+1}2^6 + \cdots + b_{8i+7}, \quad i = 0, 1, \ldots, n-1. \tag{1}$$

A sequence of $4l$ bytes $B_0, B_1, \ldots, B_{4l-1}$ is interpreted as a sequence of words $W_0, W_1, \ldots, W_{l-1}$ in the following way. Each group of 4 consecutive bytes is considered as

a word, the first byte of such a word being the *least* significant byte of that word (!). Hence,

$$W_i := B_{4i+3}(256)^3 + B_{4i+2}(256)^2 + B_{4i+1}(256) + B_{4i}, \quad i = 0, 1, \ldots, l-1. \quad (2)$$

Notice the difference in convention between the notations (1) and (2). This is done for reasons of performance, see [Riv90] and Section 7.
In accordance with the notations above, the bits of a word W are denoted as

$$W = (w_0, w_1, \ldots, w_{31}), \quad (3)$$

where

$$W = \sum_{i=0}^{3} \sum_{j=0}^{7} w_{8i+j} 2^{8i+(7-j)}. \quad (4)$$

In this chapter words are always denoted by uppercase letters and the bits of this word by the corresponding lowercase letter with indices as in Equation (3). Likewise, bytes are indicated by uppercase letters and the bits that constitute this byte by lowercase letters with indices as in Equation (1). The ordering of bytes in a word is given by Equation (2).

2.4 Definitions and Basic Operations

- A *string* is a sequence of bits. If X is a string consisting of n bits, then those bits are denoted from left to right by $x_0, x_1, \ldots, x_{n-2}, x_{n-1}$.
- For a string X the *length* of X is denoted as $|X|$. That is, $|X|$ is the number of bits in the string X. If $|X| = n$, then X is said to be an n-bit string.
- Strings of length 32 will also be considered as words according to the representation defined by (4), and vice versa. Hence, if X is a string of length 32, then the corresponding word is equal to

  $$X = \sum_{i=0}^{3} \sum_{j=0}^{7} x_{8i+j} 2^{8i+(7-j)}.$$

  Note that the same symbol is used for both the string and the corresponding word. It will be clear from the context which representation is intended.
- For an integer N, the *length* of N is defined as the length of the shortest binary representation of N. This is the representation with most significant bit equal to 1. (All "leading zeros" are removed.) The length of N is denoted as $|N|$.
- For a nonnegative integer A and a positive integer B, the numbers A div B and A mod B are defined as the nonnegative integers Q, respectively R, such that

  $$A = QB + R \quad \text{and} \quad 0 \leq R < B.$$

  That is, A mod B is the *remainder*, and A div B is the *quotient* of an integer division of A by B.

- For two words X and Y, the words $U = X \oplus Y$, $V = X \wedge Y$ and $W = X \vee Y$ are defined as the bitwise *XOR*, *AND* and *OR* of X and Y, respectively. Hence, according to Equation (3):

$$\begin{aligned} u_i &:= (x_i + y_i) \bmod 2 \\ v_i &:= x_i y_i \qquad\qquad (i = 0, 1, \ldots, 31). \\ w_i &:= (x_i + y_i - x_i y_i) \bmod 2 \end{aligned}$$

- For a word X, the word $W = \overline{X}$ is defined as the bitwise *complement* of X, hence

$$W = \overline{X} := X \oplus \texttt{0xffffffff},$$

or according to Equation (3):

$$w_i := (x_i + 1) \bmod 2, \quad i = 0, 1, \ldots, 31.$$

- for a word X and an integer $0 \leq s < 32$, the word $W = X \lll s$ is defined as the result of (cyclicly) rotating X over s bits to the left, hence

$$W = X \lll s := (X 2^s + (X \text{ div } 2^{32-s})) \bmod 2^{32}.$$

- The notation "$X \equiv Y \pmod N$" (X is equivalent to Y modulo N) is used to indicate that $X \bmod N = Y \bmod N$.

2.5 Functions and Operations used in RIPEMD

RIPEMD uses three basic functions that each map three words onto a word. These functions are defined as follows.

$$F(X,Y,Z) := (X \wedge Y) \vee (\overline{X} \wedge Z) \tag{5}$$
$$G(X,Y,Z) := (X \wedge Y) \vee (X \wedge Z) \vee (Y \wedge Z) \tag{6}$$
$$H(X,Y,Z) := X \oplus Y \oplus Z \tag{7}$$

For each bit of X, the function F *selects* a bit of either Y or Z, depending on the bit of X: if the bit X_i is 1, then the corresponding bit Y_i is selected, else Z_i is selected ($i = 0, 1, \ldots, 31$).

The function G is the bitwise *majority* of the bits of X, Y and Z: if two or three of the bits of X_i, Y_i and Z_i are 1, then the corresponding bit G_i of $G(X,Y,Z)$ is 1, else $G_i = 0$.

The function H is the bitwise *parity* (or XOR) of X, Y and Z: a bit H_i of $H = H(X,Y,Z)$ is 1 if an odd number of the bits X_i, Y_i and Z_i is 1, else $H_i = 0$.

Those three functions are used in six other (higher level) operations. These are used to process a message consisting of 16 words X_0, X_1, ..., X_{15}. For words A, B, C and D, and integers $0 \leq k < 16$ and $0 \leq s < 32$ these operations are defined as follows.

- $FF(A, B, C, D, X_k, s)$ denotes the operation

$$A := ((A + F(B, C, D) + X_k) \bmod 2^{32}) \lll s. \tag{8}$$

- $GG(A, B, C, D, X_k, s)$ denotes the operation

$$A := ((A + G(B, C, D) + X_k + \texttt{0x5a827999}) \bmod 2^{32}) \lll s. \tag{9}$$

- $HH(A, B, C, D, X_k, s)$ denotes the operation

$$A := ((A + H(B, C, D) + X_k + \texttt{0x6ed9eba1}) \bmod 2^{32}) \lll s. \tag{10}$$

- $FFF(A, B, C, D, X_k, s)$ denotes the operation

$$A := ((A + F(B, C, D) + X_k + \texttt{0x50a28be6}) \bmod 2^{32}) \lll s. \tag{11}$$

- $GGG(A, B, C, D, X_k, s)$ denotes the operation

$$A := ((A + G(B, C, D) + X_k) \bmod 2^{32}) \lll s. \tag{12}$$

- $HHH(A, B, C, D, X_k, s)$ denotes the operation

$$A := ((A + H(B, C, D) + X_k + \texttt{0x5c4dd124}) \bmod 2^{32}) \lll s. \tag{13}$$

That is, all six functions add the result of an application of F, G or H to (B, C, D), a message word X_k and possibly a constant to A and rotate the result over s positions to the left.

The four constants used in these operations are not randomly chosen; they are the integer part of $2^{30}\sqrt{2}$, $2^{30}\sqrt{3}$, $2^{30}\sqrt[3]{2}$ respectively $2^{30}\sqrt[3]{3}$. However, there is no specific reason for this choice.

3 Description of the Primitive

3.1 Global Structure of the Primitive

As stated in the introduction of this chapter, RIPEMD is a hash function that maps a message M consisting of an arbitrary number of bits onto a 128-bit block RIPEMD(M). The basis of RIPEMD is the compression function compress. This function compresses a 20-word input to a 4-word output. This function is used in the following way.

First, the message is expanded to an appropriate length and represented by a sequence of words. Then, starting with a 4-word *initial vector*, the resulting sequence of words is compressed by repeatedly appending sixteen message words and compressing the resulting twenty words to four words by applying compress until the message is exhausted. Thus, a 4-word result can be obtained. Below, this is explained in detail.

3.1.1 Outline of compress

The compression function compress accepts an input consisting of four words A, B, C, D and sixteen message words X_0, X_1, ..., X_{15}. It produces four output words denoted as:

$$\mathsf{compress}((A, B, C, D), (X_0, X_1, \ldots, X_{15})). \tag{14}$$

The function compress consists of two separate halves, followed by a final step combining the results of both halves into four words. Each of the halves can be decomposed into three rounds. Each of these rounds consists of sixteen steps. These steps, finally, consist of the application of one of the six operations *FF*, *GG*, *HH*, *FFF*, *GGG* and *HHH*; a different operation for each round. See also Figure 1.

In other words: the input is passed to both halves. Both halves then independently transform, in three rounds of sixteen steps each, a copy of the 4-tuple (A, B, C, D). In a final step, the results of both independent transformations are combined into the result of compress.

3.1.2 Outline of RIPEMD

Let $M = (m_0, m_1, \ldots, m_{n-1})$ be a message consisting of n bits. The hashcode RIPEMD(M) of M is computed as follows, see also Figure 2.

expansion: First, M is expanded to a message X consisting of $16N-2$ words X_0, X_1, ..., X_{16N-3}, where $N = ((n+64) \text{ div } 512)+1$. That is, the message is expanded such that it is exactly 64 bits shy of being a multiple of 512 bits long. (Stated in words: 2 words shy of a multiple of 16 words long.)

Next, X is completed to a multiple of 512 bits (16 words) by appending two words X_{16N-2} and X_{16N-1} representing the length of the original message.

compression: The resulting message X is compressed using the function compress.

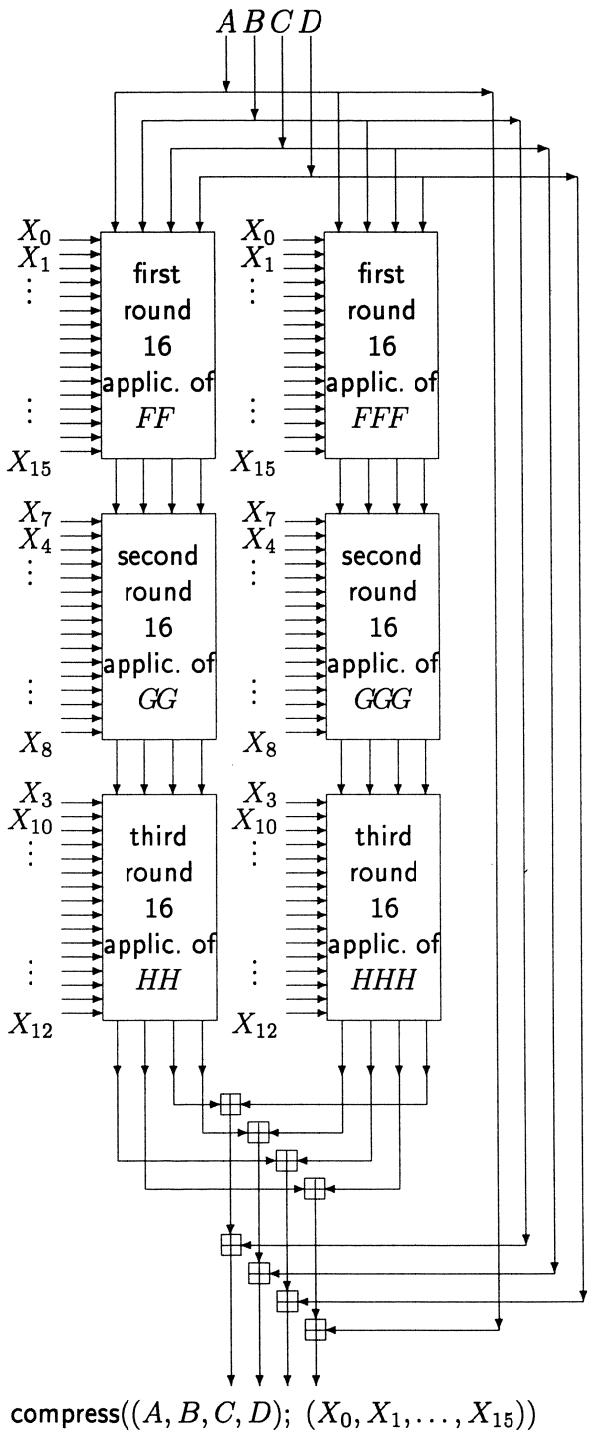

Figure 1: Outline of the compression function compress. The input to the right half is identical to that in the left half, only the constants used are different. The symbol $\boxplus$ denotes addition modulo 2^{32}.

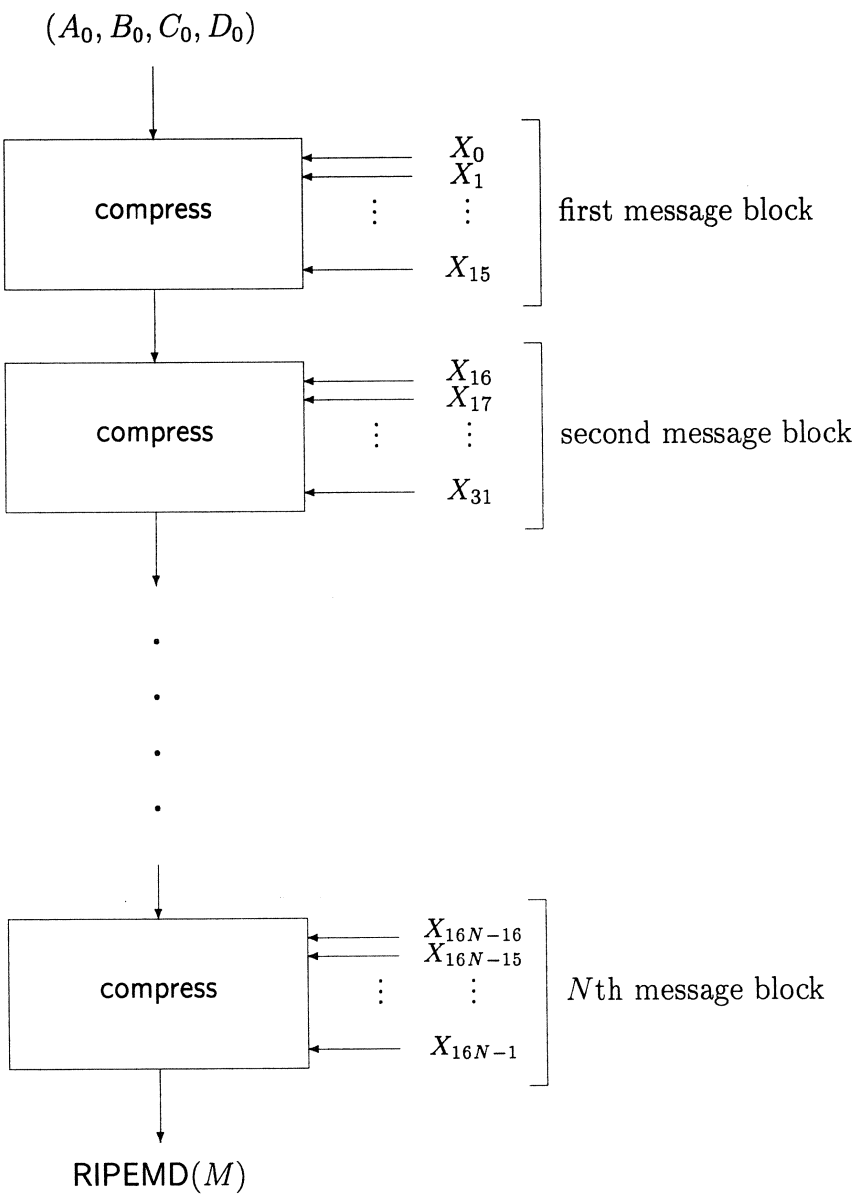

Figure 2: Outline of RIPEMD. The message M is expanded first to X which is a multiple of 16 words long. Then X is processed as in this picture. The final result is RIPEMD(M).

Define the words A_0, B_0, C_0, D_0 as

$$\begin{aligned} A_0 &:= \texttt{0x67452301} \\ B_0 &:= \texttt{0xefcdab89} \\ C_0 &:= \texttt{0x98badcfe} \\ D_0 &:= \texttt{0x10325476} \end{aligned}$$

For $i = 0, 1, \ldots, N-1$, the words A_{i+1}, B_{i+1}, C_{i+1} and D_{i+1} are computed as follows from the words A_i, B_i, C_i and D_i and sixteen message words X_{16i+j}, $j = 0, 1, \ldots, 15$:

$$\begin{aligned} &(A_{i+1}, B_{i+1}, C_{i+1}, D_{i+1}) \\ &\quad := \mathsf{compress}((A_i, B_i, C_i, D_i); (X_{16i}, X_{16i+1}, \ldots, X_{16i+15})), \end{aligned}$$

That is, each 16-word message block is used to transform (A_i, B_i, C_i, D_i) into $(A_{i+1}, B_{i+1}, C_{i+1}, D_{i+1})$ for the current value of i.

Finally, the hash-value $\mathsf{RIPEMD}(M)$ is equal to (A_n, B_n, C_n, D_n). The 32 most significant bits of $\mathsf{RIPEMD}(M)$ are in A_n; the 32 least significant bits in D_n.

These two steps are given in detail in the rest of this section.

3.2 Expanding the Message

Let $N = ((n+64) \text{ div } 512)+1$. The n-bit message $M = (m_0, m_1, \ldots, m_{n-1})$ is expanded to the $16N$-word message $X = (X_0, X_1, \ldots, X_{16N-1})$ in the following three steps.

1. First of all, the following $k+1$ bits are appended to the message M, where $0 \leq k < 512$ and $n + k + 1 \equiv 448 \pmod{512}$:

$$m_n := 1, \quad m_{n+1} := m_{n+2} := \cdots := m_{n+k} := 0. \tag{15}$$

That is, a single bit '1' is appended; next, a number of zero bits is appended until the message is 64 bits shy of a multiple of 512 bits.

2. This $(n+k+1)$-bit extended message is transformed into the $\frac{n+k+1}{32} = 16N - 2$ words $X_0, X_1, \ldots, X_{16N-3}$ according to conventions given in Section 2.3, hence

$$X_i := \sum_{j=0}^{3} \sum_{l=0}^{7} m_{32i+8j+l} 2^{8j+7-l}, \quad i = 0, 1, \ldots, 16N-3. \tag{16}$$

3. Finally, the expansion is completed by appending the length n of the original message is in the following way.

$$\begin{aligned} X_{16N-2} &:= n \bmod 2^{32}, \\ X_{16N-1} &:= (n \bmod 2^{64}) \text{ div } 2^{32}. \end{aligned}$$

3.3 The Compression Function compress

For the 4-word number (A, B, C, D) and the 16-word message X_0, X_1, ..., X_{15}, the 4-word function value

$$\mathsf{compress}((A, B, C, D); (X_0, X_1, \ldots, X_{15})) \tag{17}$$

is computed in two parallel sets of three rounds, each round consisting of 16 applications of one of the auxiliary operations defined by Equations (8)–(13).

First, two copies of (A, B, C, D) are made. Next, both halves independently transform those copies in three rounds of sixteen applications of one of the operations *FF* through *HHH*. Finally, the results of both halves and the initial values of A, B, C and D are combined into one 4-word output, here denoted as (AA, BB, CC, DD).

Pseudocode:

```
/* copy (A,B,C,D) for both halves */
aa = A;  bb = B;  cc = C;  dd = D;
aaa = A;  bbb = B;  ccc = C;  ddd = D;

/* Round 1 for both parallel halves */
FF(aa, bb, cc, dd, X0, 11);        FFF(aaa, bbb, ccc, ddd, X0, 11);
FF(dd, aa, bb, cc, X1, 14);        FFF(ddd, aaa, bbb, ccc, X1, 14);
FF(cc, dd, aa, bb, X2, 15);        FFF(ccc, ddd, aaa, bbb, X2, 15);
FF(bb, cc, dd, aa, X3, 12);        FFF(bbb, ccc, ddd, aaa, X3, 12);
FF(aa, bb, cc, dd, X4, 5);         FFF(aaa, bbb, ccc, ddd, X4, 5);
FF(dd, aa, bb, cc, X5, 8);         FFF(ddd, aaa, bbb, ccc, X5, 8);
FF(cc, dd, aa, bb, X6, 7);         FFF(ccc, ddd, aaa, bbb, X6, 7);
FF(bb, cc, dd, aa, X7, 9);         FFF(bbb, ccc, ddd, aaa, X7, 9);
FF(aa, bb, cc, dd, X8, 11);        FFF(aaa, bbb, ccc, ddd, X8, 11);
FF(dd, aa, bb, cc, X9, 13);        FFF(ddd, aaa, bbb, ccc, X9, 13);
FF(cc, dd, aa, bb, X10, 14);       FFF(ccc, ddd, aaa, bbb, X10, 14);
FF(bb, cc, dd, aa, X11, 15);       FFF(bbb, ccc, ddd, aaa, X11, 15);
FF(aa, bb, cc, dd, X12, 6);        FFF(aaa, bbb, ccc, ddd, X12, 6);
FF(dd, aa, bb, cc, X13, 7);        FFF(ddd, aaa, bbb, ccc, X13, 7);
FF(cc, dd, aa, bb, X14, 9);        FFF(ccc, ddd, aaa, bbb, X14, 9);
FF(bb, cc, dd, aa, X15, 8);        FFF(bbb, ccc, ddd, aaa, X15, 8);

/* Round 2 for both parallel halves */
GG(aa, bb, cc, dd, X7, 7);         GGG(aaa, bbb, ccc, ddd, X7, 7);
GG(dd, aa, bb, cc, X4, 6);         GGG(ddd, aaa, bbb, ccc, X4, 6);
GG(cc, dd, aa, bb, X13, 8);        GGG(ccc, ddd, aaa, bbb, X13, 8);
GG(bb, cc, dd, aa, X1, 13);        GGG(bbb, ccc, ddd, aaa, X1, 13);
GG(aa, bb, cc, dd, X10, 11);       GGG(aaa, bbb, ccc, ddd, X10, 11);
GG(dd, aa, bb, cc, X6, 9);         GGG(ddd, aaa, bbb, ccc, X6, 9);
GG(cc, dd, aa, bb, X15, 7);        GGG(ccc, ddd, aaa, bbb, X15, 7);
GG(bb, cc, dd, aa, X3, 15);        GGG(bbb, ccc, ddd, aaa, X3, 15);
```

$GG(aa, bb, cc, dd, X_{12}, 7)$;
$GG(dd, aa, bb, cc, X_0, 12)$;
$GG(cc, dd, aa, bb, X_9, 15)$;
$GG(bb, cc, dd, aa, X_5, 9)$;
$GG(aa, bb, cc, dd, X_{14}, 7)$;
$GG(dd, aa, bb, cc, X_2, 11)$;
$GG(cc, dd, aa, bb, X_{11}, 13)$;
$GG(bb, cc, dd, aa, X_8, 12)$;

$GGG(aaa, bbb, ccc, ddd, X_{12}, 7)$;
$GGG(ddd, aaa, bbb, ccc, X_0, 12)$;
$GGG(ccc, ddd, aaa, bbb, X_9, 15)$;
$GGG(bbb, ccc, ddd, aaa, X_5, 9)$;
$GGG(aaa, bbb, ccc, ddd, X_{14}, 7)$;
$GGG(ddd, aaa, bbb, ccc, X_2, 11)$;
$GGG(ccc, ddd, aaa, bbb, X_{11}, 13)$;
$GGG(bbb, ccc, ddd, aaa, X_8, 12)$;

/* Round 3 for both parallel halves */

$HH(aa, bb, cc, dd, X_3, 11)$;
$HH(dd, aa, bb, cc, X_{10}, 13)$;
$HH(cc, dd, aa, bb, X_2, 14)$;
$HH(bb, cc, dd, aa, X_4, 7)$;
$HH(aa, bb, cc, dd, X_9, 14)$;
$HH(dd, aa, bb, cc, X_{15}, 9)$;
$HH(cc, dd, aa, bb, X_8, 13)$;
$HH(bb, cc, dd, aa, X_1, 15)$;
$HH(aa, bb, cc, dd, X_{14}, 6)$;
$HH(dd, aa, bb, cc, X_7, 8)$;
$HH(cc, dd, aa, bb, X_0, 13)$;
$HH(bb, cc, dd, aa, X_6, 6)$;
$HH(aa, bb, cc, dd, X_{11}, 12)$;
$HH(dd, aa, bb, cc, X_{13}, 5)$;
$HH(cc, dd, aa, bb, X_5, 7)$;
$HH(bb, cc, dd, aa, X_{12}, 5)$;

$HHH(aaa, bbb, ccc, ddd, X_3, 11)$;
$HHH(ddd, aaa, bbb, ccc, X_{10}, 13)$;
$HHH(ccc, ddd, aaa, bbb, X_2, 14)$;
$HHH(bbb, ccc, ddd, aaa, X_4, 7)$;
$HHH(aaa, bbb, ccc, ddd, X_9, 14)$;
$HHH(ddd, aaa, bbb, ccc, X_{15}, 9)$;
$HHH(ccc, ddd, aaa, bbb, X_8, 13)$;
$HHH(bbb, ccc, ddd, aaa, X_1, 15)$;
$HHH(aaa, bbb, ccc, ddd, X_{14}, 6)$;
$HHH(ddd, aaa, bbb, ccc, X_7, 8)$;
$HHH(ccc, ddd, aaa, bbb, X_0, 13)$;
$HHH(bbb, ccc, ddd, aaa, X_6, 6)$;
$HHH(aaa, bbb, ccc, ddd, X_{11}, 12)$;
$HHH(ddd, aaa, bbb, ccc, X_{13}, 5)$;
$HHH(ccc, ddd, aaa, bbb, X_5, 7)$;
$HHH(bbb, ccc, ddd, aaa, X_{12}, 5)$;

/* combination of results into output */

$AA := (B + cc + ddd) \bmod 2^{32}$;
$BB := (C + dd + aaa) \bmod 2^{32}$;
$CC := (D + aa + bbb) \bmod 2^{32}$;
$DD := (A + bb + ccc) \bmod 2^{32}$;

4 Use of the Primitive

The primitive RIPEMD (and any other hash function) can be used in the following ways.

- Firstly, it can be used in a data integrity mechanism, namely for the verification of the authenticity of a message.
- Secondly, it can be used in a signature mechanism, namely to compress a message before signing it.
- Thirdly, it can be used for the generation of publicly accessible password or passphrase files.

Those modes of use are explained below.

Use in data integrity mechanisms The primitive RIPEMD can be used for the verification of the authenticity of a message. This is done as follows, and under the following constraints.

After receipt of a message and the corresponding hashcode, one (re)calculates the hashcode of the message. If the calculated hashcode is equal to the value originally received, it is reasonable to assume that the original hashcode is computed from the same message. This holds, as computation of another message with the same hashcode is claimed to be infeasible (see Section 5.1). Therefore, if it can be guaranteed that the hashcode is unmodified, it is reasonable to assume that the message is authentic (unmodified).

So, in order to verify the authenticity of a message, only the hashcode must be protected against modification. For the message itself this is not required. The protection of the hashcode may be achieved by public key methods, such as digital signatures, by secret key methods, such as encryption, or through physical or logical means. For more details, see Part II of this report. An example of a public key primitive that achieves this is RSA that is described in Chapter 7 of this report.

Use in signature mechanisms A digital signature enables the receiver of a message to prove to a third party that the message has been created by a specific entity.

If the message is long, it is much more efficient to sign the hashcode of a message than to sign the message itself. Because of the collision resistance and one-way property, it is infeasible to find two messages with the same hashcode or to find a message resulting in a given hashcode (see Section 5.1). As a consequence, it is infeasible to find two messages with the same signature, or to find a message resulting in a given signature as well.

For a digital signature primitive we refer to Chapter 7 of this report.

Use for readable password files Because of the collision-resistance and the one-way property, RIPEMD can be used for the generation of publicly accessible (readable) password or passphrase files. This is done by storing RIPEMD(*password*) in this file, instead of the password itself. Because of the one-way property, it is infeasible to derive a valid password from an entry in the file. Because of the collision resistance, this holds even if a valid password is known.

5 Security Evaluation

5.1 Claimed Properties

RIPEMD is conjectured to be a one-way collision resistant hash function. That is, it is conjectured to satisfy the following two conditions:

1. It is computationally infeasible to compute two distinct messages M_1 and M_2 that have the same hashcode (message digest), i.e., RIPEMD(M_1) = RIPEMD(M_2). A hash function that satisfies this property is called *collision resistant.*

2. It is computationally infeasible, given a message M and the corresponding hashcode, to compute a different message M' that has the same hashcode. That is, given a message M, it is infeasible to compute a message $M' \neq M$ such that RIPEMD(M') = RIPEMD(M). A hash function that satisfies this property is called *one-way.*

By "computationally infeasible" we mean to express the practical impossibility of computing something with the technology that is currently available or can be foreseen to become available in the near future.

It is hard to give a bound beyond which a computation is infeasible, but currently, a computation requiring 2^{60} (or 10^{18}) operations is computationally infeasible. On the other hand, a computation taking 2^{40} (about 10^{12}) operations is hard, but not impossible.

In view of the past development of computing power and the relatively new possibility of using huge amounts of computing power of for example idle workstations in a network, one should cater for a substantial increase in the bound for computational infeasibility on the longer term.

Since a proof for the conjectured strength of RIPEMD cannot be given, the rest of this section is dedicated to the statistical and algebraic evaluation of the algorithm.

5.2 Statistical Evaluation

It is the general view that in order to avoid possible statistical attacks, a good cryptographic function should behave like a random function. The same holds for the compression function compress used in the hash function RIPEMD. Statistical irregularities in the behavior of this function could be used in finding collisions or messages that produce a given hashcode (see Section 5.1).

In order to determine the statistical properties of compress, three tests were applied to the function. These tests investigate the following properties of a cryptographic function:

1. The dependence of input and output bits of the function.

2. Possible linear relations between input and output bits of the function.

3. The periodicity properties of the function.

Below, these tests and the results of the application of these tests to compress are shortly described.

5.2.1 The Dependence Test

It has been stated in the literature that a good cryptographic function should satisfy the following three properties:

Completeness: a function is said to be *complete* if each output bit depends on all input bits [KaDa79].

Avalanche effect: a function exhibits the *avalanche effect* if for each input bit, a change of this single input bit results in a change of an average of half of the output bits [Fei73].

Strict avalanche criterion: a function satisfies the *strict avalanche criterion* if every output bit changes with probability $\frac{1}{2}$ whenever one input bit is changed [WeTa85].

Obviously, if a function satisfies the strict avalanche criterion, then it will also be complete and exhibit the avalanche effect.

The dependence test determines to what extent a function satisfies these three properties. More precisely, the test computes for a number of randomly chosen inputs the so-called *dependence* and *distance* matrices of the function that are defined as follows. The (i, j)-th entry of the dependence matrix is defined as the number of inputs for which changing the i-th input bit results in a change of the j-th output bit. The (i, j)-th entry of the distance matrix is defined as the number of inputs for which changing the i-th input bit results in the changing of j output bits. From these matrices the test computes the degree to which the function satisfies the three properties above, and furthermore the test determines whether these matrices differ significantly from what can be expected for a random function.

5.2.2 The Linear Factors Test

A function is said to have a *linear factor* if the modulo 2 sum of a set of input and output bits never changes when a specific input bit is changed. Linear factors indicate a serious problem with the security of a cryptographic function. For example, in [ChEv85] an attack is described on DES with a reduced number of rounds that is based on linear factors.

The linear factors test is a test that determines the linear factors of a function in a very efficient way. Furthermore, this test can also be used to compute *pseudo linear factors*, that is linear factors that hold with a probability significantly larger than one half. However, while the test will always yield all the linear factors of a function, the same does not hold for pseudo linear factors.

5.2.3 The Cycling Test

For any function f that maps n bits onto n bits, and any n-bit number x (the *initial value*), the sequence $x, f(x), f^2(x) = f(f(x)), \ldots$ will ultimately be periodic. More precisely, there exist numbers l and c such that the $l+c$ values $x, f(x), f^2(x), \ldots, f^{l+c-1}(x)$ are distinct, but $f^{l+c}(x) = f^l(x)$. The number l is called the *leader length* of the sequence, and c the *cycle length*.

The cycling test determines the leader and cycle lengths of sequences of the above form for a number of random values of x, and compares the results with what can be expected for a random function. For more details on the cycling test, see [Knu81, Chapter 3] and [KRS88].

5.2.4 Results of the Tests

The results of the tests are as follows.

1. The dependence test was applied to the three consecutive rounds of the function compress (see Section 3.3). For each application of the test, 25,000 random 128-bit initial values and 512-bit messages were used. After one round of compress each of the 128 output bits depends on each of the 128 bits of the initial values. After two rounds, each of the 128 output bits also depends on each of the 512 message bits. In general, the results of the dependence test applied to two rounds and three rounds (and hence the whole function) of the function compress do not differ significantly from what can be expected for a random function.

2. The function compress does not contain any linear factors. Furthermore, a search for pseudo linear factors yielded no results.

3. The cycling test was applied to a number of 32-bit functions obtained by considering random sets of 32 input bits and 32 output bits of the function compress. Also, for each of these functions, the test was applied for a number of random initial values. The results of these cycling experiments on compress do not differ significantly from what can be expected for a random function.

We can conclude from the results of these tests that there is no indication that the statistical behavior of the function compress used in RIPEMD differs significantly from what can be expected for a random function.

5.3 Algebraic Evaluation

5.3.1 Justification of the Focus on compress

A theorem by Damgård [Dam89] states that a family of hash functions satisfying certain requirements is collision resistant and one-way if the compression function is collision resistant. This implies that a member of such a family with sufficiently large input block size, is collision resistant and one-way with high probability. As RIPEMD can easily be embedded in such a family, Damgård's theorem provides evidence for the

fact that RIPEMD is collision resistant and one-way if compress is collision resistant. This justifies the fact that this section focuses on the properties of compress rather than RIPEMD.

5.3.2 Collision Resistance of compress

The fact that compress consists of two independent parallel sets of three rounds makes constructing collisions much harder, as such a construction has to cope with both halves simultaneously: any change in one half has to be cancelled out by the other half.

Moreover, both halves consist of three serial rounds of sixteen steps each. As in both halves each input word is used in all three rounds, a change in one input word results in a cascade of three changes. Because of the choice of the shifts in the different steps, and the order of use of the input words, the effects of such a change are hard to control or restrict. For example, for each message block X_i, three different rotations are used in the three rounds; message blocks that are used close to each other in one round are far apart in the other two; for each of the four words *aa* through *dd* (*aaa* through *ddd*) there is at least one round with an odd sum of rotations, the difference of two consecutive rotations is never divisible by 4, etc.

So, finding a collision means simultaneously solving the 'serial' problem per set of three rounds, and the 'parallel' problem of making the results of both halves cancel each other out. This seems infeasible.

Brute force attacks to find a collision are certainly computationally infeasible, as the best known brute force attack, a so-called 'birthday attack' will require 2^{64} steps. Here each step essentially consists of an application of compress.

The best attack that finds an input or a second input to compress that yields a prespecified output is a 'brute force' attack. This implies that no better attack on RIPEMD can be found.

The best attack that finds a preimage that compresses to a prespecified hashcode, given a value for $(H1, H2)$, is a pure brute force attack. This would take in the order of 2^{127} evaluations of compress. Of course, this number is also limited by the effective size of the message space, hence this space should not be too small.

Finding a different message that compresses to the same preimage as a given message also requires in the order of 2^{127} evaluations of compress. If we allow n different hashcodes, the complexity of this attack would decrease by a factor of n, but a storage in the order of n input/hashcode pairs is necessary.

Hence, brute force attacks are computationally infeasible.

6 Performance Evaluation

6.1 Software Implementations

The design of the RIPEMD algorithm is oriented towards a fast software implementation on 32-bit architectures: it is based on a simple set of primitive operations on 32-bit words. Moreover, no substitution tables are used. Therefore, the implementation can be quite compact as well. Both a C and a 80386 Assembly language implementation are considered. The C version has the advantage of being portable. The difference in performance between the C and the Assembly language version is mainly due to the inability to efficiently implement a rotation instruction in C. The figures are for an IBM-compatible 33 MHz 80386DX based PC with 64K cache memory using WATCOM C/386 9.0 in combination with the DOS/4GW DOS extender. Hence, both programs use a 32-bit instruction set and are run in protected mode. The codesize entry in Table 1 refers to the size of the compression function code.

	C		Assembly	
	Codesize	Speed	Codesize	Speed
RIPEMD	4125	3.44 Mbit/s	2250	6.80 Mbit/s

Table 1: Software performance of RIPEMD on a 33 MHz 80386DX based PC with a 64K memory cache using WATCOM C/386 9.0 in combination with the DOS/4GW DOS extender. No tables are used.

6.2 Hardware Implementations

The algorithm is specifically designed for software implementations. The large input block size is unfavorable for compact hardware implementations. However, in view of the performance of the software implementations mentioned above, and taking into account that only basic operations like addition modulo 2^{32}, bitwise XOR, AND, rotation and complementation are used, a high speed in hardware is certainly possible. Parallelism can only be used to run both halves of RIPEMD simultaneously, as every register in each half gets updated sequentially.

7 Guidelines for Software Implementation

The C-implementation (and the comments in the code), given in Appendix A can be used as a starting point for an implementation. However, it should be noted that this implementation was written for the sole purpose of documentation. No optimization whatsoever is performed, but only readability and portability were kept in mind. The rest of this section is dedicated to possible optimizations. Since this is to a large extent dependent on the specific architecture on which an implementation is to be performed, only guidelines can be given.

Firstly, the ordering of the bytes in a word enables the following optimization in the conversion of an array of bytes to an array of words on so-called "little endian" architectures. On such architectures, it is not necessary to shift each byte to the left over 0, 8, 16 or 24 positions, and then copy it to a word X_i. (Note that this is done in the macro `BYTES_TO_WORD` in `ripemd.h`.) Instead, the array of bytes can be treated as an array of words, as the bytes already are in the correct order.

That is, replace the lines

```
for (i=0; i<16; i++) {
   X[i] = BYTES_TO_WORD(strptr);
   strptr += 4;
}
compress(MDbuf, X);
```

with

```
compress(MDbuf, (dword *)strptr);
strptr += 64;
```

The same thing can be done for the output: the conversion of an array of words to an array of bytes can be done by just returning `(byte *)MDbuf`.

Those optimizations yield in the order of 20% increase in speed.

The rationale for this ordering of bytes was that "big endian" machines tend to be faster than little endian machines, so they should do the extra work of reordering the bytes into a word [Riv90].

Secondly, if at all possible, use a 32-bit instruction set. On PC's with a 32-bit architecture (e.g, 80386 and 80486 based IBM-compatibles) this might require another compiler, as many popular compilers only use a 16-bit instruction set. This will increase the speed with a factor in the order of three.

It is worth while to force the buffer variables *aa* through *dd* and *aaa* through *ddd* in registers.

Some other small optimizations are possible by rewriting the macros `G()`, `ROT()` and `FF()` through `HHH()`.

`G()` can be optimized by observing that $(X \wedge Y) \vee (X \wedge Z) \vee (Y \wedge Z)$ is equivalent to $(X \wedge (Y \vee Z)) \vee (Y \wedge Z)$.

By using a temporary variable (that should be declared in the calling function) the rotation can be optimized as follows:

```
#define ROT(x, n)    (tmp=(x), (tmp << (n)) | (tmp >> (32-(n))))
```

By doing this, the macros `FF()` through `HHH()` become slightly more efficient (an optimizing compiler will keep `tmp` in a register):

```
#define FF(a, b, c, d, x, n)   (a) = ROT((a) + F((b), (c), (d)) + (x), (s))
```

Assembly implementations should incorporate the above ideas. Note that most processors have a rotate instruction, which of course should be used.

References

[Boe92] B. den Boer, *Personal Communication*, March 1992.

[BoBo91] B. den Boer and A. Bosselaers, "An attack on the last two rounds of MD4," in: *Advances in Cryptology - CRYPTO'91*, J. Feigenbaum ed., Lecture Notes in Computer Science no. 576, Springer-Verlag, Berlin-Heidelberg-New York, pp. 194-203, 1992.

[ChEv85] D. Chaum and J.-H. Evertse, "Cryptanalysis of DES with a reduced number of rounds," in: *Advances in Cryptology - CRYPTO'85*, H.C. Williams ed., Lecture Notes in Computer Science no. 218, Springer-Verlag, Berlin-Heidelberg-New York, pp. 192-211, 1986.

[Dam89] I.B. 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, Berlin-Heidelberg-New York, pp. 416-427, 1990.

[Fei73] H. Feistel, "Cryptography and computer privacy," *Scientific American*, Vol. 228, pp. 15-23, 1973.

[NBS77] National Bureau of Standards, *Data Encryption Standard*, Federal Information Processing Standard, Publication 46, US Department of Commerce, January 1977.

[KRS88] B.B. Kaliski, R.L. Rivest and A.T. Sherman, "Is the data encryption standard a group? (Results of cycling experiments on DES)," *Journal of Cryptology*, Vol. 1, no. 1, pp. 3-36, 1988.

[KaDa79] J.B. Kam and G.I. Davida, "Structured design of substitution-permutation encryption networks," *IEEE Transactions on Computers*, Vol. C-28, no. 10, pp. 747-753, 1979.

[Knu81] D.E. Knuth, *The Art of Computer Programming, Vol. 2: Seminumerical Algorithms, 2nd Edition*, Addison-Wesley, Reading Mass., 1981.

[Mer90] R.C. Merkle, *Personal Communication*, 1990.

[Riv90] R.L. Rivest, "The MD4 message-digest algorithm," in: *Advances in Cryptology - CRYPTO'90*, A.J. Menezes and S.A. Vanstone eds., Lecture Notes in Computer Science no. 537, Springer-Verlag, Berlin-Heidelberg-New York, pp. 303-311, 1991.

[RiDu91] R.L. Rivest and S. Dussé, *The MD5 message-digest algorithm*, INTERNET Draft, July 10th, 1991.

[WeTa85] A.F. Webster and S.E. Tavares, "On the design of S-boxes," in: *Advances in Cryptology - CRYPTO'85*, H.C. Williams ed., Lecture Notes in Computer Science no. 218, Springer-Verlag, Berlin-Heidelberg-New York, pp. 523-534, 1986.

A C Implementation of the Primitive

This appendix provides an ANSI C-implementation of the primitive RIPEMD and an example program that uses RIPEMD to hash messages. This program can be used for testing purposes as well, as it can provide the test values of Appendix B.

Use of this implementation The file `ripemd.c` contains the functions `MDinit`, `compress` and `MDfinish`. `MDinit` performs initialization of $H1$ and $H2$, `compress` applies compress, `MDfinish` pads the message, appends the length and compresses the resulting blocks. The header file `ripemd.h` contains the function prototypes and some macros.

The test program `hashtest` can be compiled as follows. First, compile `ripemd.c` and `hashtest.c` with an (ANSI) C compiler. Next, link the resulting object files.

This implementation has been tested on a wide variety of environments, so it should be portable or at worst easy to port. The testing environments include MS-DOS (4.2 and 5.0 on both 80286 and 80386 processors) with several compilers, VAX/VMS, RISC ULTRIX, ConvexOs and Macintosh.

Note that VAX/VMS does not allow the `run` command to pass arguments to an executable. A patch for this is given in the comment of the `main()` function in `hashtest.c`.

A.1 The Header File for RIPEMD

```
/*************************************************************************/
/*  file: ripemd.h                                                       */
/*                                                                       */
/*  description: header file for RIPEMD, a sample C-implementation       */
/*           This function is derived from the MD4 Message Digest        */
/*           Algorithm from RSA Data Security, Inc.                      */
/*           This implementation was developed by RIPE.                  */
/*                                                                       */
/*  copyright (C)                                                        */
/*           Centre for Mathematics and Computer Science, Amsterdam      */
/*           Siemens AG                                                  */
/*           Philips Crypto BV                                           */
/*           PTT Research, the Netherlands                               */
/*           Katholieke Universiteit Leuven                              */
/*           Aarhus University                                           */
/*  1992, All Rights Reserved                                            */
/*                                                                       */
/*  date:    05/06/92                                                    */
/*  version: 1.0                                                         */
/*                                                                       */
/*************************************************************************/

#ifndef  RIPEMDH           /* make sure this file is read only once */
#define  RIPEMDH

/*************************************************************************/

/* typedef 8, 16 and 32 bit types, resp.  */
/* adapt these, if necessary,
   for your operating system and compiler */
typedef    unsigned long        dword;
typedef    unsigned short       word;
typedef    unsigned char        byte;


/*************************************************************************/

/* macro definitions */

/* collect four bytes into one word: */
#define BYTES_TO_WORD(strptr)                    \
            (((dword) *((strptr)+3) << 24) | \
             ((dword) *((strptr)+2) << 16) | \
             ((dword) *((strptr)+1) <<  8) | \
             ((dword) *(strptr)))

/* ROL(x, n) cyclically rotates x over n bits to the left */
/* x must be of an unsigned 32 bits type and 0 <= n < 32. */
#define ROL(x, n)        (((x) << (n)) | ((x) >> (32-(n))))

/* the three basic functions F(), G() and H() */
```

```
#define F(x, y, z)        (((x) & (y)) | ((~x) & (z)))
#define G(x, y, z)        (((x) & (y)) | ((x) & (z)) | ((y) & (z)))
#define H(x, y, z)        ((x) ^ (y) ^ (z))

/* the six basic operations FF() through HHH() */
#define FF(a, b, c, d, x, s)        {\
      (a) += F((b), (c), (d)) + (x);\
      (a) = ROL((a), (s));\
   }
#define GG(a, b, c, d, x, s)        {\
      (a) += G((b), (c), (d)) + (x) + (dword)0x5a827999UL;\
      (a) = ROL((a), (s));\
   }
#define HH(a, b, c, d, x, s)        {\
      (a) += H((b), (c), (d)) + (x) + (dword)0x6ed9eba1UL;\
      (a) = ROL((a), (s));\
   }
#define FFF(a, b, c, d, x, s)        {\
      (a) += F((b), (c), (d)) + (x) + (dword)0x50a28be6UL;\
      (a) = ROL((a), (s));\
   }
#define GGG(a, b, c, d, x, s)        {\
      (a) += G((b), (c), (d)) + (x);\
      (a) = ROL((a), (s));\
   }
#define HHH(a, b, c, d, x, s)        {\
      (a) += H((b), (c), (d)) + (x) + (dword)0x5c4dd124UL;\
      (a) = ROL((a), (s));\
   }

/***********************************************************************/

/* function prototypes */

void MDinit(dword *MDbuf);
/*
 *  initializes MDbuffer to "magic constants"
 */

void compress(dword *MDbuf, dword *X);
/*
 *  the compression function.
 *  transforms MDbuf using message bytes X[0] through X[15]
 */

void MDfinish(dword *MDbuf, byte *strptr, dword lswlen, dword mswlen);
/*
 *  puts bytes from strptr into X and pad out; appends length
 *  and finally, compresses the last block(s)
 *  note: length in bits == 8 * (lswlen + 2^32 mswlen).
 *  note: there are (lswlen mod 64) bytes left in strptr.
 */
```

```
#endif  /* RIPEMDH */

/************************ end of file ripemd.h ************************/
```

A.2 The C Source Code for RIPEMD

```
/********************************************************************/
/*  file: ripemd.c                                                  */
/*                                                                  */
/*  description: A sample C-implementation of the RIPEMD           */
/*           hash-function. This function is derived from the MD4   */
/*           Message Digest Algorithm from RSA Data Security, Inc.  */
/*           This implementation was developed by RIPE.             */
/*                                                                  */
/*  copyright (C)                                                   */
/*           Centre for Mathematics and Computer Science, Amsterdam */
/*           Siemens AG                                             */
/*           Philips Crypto BV                                      */
/*           PTT Research, the Netherlands                          */
/*           Katholieke Universiteit Leuven                         */
/*           Aarhus University                                      */
/*  1992, All Rights Reserved                                       */
/*                                                                  */
/*  date:    05/25/92                                               */
/*  version: 1.0                                                    */
/*                                                                  */
/********************************************************************/


/*  header files */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "ripemd.h"

/********************************************************************/

void MDinit(dword *MDbuf)
{
   MDbuf[0] = 0x67452301UL;
   MDbuf[1] = 0xefcdab89UL;
   MDbuf[2] = 0x98badcfeUL;
   MDbuf[3] = 0x10325476UL;

   return;
}

/********************************************************************/

void compress(dword *MDbuf, dword *X)
{
   dword aa = MDbuf[0],  bb = MDbuf[1],  cc = MDbuf[2],  dd = MDbuf[3];
   dword aaa = MDbuf[0], bbb = MDbuf[1], ccc = MDbuf[2], ddd = MDbuf[3];

   /* round 1 */
   FF(aa, bb, cc, dd, X[0], 11);
   FF(dd, aa, bb, cc, X[1], 14);
```

```
FF(cc, dd, aa, bb, X[2], 15);
FF(bb, cc, dd, aa, X[3], 12);
FF(aa, bb, cc, dd, X[4],  5);
FF(dd, aa, bb, cc, X[5],  8);
FF(cc, dd, aa, bb, X[6],  7);
FF(bb, cc, dd, aa, X[7],  9);
FF(aa, bb, cc, dd, X[8], 11);
FF(dd, aa, bb, cc, X[9], 13);
FF(cc, dd, aa, bb, X[10],14);
FF(bb, cc, dd, aa, X[11],15);
FF(aa, bb, cc, dd, X[12], 6);
FF(dd, aa, bb, cc, X[13], 7);
FF(cc, dd, aa, bb, X[14], 9);
FF(bb, cc, dd, aa, X[15], 8);

/* round 2 */
GG(aa, bb, cc, dd, X[7],  7);
GG(dd, aa, bb, cc, X[4],  6);
GG(cc, dd, aa, bb, X[13], 8);
GG(bb, cc, dd, aa, X[1], 13);
GG(aa, bb, cc, dd, X[10],11);
GG(dd, aa, bb, cc, X[6],  9);
GG(cc, dd, aa, bb, X[15], 7);
GG(bb, cc, dd, aa, X[3], 15);
GG(aa, bb, cc, dd, X[12], 7);
GG(dd, aa, bb, cc, X[0], 12);
GG(cc, dd, aa, bb, X[9], 15);
GG(bb, cc, dd, aa, X[5],  9);
GG(aa, bb, cc, dd, X[14], 7);
GG(dd, aa, bb, cc, X[2], 11);
GG(cc, dd, aa, bb, X[11],13);
GG(bb, cc, dd, aa, X[8], 12);

/* round 3 */
HH(aa, bb, cc, dd, X[3], 11);
HH(dd, aa, bb, cc, X[10],13);
HH(cc, dd, aa, bb, X[2], 14);
HH(bb, cc, dd, aa, X[4],  7);
HH(aa, bb, cc, dd, X[9], 14);
HH(dd, aa, bb, cc, X[15], 9);
HH(cc, dd, aa, bb, X[8], 13);
HH(bb, cc, dd, aa, X[1], 15);
HH(aa, bb, cc, dd, X[14], 6);
HH(dd, aa, bb, cc, X[7],  8);
HH(cc, dd, aa, bb, X[0], 13);
HH(bb, cc, dd, aa, X[6], 6);
HH(aa, bb, cc, dd, X[11],12);
HH(dd, aa, bb, cc, X[13], 5);
HH(cc, dd, aa, bb, X[5],  7);
HH(bb, cc, dd, aa, X[12], 5);

/* parallel round 1 */
FFF(aaa, bbb, ccc, ddd, X[0], 11);
```

```
FFF(ddd, aaa, bbb, ccc, X[1], 14);
FFF(ccc, ddd, aaa, bbb, X[2], 15);
FFF(bbb, ccc, ddd, aaa, X[3], 12);
FFF(aaa, bbb, ccc, ddd, X[4],  5);
FFF(ddd, aaa, bbb, ccc, X[5],  8);
FFF(ccc, ddd, aaa, bbb, X[6],  7);
FFF(bbb, ccc, ddd, aaa, X[7],  9);
FFF(aaa, bbb, ccc, ddd, X[8], 11);
FFF(ddd, aaa, bbb, ccc, X[9], 13);
FFF(ccc, ddd, aaa, bbb, X[10],14);
FFF(bbb, ccc, ddd, aaa, X[11],15);
FFF(aaa, bbb, ccc, ddd, X[12], 6);
FFF(ddd, aaa, bbb, ccc, X[13], 7);
FFF(ccc, ddd, aaa, bbb, X[14], 9);
FFF(bbb, ccc, ddd, aaa, X[15], 8);

/* parallel round 2 */
GGG(aaa, bbb, ccc, ddd, X[7],  7);
GGG(ddd, aaa, bbb, ccc, X[4],  6);
GGG(ccc, ddd, aaa, bbb, X[13], 8);
GGG(bbb, ccc, ddd, aaa, X[1], 13);
GGG(aaa, bbb, ccc, ddd, X[10],11);
GGG(ddd, aaa, bbb, ccc, X[6],  9);
GGG(ccc, ddd, aaa, bbb, X[15], 7);
GGG(bbb, ccc, ddd, aaa, X[3], 15);
GGG(aaa, bbb, ccc, ddd, X[12], 7);
GGG(ddd, aaa, bbb, ccc, X[0], 12);
GGG(ccc, ddd, aaa, bbb, X[9], 15);
GGG(bbb, ccc, ddd, aaa, X[5],  9);
GGG(aaa, bbb, ccc, ddd, X[14], 7);
GGG(ddd, aaa, bbb, ccc, X[2], 11);
GGG(ccc, ddd, aaa, bbb, X[11],13);
GGG(bbb, ccc, ddd, aaa, X[8], 12);

/* parallel round 3 */
HHH(aaa, bbb, ccc, ddd, X[3], 11);
HHH(ddd, aaa, bbb, ccc, X[10],13);
HHH(ccc, ddd, aaa, bbb, X[2], 14);
HHH(bbb, ccc, ddd, aaa, X[4],  7);
HHH(aaa, bbb, ccc, ddd, X[9], 14);
HHH(ddd, aaa, bbb, ccc, X[15], 9);
HHH(ccc, ddd, aaa, bbb, X[8], 13);
HHH(bbb, ccc, ddd, aaa, X[1], 15);
HHH(aaa, bbb, ccc, ddd, X[14], 6);
HHH(ddd, aaa, bbb, ccc, X[7],  8);
HHH(ccc, ddd, aaa, bbb, X[0], 13);
HHH(bbb, ccc, ddd, aaa, X[6], 6);
HHH(aaa, bbb, ccc, ddd, X[11],12);
HHH(ddd, aaa, bbb, ccc, X[13], 5);
HHH(ccc, ddd, aaa, bbb, X[5],  7);
HHH(bbb, ccc, ddd, aaa, X[12], 5);

/* combine results */
```

```
   ddd += cc + MDbuf[1];                /* final result for MDbuf[0] */
   MDbuf[1] = MDbuf[2] + dd + aaa;
   MDbuf[2] = MDbuf[3] + aa + bbb;
   MDbuf[3] = MDbuf[0] + bb + ccc;
   MDbuf[0] = ddd;

   return;
}

/***********************************************************************/

void MDfinish(dword *MDbuf, byte *strptr, dword lswlen, dword mswlen)
{
   dword        i;                                  /* counter       */
   dword        X[16];                              /* message words */

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

   /* put bytes from strptr into X */
   for (i=0; i<(lswlen&63); i++) {
      /* byte i goes into word X[i div 4] at pos.  8*(i mod 4)  */
      X[i>>2] ^= (dword) *(strptr++) << (8 * (i&3));
   }

   /* append the bit m_n == 1 */
   X[(lswlen>>2)&15] ^= (dword)1 << (8*(lswlen&3) + 7);

   if ((lswlen & 63) > 55) {
      /* length goes to next block */
      compress(MDbuf, X);
      for (i=0; i<16; i++) {
         X[i] = 0;
      }
   }

   /* append length in bits*/
   X[14] = lswlen << 3;
   X[15] = (lswlen >> 29) | (mswlen << 3);

   compress(MDbuf, X);

   return;
}

/************************ end of file ripemd.c **********************/
```

A.3 An Example Program

This section gives the listing of an example program. By means of command line options, several different tests can be performed (see the top of the file). Test values can be found in Appendix B. The file `test.bin` used by the test suite `hashtest -a` should contain the string `abc` and nothing else. (Make sure no newline or linefeed character is appended by your editor.)

```
/*****************************************************************/
/*  file: hashtest.c                                             */
/*                                                               */
/*  description: test file for ripemd.c, a sample C-implementation */
/*           of the RIPEMD hash-function.                        */
/*                                                               */
/*  command line arguments:                                      */
/*           filename  -- compute hash code of file read binary  */
/*           -sstring  -- print string & hascode                 */
/*           -t        -- perform time trial                     */
/*           -a        -- execute standard test suite, ASCII input */
/*                        and binary input from file test.bin    */
/*           -x        -- execute standard test suite, hexadecimal */
/*                        input read from file test.hex          */
/*                                                               */
/*  copyright (C)                                                */
/*           Centre for Mathematics and Computer Science, Amsterdam */
/*           Siemens AG                                          */
/*           Philips Crypto BV                                   */
/*           PTT Research, the Netherlands                       */
/*           Katholieke Universiteit Leuven                      */
/*           Aarhus University                                   */
/*  1992, All Rights Reserved                                    */
/*                                                               */
/*  date:    06/25/92                                            */
/*  version: 1.0                                                 */
/*                                                               */
/*****************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <string.h>
#include "ripemd.h"

#ifndef CLK_TCK
#define CLK_TCK CLOCKS_PER_SEC   /* this is a hack for c89 compiler */
#endif                           /* (DEC C for ULTRIX on RISC v1.0) */

#define TEST_BLOCK_SIZE 8000
#define TEST_BLOCKS 1250

/* number of test bytes = TEST_BLOCK_SIZE * TEST_BLOCKS */
static long TEST_BYTES = (long)TEST_BLOCK_SIZE * (long)TEST_BLOCKS;
```

```
/********************************************************************/

byte *RIPEMD(byte *message)
/*
 * returns RIPEMD(message)
 * message should be a string terminated by '\0'
 */
{
   dword         MDbuf[4];     /* contains (A, B, C, D)             */
   static byte   hashcode[16];   /* for final hash-value            */
   dword         X[16];        /* current 16-word chunk             */
   word          i;            /* counter                           */
   dword         length;       /* length in bytes of message        */
   dword         nbytes;       /* # of bytes not yet processed      */
   byte         *strptr;       /* points to the current mess. chunk */

   /* initialize */
   MDinit(MDbuf);
   strptr = message;                /* strptr points to first chunk */
   length = (dword)strlen((char *)message);
   nbytes = length;

   /* process message in 16-word chunks */
   while (nbytes > 63) {
      for (i=0; i<16; i++) {
         X[i] = BYTES_TO_WORD(strptr);
         strptr += 4;
      }
      compress(MDbuf, X);
      nbytes -= 64;                    /* 64 bytes less to process */
   }                                   /* length mod 64 bytes left */

   /* finish: */
   MDfinish(MDbuf, strptr, length, 0);

   for (i=0; i<16; i+=4) {
      hashcode[i]   =  MDbuf[i>>2];         /* implicit cast to byte  */
      hashcode[i+1] = (MDbuf[i>>2] >>  8);  /*  extracts the 8 least  */
      hashcode[i+2] = (MDbuf[i>>2] >> 16);  /*  significant bits.     */
      hashcode[i+3] = (MDbuf[i>>2] >> 24);
   }

   return (byte *)hashcode;
}

/********************************************************************/

byte *RIPEMDbinary(char *fname)
/*
 * returns RIPEMD(message in file fname)
 * fname is read as binary data.
 */
{
```

```
  FILE         *mf;                /* pointer to file <fname>      */
  byte          data[1024];        /* contains current mess. block */
  dword         nbytes;            /* length of this block         */
  dword         MDbuf[4];          /* contains (A, B, C, D)        */
  static byte   hashcode[16];      /* for final hash-value         */
  dword         X[16];             /* current 16-word chunk        */
  word          i, j;              /* counters                     */
  dword         length[2];         /* length in bytes of message   */
  dword         offset;            /* # of unprocessed bytes at    */
       /* call of MDfinish */

  /* initialize */
  MDinit(MDbuf);
  if ((mf = fopen(fname, "rb")) == NULL) {
     fprintf(stderr, "\nRIPEMDbinary: cannot open file \"%s\".\n",
             fname);
     exit(1);
  }

  while ((nbytes = fread(data, 1, 1024, mf)) != 0) {
     /* process all complete blocks */
     for (i=0; i<(nbytes>>6); i++) {
        for (j=0; j<16; j++) {
           X[j] = BYTES_TO_WORD(data+64*i+4*j);
        }
        compress(MDbuf, X);
     }
     /* update length[] */
     if (length[0] + nbytes < length[0])
        length[1]++;                  /* overflow to msb of length */
     length[0] += nbytes;
  }

  /* finish: */
  offset = length[0] & 0x3C0;   /* extract bytes 6 to 10 inclusive */
  MDfinish(MDbuf, data+offset, length[0], length[1]);

  for (i=0; i<16; i+=4) {
     hashcode[i]   =  MDbuf[i>>2];
     hashcode[i+1] = (MDbuf[i>>2] >>  8);
     hashcode[i+2] = (MDbuf[i>>2] >> 16);
     hashcode[i+3] = (MDbuf[i>>2] >> 24);
  }

  fclose(mf);

  return (byte *)hashcode;
}

/*********************************************************************/


byte *RIPEMDhex(char *fname)
```

```
/*
 * returns RIPEMD(message in file fname)
 * fname should contain the message in hex format;
 * first number of bytes, then the bytes in hexadecimal.
 */
{
   FILE         *mf;               /* pointer to file <fname>      */
   byte          data[64];         /* contains current mess. block */
   dword         nbytes;           /* length of the message        */
   dword         MDbuf[4];         /* contains (A, B, C, D)        */
   static byte   hashcode[16];       /* for final hash-value       */
   dword         X[16];            /* current 16-word chunk        */
   word          i, j;             /* counters                     */
   int           val;              /* temp for reading from file   */

   /* initialize */
   if ((mf = fopen(fname, "r")) == NULL) {
      fprintf(stderr, "\nRIPEMDhex: cannot open file \"%s\".\n",
              fname);
      exit(1);
   }
   MDinit(MDbuf);

   fscanf(mf, "%x", &val);
   nbytes = val;
   i = 0;
   while (nbytes - i > 63) {
      /* read and process complete block */
      for (j=0; j<64; j++) {
         fscanf(mf, "%x", &val);
         data[j] = (byte)val;
      }
      for (j=0; j<16; j++) {
         X[j] = BYTES_TO_WORD(data+4*j);
      }
      compress(MDbuf, X);
      i += 64;
   }

   /* read last nbytes-i bytes: */
   j = 0;
   while (i<nbytes) {
      fscanf(mf, "%x", &val);
      data[j++] = (byte)val;
      i++;
   }

   /* finish */
   MDfinish(MDbuf, data, nbytes, 0);

   for (i=0; i<16; i+=4) {
      hashcode[i]   =  MDbuf[i>>2];
      hashcode[i+1] = (MDbuf[i>>2] >>  8);
```

```
      hashcode[i+2] = (MDbuf[i>>2] >> 16);
      hashcode[i+3] = (MDbuf[i>>2] >> 24);
   }

   fclose(mf);

   return (byte *)hashcode;
}

/********************************************************************/

void speedtest(void)
/*
 * A time trial routine, to measure the speed of ripemd.
 * Measures processor time required to process TEST_BLOCKS times
 *  a message of TEST_BLOCK_SIZE characters.
 */
{
  clock_t    t0, t1;
  byte       data[TEST_BLOCK_SIZE];
  byte       hashcode[64];
  dword      X[16];
  dword      MDbuf[4];
  word       i, j, k;

  /* initialize test data */
  for (i=0; i<TEST_BLOCK_SIZE; i++)
     data[i] = (byte)(i%1000);

  /* start timer */
  printf ("RIPEMD time trial. Processing %ld characters...\n", TEST_BYTES);
  t0 = clock();

  /* process data */
  MDinit(MDbuf);
  for (i=0; i<TEST_BLOCKS; i++) {
     for (j=0; j<TEST_BLOCK_SIZE; j+=64) {
        for (k=0; k<16; k++) {
           X[k] = BYTES_TO_WORD(data+j+4*k);
        }
        compress(MDbuf, X);
     }
  }
  MDfinish(MDbuf, data, TEST_BYTES, 0);

  /* stop timer, get time difference */
  t1 = clock();
  printf("\nTest input processed in %g seconds.\n",
         ((double)(t1-t0)/(double)CLK_TCK));
  printf("Characters processed per second: %g\n",
         (double)CLK_TCK*TEST_BYTES/((double)t1-t0));

  for (i=0; i<16; i+=4) {
```

```
      hashcode[i]   =  MDbuf[i>>2];
      hashcode[i+1] = (MDbuf[i>>2] >>  8);
      hashcode[i+2] = (MDbuf[i>>2] >> 16);
      hashcode[i+3] = (MDbuf[i>>2] >> 24);
   }
   printf("\nhashcode: ");
   for (i=0; i<16; i++)
      printf("%02x", hashcode[i]);

   return;
}

/********************************************************************/

void testascii (void)
/*
 *   standard test suite, ASCII input
 */
{
   int i;
   byte *hashcode;

   printf("\nRIPEMD test suite results (ASCII):\n");

   hashcode = RIPEMD((byte *)"");
   printf("\n\nmessage: \"\"  (empty string)\nhashcode: ");
   for (i=0; i<16; i++)
      printf("%02x", hashcode[i]);

   hashcode = RIPEMD((byte *)"a");
   printf("\n\nmessage: \"a\"\nhashcode: ");
   for (i=0; i<16; i++)
      printf("%02x", hashcode[i]);

   hashcode = RIPEMD((byte *)"abc");
   printf("\n\nmessage: \"abc\"\nhashcode: ");
   for (i=0; i<16; i++)
      printf("%02x", hashcode[i]);

   hashcode = RIPEMD((byte *)"message digest");
   printf("\n\nmessage: \"message digest\"\nhashcode: ");
   for (i=0; i<16; i++)
      printf("%02x", hashcode[i]);

   hashcode = RIPEMD((byte *)"abcdefghijklmnopqrstuvwxyz");
   printf("\n\nmessage: \"abcdefghijklmnopqrstuvwxyz\"\nhashcode: ");
   for (i=0; i<16; i++)
      printf("%02x", hashcode[i]);

   hashcode = RIPEMD((byte *)
      "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789");
   printf("\n\nmessage: A...Za...z0...9\nhashcode: ");
   for (i=0; i<16; i++)
```

```
      printf("%02x", hashcode[i]);

   hashcode = RIPEMD((byte *)"12345678901234567890123456789012345678901\
23456789012345678901234567890123456789012");
   printf("\n\nmessage: 8 times \"1234567890\"\nhashcode: ");
   for (i=0; i<16; i++)
      printf("%02x", hashcode[i]);

   /* Contents of binary created file test.bin are "abc" */
   printf("\n\nmessagefile (binary): test.bin\nhashcode: ");
   hashcode = RIPEMDbinary ("test.bin");
   for (i=0; i<16; i++)
      printf("%02x", hashcode[i]);

   return;
}

/*********************************************************************/

void testhex (void)
/*
 *   standard test suite, hex input, read from files
 */
{
   int i;
   byte *hashcode;

   printf("\nRIPEMD test suite results (hex):\n");

   hashcode = RIPEMDhex("test1.hex");
   printf("\n\nfile test1.hex; hashcode: ");
   for (i=0; i<16; i++)
      printf("%02x", hashcode[i]);

   hashcode = RIPEMDhex("test2.hex");
   printf("\n\nfile test2.hex; hashcode: ");
   for (i=0; i<16; i++)
      printf("%02x", hashcode[i]);

   hashcode = RIPEMDhex("test3.hex");
   printf("\n\nfile test3.hex; hashcode: ");
   for (i=0; i<16; i++)
      printf("%02x", hashcode[i]);

   hashcode = RIPEMDhex("test4.hex");
   printf("\n\nfile test4.hex; hashcode: ");
   for (i=0; i<16; i++)
      printf("%02x", hashcode[i]);

   hashcode = RIPEMDhex("test5.hex");
   printf("\n\nfile test5.hex; hashcode: ");
   for (i=0; i<16; i++)
      printf("%02x", hashcode[i]);
```

```
   hashcode = RIPEMDhex("test6.hex");
   printf("\n\nfile test6.hex; hashcode: ");
   for (i=0; i<16; i++)
      printf("%02x", hashcode[i]);

   hashcode = RIPEMDhex("test7.hex");
   printf("\n\nfile test7.hex; hashcode: ");
   for (i=0; i<16; i++)
      printf("%02x", hashcode[i]);

   return;
}

/*********************************************************************/

main (int argc, char *argv[])
/*
 *  main program. calls one or more of the test routines depending
 *  on command line arguments. see the header of this file.
 *
 *  (For VAX/VMS, do: HASHTEST :== $<pathname>HASHTEST.EXE
 *   at the command prompt (or in login.com) first.
 *   (The run command does not allow command line args.)
 *   The <pathname> must include device, e.g., "DSKD:".)
 */
{
  int   i, j;
  byte *hashcode;

   if (argc == 1) {
      fprintf(stderr, "hashtest: no command line arguments supplied.\n");
      exit(1);
   }
   else {
      for (i = 1; i < argc; i++) {
         if (argv[i][0] == '-' && argv[i][1] == 's') {
            printf("\n\nmessage: %s", argv[i]+2);
            hashcode = RIPEMD((byte *)argv[i] + 2);
            printf("\nhashcode: ");
            for (j=0; j<16; j++)
               printf("%02x", hashcode[j]);
         }
         else if (strcmp (argv[i], "-t") == 0)
            speedtest ();
         else if (strcmp (argv[i], "-a") == 0)
            testascii ();
         else if (strcmp (argv[i], "-x") == 0)
            testhex ();
         else {
            hashcode = RIPEMDbinary (argv[i]);
            printf("\n\nmessagefile (binary): %s", argv[i]);
            printf("\nhashcode: ");
```

```
            for (j=0; j<16; j++)
               printf("%02x", hashcode[j]);
         }
      }
   }
   printf("\n");

   return 0;
}

/********************** end of file hashtest.c **********************/
```

B Test Values

Below, the files `test1.hex` up to `test7.hex` as used by the test `hashtest -x` are listed. The format of those files is as follows. All numbers are hexadecimal; all numbers except possibly the first one are one byte long. The first number represents the number of bytes in the message to be hashed; it is followed by (at least) this number of bytes. For example, `test3.hex` represents the message consisting of the three bytes 0x61, 0x62 and 0x63. This is the string "abc" in ASCII.

`test1.hex`:

```
0
```

`test2.hex`:

```
1
61
```

`test3.hex`:

```
3
61 62 63
```

`test4.hex`:

```
e
6d 65 73 73 61 67 65 20 64 69 67 65 73 74
```

`test5.hex`:

```
1a
61 62 63 64 65 66 67 68 69 6a 6b 6c 6d 6e 6f 70
71 72 73 74 75 76 77 78 79 7a
```

`test6.hex`:

```
3e
41 42 43 44 45 46 47 48 49 4a 4b 4c 4d 4e 4f 50
51 52 53 54 55 56 57 58 59 5a 61 62 63 64 65 66
67 68 69 6a 6b 6c 6d 6e 6f 70 71 72 73 74 75 76
77 78 79 7a 30 31 32 33 34 35 36 37 38 39
```

`test7.hex`:

```
50
31 32 33 34 35 36 37 38 39 30
31 32 33 34 35 36 37 38 39 30
31 32 33 34 35 36 37 38 39 30
31 32 33 34 35 36 37 38 39 30
31 32 33 34 35 36 37 38 39 30
31 32 33 34 35 36 37 38 39 30
31 32 33 34 35 36 37 38 39 30
31 32 33 34 35 36 37 38 39 30
```

The following test values were obtained by running `hashtest -x`. If ASCII encoding is used, `hashtest -a` should provide the same answers, followed by the result of hashing "`abc`" again. The latter only holds if the file `test.bin` exists and contains nothing but this string.

```
RIPEMD test suite results (hex):


file test1.hex; hashcode: 9f73aa9b372a9dacfb86a6108852e2d9

file test2.hex; hashcode: 486f74f790bc95ef7963cd2382b4bbc9

file test3.hex; hashcode: 3f14bad4c2f9b0ea805e5485d3d6882d

file test4.hex; hashcode: 5f5c7ebe1abbb3c7036482942d5f9d49

file test5.hex; hashcode: ff6e1547494251a1cca6f005a6eaa2b4

file test6.hex; hashcode: ff418a5aed3763d8f2ddf88a29e62486

file test7.hex; hashcode: dfd6b45f60fe79bbbde87c6bfc6580a5
```