RIPE integrity primitives Part II
Final report of RACE 1040

RIPE Consortium

Computer Science/Department of Algorithmics and Architecture

Report CS-R9325 April 1993

RIPE Integrity Primitives
Part II

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-R9325
ISSN 0169-118X
CWI
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

Contents

1 Introduction

This chapter describes the integrity primitives RIPE-MAC1 and RIPE-MAC3, commonly denoted as RIPE-MAC. Both are so-called *message authentication codes* (MACs) or *keyed hash functions* that, under control of a (secret) key, compress messages of arbitrary length to a 64-bit output block, the *hashcode* of the message. RIPE-MAC1 uses a 56-bit key and RIPE-MAC3 a 112-bit key. It is conjectured that for someone not in possession of the secret key it is computationally infeasible to produce for a given message the corresponding hashcode, or to produce, given a message and the corresponding hashcode, a different message having the same hashcode (i.e., a second preimage). Moreover it is conjectured that even when a large number of message-hashcode pairs are known, where the messages are selected by the opponent, it is computationally infeasible to determine the key or to produce the hashcode of a message not in this set of selected messages. Keyed hash functions with these properties are used in message authentication applications providing both data integrity and data origin authentication, as well as in identification schemes (see Section 4 of this chapter and Part II of this report).

The RIPE-MAC algorithm is based on the ISO/IEC standard 9797 *data integrity mechanism using a cryptographic check function employing a block cipher* [ISO89], but differs from it with respect to the internal structure of the compression function, the padding mechanism and the final processing. It consists of three parts. First the message is expanded to a length that is a multiple of 64 bits. Next the expanded message is divided up in blocks of 64 bits. A keyed compression function is used to iteratively compress these blocks under control of the secret key to a single block of 64 bits. For this keyed compression function a keyed one-way function is used based on the DES [NBS77] or a triple encryption mode of the DES, in order to provide a higher security level. The numbers at the end of the names RIPE-MAC1 and RIPE-MAC3 refer to the number of DES operations in a single application of this compression function. Finally the output of this iterative compression is subjected to a DES based encryption with a different key, derived from the key used in the compression.

Since the RIPE-MAC algorithm uses essentially one or three DES operations per message block, its design is oriented towards implementations using fast DES hardware. The performance of a pure software implementation will suffer from the low software performance of the DES. However, as the same key is used throughout the compression, the slow DES key scheduling has to be done only twice: once for the entire compression and once for the encryption at the end. Moreover, the inverse initial permutation has to be done only once at the end of the MAC calculation. This will help the software speed.

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 algebraic evaluation of the primitive. Finally, in Section 6 the performance aspects of RIPE-

MAC are considered, and Section 7 gives some guidelines for software implementation.

This chapter has two appendices. Appendix A contains a straightforward software implementation of RIPE-MAC 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 Numbers

In this chapter a *word* is defined as a 64-bit quantity. A word is considered to be a nonnegative integer. That is, it can take on the values 0 through $2^{64} - 1 = 18446744073709551615$. Normally the value of a word will be given in hexadecimal form. In that case the number is written as '0x' followed immediately by at most 16 hexadecimal digits, the most significant first. For example, the hexadecimal representation of the 64-bit number 7017280452245743464 is 0x6162636465666768.

A sequence of $64n$ bits $w_0, w_1, \ldots, w_{64n-1}$ is interpreted as a sequence of n words in the following way. Each group of 64 consecutive bits is considered as a word, the first bit of such a group being the most significant bit of that word. Hence,

$$W_i := \sum_{j=0}^{63} w_{64i+j} 2^{63-j} \qquad i = 0, 1, \ldots, n-1 \tag{1}$$

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 (1).

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}. X is said to be an n-bit string.

- For two n-bit strings X and Y the $2n$-bit string $W = X \parallel Y$ is defined as the *concatenation* of the strings X and Y. That is, according to the definition of a string above,
$$\begin{array}{rcl} w_i & := & x_i \\ w_{i+n} & := & y_i \end{array} \qquad i = 0, 1, \ldots, n-1.$$

- A 64-bit string will also be considered as a word according to the representation defined by Equation (1), and vice versa. That is, if X is an n-bit string, then the corresponding word is equal to
$$X = \sum_{i=0}^{63} x_i 2^{63-i}.$$
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 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 word $U = X \oplus Y$ is defined as the bitwise *XOR* of X and Y, respectively. Hence, according to Equation (1):
$$u_i := (x_i + y_i) \bmod 2, \qquad i = 0, 1, \ldots, 63.$$

2.5 Functions used by the Primitive

RIPE-MAC1 uses a single application and RIPE-MAC3 uses three applications of the Data Encryption Standard (DES) [NBS77] to map a word under control of a secret parameter, called the key, onto another word.

A DES encryption operation $E(\cdot)$ will be graphically represented as shown in Figure 1 and mathematically written as
$$Y = E(K, X).$$
The key K is represented as a word, but the eight parity bits are ignored. That is, the bits k_7, k_{15}, k_{23}, k_{39}, k_{47}, k_{55} and k_{63} of $K = (k_0, k_1, \ldots, k_{63})$ are not used. Hence K

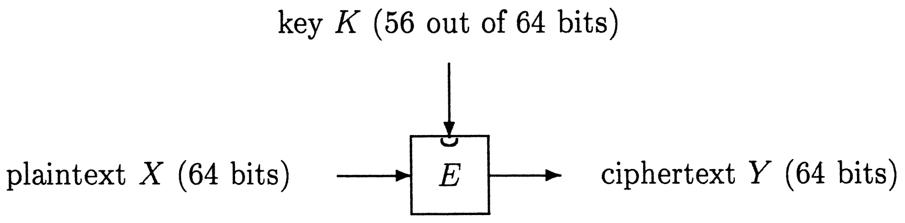

Figure 1: The basic DES encryption operation.

has in effect only a length of 56 bits. Whenever a reference is made to a 56-bit key K, the length of the string representing K will be 64 bits of which only 56 are used.

A DES decryption operation $D(\cdot)$ will be graphically and mathematically represented in the same way as the encryption operation, with E replaced by D. Hence,

$$D(K, E(K, X)) = X.$$

RIPE-MAC3 obtains a higher security level than RIPE-MAC1 by replacing the single DES encryption operation with a triple DES operation $E_3(\cdot)$ with two different 56-bit keys K_1 and K_2 [MeMa82]:

$$Y = E_3(K, X) := E(K_1, D(K_2, E(K_1, X))),$$

where $K = K_1 \parallel K_2$ is a single 112-bit key. Once again both K_1 and K_2 are represented as words, but their parity bits are ignored. Whenever a reference is made to a 112-bit key K, the length of the bit string representing K will be 128 bits of which only 112 are used. This primitive is depicted in Figure 2, together with a shorthand. Note that for $K_1 = K_2$ the result of $E_3(\cdot)$ is reduced to a single DES encryption with that same key:

$$E_3(K \parallel K, X) = E(K, X).$$

Hence an implementation of RIPE-MAC3 can be used to simulate RIPE-MAC1, although it will of course be slower (see Sections 6 and 7).

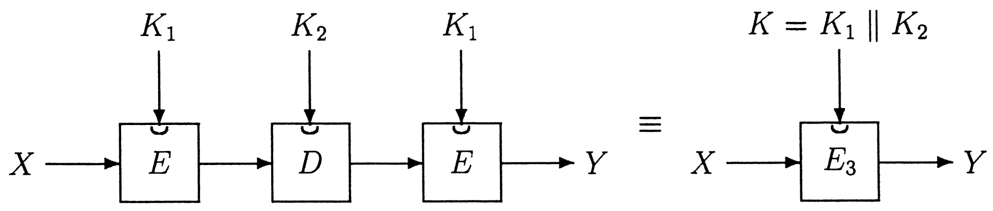

Figure 2: The triple DES operation, together with a shorthand.

In the description of the RIPE-MAC scheme, the encryption functions $E(\cdot)$ or $E_3(\cdot)$ can be substituted by other encryption functions. However the security of these new schemes has to be re-evaluated, as they depend on the properties of the new encryption function. In this chapter only the DES-based functions $E(\cdot)$ and $E_3(\cdot)$ are considered.

3 Description of the Primitive

3.1 Outline of RIPE-MAC1 and RIPE-MAC3

RIPE-MAC1 is a keyed hash function that maps a message M of arbitrary length under control of a 56-bit K onto a 64-bit block RIPE-MAC1(K, M). The basis of RIPE-MAC1 is the keyed compression function compress1$(\cdot)$. This function compresses a two word input to a single word output under control of the 56-bit key. 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 one-word initial vector, the sequence X is compressed by repeatedly appending a message word and compressing the resulting two words to one by applying compress1$(\cdot)$ until the message is exhausted. Finally, this single word result is encrypted with the function $E(\cdot)$ using a 56-bit K' derived from K. Below this is explained in detail.

Let $M = (m_0, m_1, \ldots, m_{n-1})$ be a message of n bits long. The 64-bit message authentication code RIPE-MAC1(K, M) of M is computed in three steps, see also Figure 4.

expansion: M is expanded to a sequence X consisting of N words X_0, X_1, …, X_{N-1}, where $N = ((n + 64) \text{ div } 64) + 1$. First, the message is expanded such that its length becomes a multiple of 64 bits. This padding is done even if the original message length is a multiple of 64 bits. Next, to complete X, an additional word representing the length of the original message is appended.

compression: Define the word H_0 as the all zero sequence:

$$H_0 \quad := \quad \texttt{0x0000000000000000}$$

For $i = 0, 1, \ldots, N-1$, the words H_{i+1} is computed from the words H_i and the message word X_i under control of the key K as follows:

$$H_{i+1} \quad := \quad \mathsf{compress1}(K, H_i, X_i).$$

encryption: The hashcode RIPE-MAC1(K, M) is equal to the 64-bit block $E(K', H_N)$, where

$$K' := K \oplus \texttt{0xf0f0f0f0f0f0f0f0}.$$

That is, every other 4 bits of K are complemented.

RIPE-MAC3 is described in exactly the same way as RIPE-MAC1, except that it uses a 112-bit key. A message M of arbitrary length is compressed under control of this 112-bit key K by means of the compression compress3$(\cdot)$ to a 64-bit block H_N. The hashcode RIPE-MAC3(K, M) is equal to the 64-bit block $E_3(K', H_N)$, where $K' = K_1' \parallel K_2'$ is derived from $K = K_1 \parallel K_2$ by complementing every other 4 bits of K. That is,

$$K_i' := K_i \oplus \texttt{0xf0f0f0f0f0f0f0f0}, \quad \text{for } i = 1, 2.$$

3.2 Expanding the Message

Let $N = (n \text{ div } 64) + 2$. 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-bit and $k = 63 - (n \bmod 64)$ 0-bits to the message M:

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

   That is, append a single 1-bit and as few (possibly none) 0-bits as necessary to obtain an expanded message that is a multiple of 64 bits. Note that padding is done even if the length of M is already a multiple of 64 bits.

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 Functions compress1 and compress3

For the word H_i, the message word X_i and the 56-bit key K the word function value compress1$(\cdot)$ is defined as (see also Figure 3):

$$\mathsf{compress1}(K, H_i, X_i) \;:=\; E(K, H_i \oplus X_i) \oplus X_i.$$

Similarly, for the word H_i, the message word X_i and the 112-bit key K the word function value compress3$(\cdot)$ is defined as:

$$\mathsf{compress3}(K, H_i, X_i) \;:=\; E_3(K, H_i \oplus X_i) \oplus X_i.$$

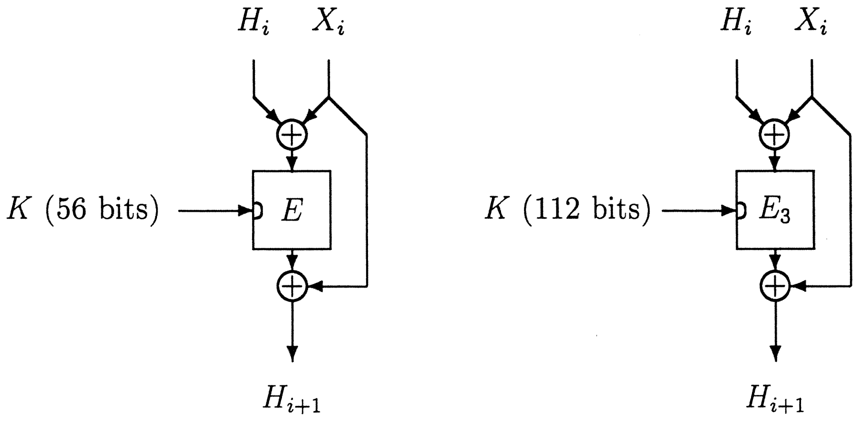

Figure 3: Outline of the two compression function compress1(·) (left, used in RIPE-MAC1) and compress3(·) (right, used in RIPE-MAC3).

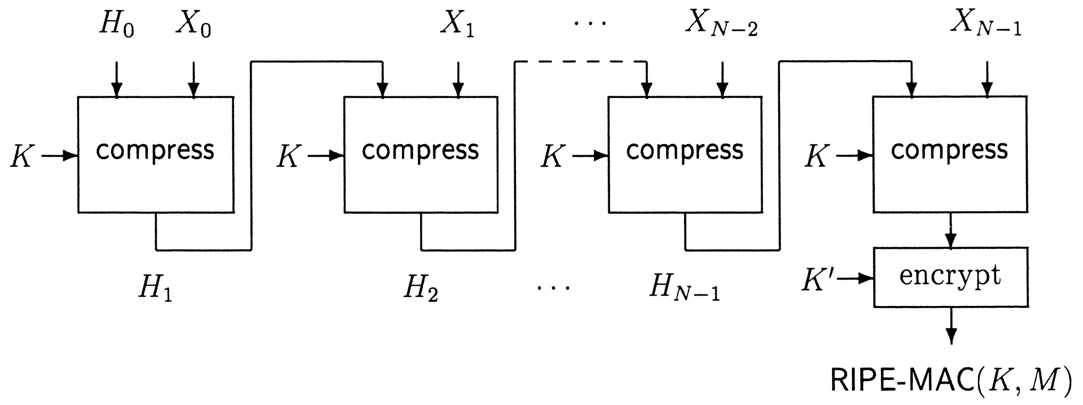

Figure 4: Outline of RIPE-MAC. The message M is first expanded to X, which is a multiple of 64 bits long. Then X is processed as in this picture. The final result is either RIPE-MAC1(K, M) or RIPE-MAC3(K, M), depending on the compression and encryption function used (respectively compress1 and E, or compress3 and E_3). The key K' is derived from the key K.

4 Use of the Primitive

The primitive RIPE-MAC has two intended applications:

- to be used in a data integrity mechanism, to provide both data integrity and data origin authentication,
- to be used in secret key identification schemes.

Below this is explained in detail.

Use in data integrity mechanisms Before a message is sent, the secret key is used to compute a MAC or keyed hashcode from it. This value is sent along with the message and can be checked by the legitimate recipient using the common secret key. If the calculated hashcode is equal to the original, received value, it is reasonable to assume that the original hashcode is computed from the same message. This holds, since computation of another message with the same hashcode is claimed to be infeasible for someone not in possession of the secret key (see Section 5). Moreover, it is reasonable to assume that the message is authentic, as it is infeasible for someone who does not know the secret key to produce a hashcode for a given message (see Section 5). The latter furthermore allows to link the hashcode to the originator of the information. Therefore, the primitive provides both data integrity and data origin authentication (see Part II of this report).

Use in secret key identification schemes Secret key identification schemes allow parties to establish that their secret key sharing counterpart is actually communicating with them at a particular moment. One party supplies the other party a random challenge. The other applies the primitive to the challenge sent and returns the result. The challenging party does the same and compares the calculated hashcode with the returned one. If they are the same, it is reasonable to assume that the other party is in possession of the secret key. Of course the roles can be reversed, so that each party is able to challenge the other. For this unilateral and bilateral authentication it is suggested to use, respectively, the RIPE primitives SKID2 and SKID3, for a description of which we refer to Chapter 6 of this document.

5 Security Evaluation

5.1 Claimed Properties

RIPE-MAC1 and RIPE-MAC3 are both claimed to be keyed one-way hash functions. That is, they should satisfy the following conditions:

1. It is computationally infeasible for someone not in possession of the key K to compute for a given message M the hashcode RIPE-MAC(K, M). Even when a large number of pairs $\{M_i, \text{RIPE-MAC}(K, M_i)\}$ are known, where the M_i have been selected by this opponent, it is computationally infeasible to determine the key K or to compute RIPE-MAC(K, M') for any $M' \neq M_i$. This attack is called an *adaptive chosen text attack*. A function that satisfies this property is called *keyed*.

2. It is computationally infeasible for someone who does not know the key K to compute, given a message and its corresponding hashcode, a second message having the same hashcode. That is, given a message M and its corresponding hashcode RIPE-MAC(K, M) it is infeasible to find a message $M' \neq M$ such that RIPE-MAC(K, M') = RIPE-MAC(K, M). Such a message is called a *second preimage*. A hash function that satisfies this property is called *one-way*.

By "computationally infeasible" we mean to express the 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 certainly, 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.

5.2 Algebraic Evaluation

The security of the RIPE-MAC scheme is intimately related with the security of the DES. Up to this point in time there is no reason to doubt it. The most effective known attack on the DES is the so-called 'differential cryptanalysis' [BiSh91a, BiSh91b]. This is a probabilistic attack 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 message blocks such that their XOR is preserved by the DES yield the same output of the compression function, as the XOR's cancel out. Hence, differential cryptanalysis is in principle applicable to RIPE-MAC.

To obtain the key K of RIPE-MAC1, an adaptive chosen text attack using differential cryptanalysis will require in the order of 2^{47} steps. The best one can do to find the key of RIPE-MAC3 is an exhaustive search requiring 2^{112} steps. The best one can do to obtain the hashcode of a message for RIPE-MAC1 or RIPE-MAC3 is to guess it with a probability of success of 2^{-64}.

The encryption of the hash value at the end of the compression chain and, to a lesser extent, the expansion of the original message are essential to prevent a chosen text attack on the RIPE-MAC scheme. Without these two steps it is possible, given the hashcodes H_1, H_2 and H_3 of three messages chosen by the user, to calculate the hashcode of a fourth message. Denote by RIPE-MAC' the RIPE-MAC scheme without the encryption and the expansion. Then, given

$$\begin{aligned} H_1 &= \text{RIPE-MAC'}(K, M_1) \\ H_2 &= \text{RIPE-MAC'}(K, M_2) \\ H_3 &= \text{RIPE-MAC'}(K, M_1 \parallel M_3), \end{aligned}$$

where each of the M_i is a 64-bit message block, the hashcode of the message $M_2 \parallel (M_3 \oplus H_1 \oplus H_2)$ equals

$$\text{RIPE-MAC'}(K, M_2 \parallel (M_3 \oplus H_1 \oplus H_2)) = H_1 \oplus H_2 \oplus H_3.$$

Moreover the compression functions used in this scheme have the advantage that they strengthen the one-way character of the hash function: even for someone in possession of the secret key it requires in the order of 2^{32} steps to find a preimage of a given value. Here each step essentially consists of an application of compress1 or compress3. It is trivial to produce a so called pseudo-preimage, that is, a preimage for an initial value different from the (fixed) proposed one. This follows from the fact that for someone who knows the key K, the scheme is invertible: given H_{i+1}, it is easy to find a pair (H_i, X_i). Hence, he can work his way back through all the stages of the MAC making choices for the message blocks X_i and calculating the corresponding intermediate hashvalues H_i, until he reaches an initial value H_0. This value will of course be different from the proposed initial value (the zero word). Therefore someone in possession of the secret key can construct a preimage of a given hashvalue with a so-called 'meet in the middle attack', that requires in the order of 2^{32} steps and storage.

6 Performance Evaluation

6.1 Software Implementations

The figures for a highly performant software implementation of RIPE-MAC1 and RIPE-MAC3 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 functions compress1 and compress3. 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 this configuration only marginally slower than the Assembly language implementation. However, as explained in Section 7, this is not necessarily the case for other configurations. All versions use the same tables totalling 80K of memory, and the same keyschedule, which uses no tables and about 6.5K of code. 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 all code runs in protected mode. The codesize entry in Table 1 refers to the size of the compression function code.

	C		Assembly language	
	Codesize	Speed	Codesize	Speed
RIPE-MAC1	1383	1.27 Mbit/s	1126	1.50 Mbit/s
RIPE-MAC3	3763	0.50 Mbit/s	3018	0.60 Mbit/s

Table 1: Software performance of RIPE-MAC 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. All versions use 80K of data.

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 on chip has been achieved [VHVM88, VHVM91, Cry89, Pij92]. A faster clock of 40 MHz would allow for data rates of up to 150 Mbit/s on chip. 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.

7 Guidelines for Software Implementation

The C-implementation given in Appendix A can be used as a starting point for an implementation. Every application of the compression functions compress1(·) or compress3(·) merely involves, respectively, one or three DES applications. Therefore the speed of a software implementation of RIPE-MAC will be determined by the efficiency with which a DES encryption can be performed. In the code of Appendix A both the DES key scheduling and the DES encryption are shown as function calls only. The software implementation of the DES, let alone an efficient implementation, is beyond the scope of this document. Only some rough guidelines for such an implementation are given. As the key scheduling is used only twice for each MAC calculation, or even less if the same key is used for subsequent MACs, there is no need for an efficient implementation of the key scheduling. Moreover, it is shown how in the implementation of RIPE-MAC one can get rid of all the inverse initial permutations of the DES, except for the last one.

The key to fast software implementation of the DES is the use of equivalent representations of the algorithm [DDFG83, DDGH84, FeKa89]. In general, the implementation of these representations is a time-memory trade-off. The more memory is used for tables, the more instructions can be replaced by a single table lookup, the faster the code will be. The combination of a number of small tables into a fewer number of big tables will reduce the number of these table lookups, and hence will further increase the speed. Moreover, both the initial and inverse initial permutation, as well as the expansion operation contain a lot of structure. A rearrangement of the bit order therefore allows for a very efficient implementation of the expansion operation, while the bit rearrangement can be combined with the already available permutation tables of initial, inverse initial and P permutation. This way typically about 20% of the time for a single DES encryption is spent on the initial and inverse initial permutation, while about 75% is used for running through the 16 rounds. The remaining 5% is spent on the subroutine call and the initialization of some variables. In the case of triple DES the inverse initial permutations at the end of the first two DES applications cancel out against the initial permutations at the beginning of the last two DES applications. This way typically less than 10% of the time for a triple DES application is spent on the initial and inverse initial permutation, while about 90% is used for running through the 48 rounds.

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 actually gets slowed down quite significantly. This means that a program with more instructions but less 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.

Almost all of the time of RIPE-MAC1 and RIPE-MAC3 spent on the inverse initial permutation can be saved by noting that the initial permutation can be moved upwards over the initial combining XOR towards the inputs H_i and X_i, while similarly the inverse initial permutation can be moved downwards over the final combining XOR towards the output H_{i+1}, see Figure 5. This way the initial permutation of H_i cancels out against the inverse initial permutation at the end of the previous stage. This means that only the message blocks X_i have to be initially permuted, and that only the result at the end of the RIPE-MAC chain has to be inverse initially permuted to obtain the hashcode RIPE-MAC1(K, M) or RIPE-MAC3(K, M).

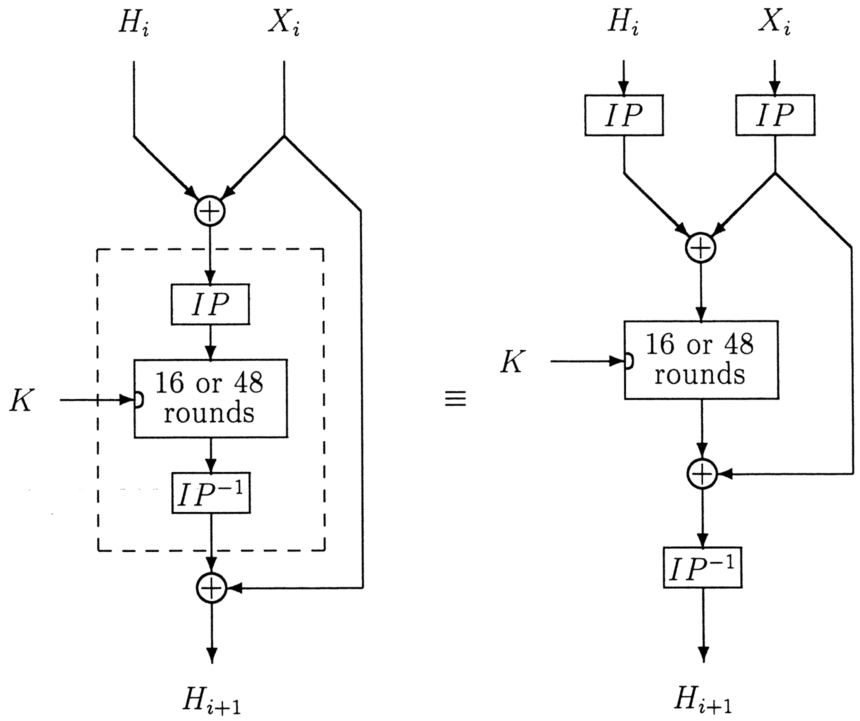

Figure 5: Equivalent representations of a RIPE-MAC stage.

References

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

[BiSh91b] E. Biham and A. Shamir, "Differential Cryptanalysis of the full 16-round DES," *Technion Technical Report # 708*, December 1991.

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

[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, J. 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 no. 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 no. 435, Springer-Verlag, Berlin-Heidelberg-New York, pp. 44-63, 1990.

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

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

[MeMa82] C.H. Meyer and S.M. Matyas, *"Cryptography: a new dimension in data security,"* Wiley & Sons, 1982.

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

[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 section provides a C implementation of the primitives RIPE-MAC1 and RIPE-MAC3. It includes an example program that uses these primitives to authenticate messages with a key chosen by the user. This example program can be used for testing purposes as well, as it can provide the test values of Appendix B.

Note that this implementation is designed for readability rather than speed. Moreover both the DES key scheduling and the DES encryption are shown as function calls only. No DES source code is included. Hence the speed of this implementation will mainly be determined by the quality of the DES implementation used with this program. For more details we refer to Section 7.

The functions in this implementation can be used in the following way. Compile the file `ripemac.c` (Appendix A.2) with an (ANSI) C compiler. Furthermore, provide a file that contains the definition of the DES functions `keyinit()` and `endes()` according to the prototypes given in the header file `ripemac.h` (Appendix A.1), taking into account the value of `NBYTES_KEY` defined in the same file. This value is either 8 or 16, and specifies the length in bytes of the data structure containing the key. Hence a value of 8 produces a RIPE-MAC1 implementation (56-bit key), and a value of 16 produces a RIPE-MAC3 implementation (112-bit key) (see comments in `ripemac.h` for more details). Next, provide a file that `#includes` the header file `ripemac.h` and contains a `main()` function calling `RIPEMAC()` or `RIPEMACfile()`. The function `RIPEMAC()` computes the hashcode of a '\0' terminated string, and the function `RIPEMACfile()` computes the hashcode of a binary file. The file `mactest.c`, given in Appendix A.3, can be used for this purpose. Finally link the resulting object files.

This implementation has been tested on a wide variety of environments, so it should be portable or at least easy to port. The testing environments include VAX/VMS, MS-DOS both with 16-bit and 32-bit compilers (Intel 80386 processor), RISC ULTRIX, Apollo DN3500 Domain/OS (Motorola 68030 processor).

A.1 The Header File for RIPE-MAC

```
/************************************************************************\
 *                                                                      *
 *     Header file for the Implementation of RIPE-MAC                   *
 *                                                                      *
 *     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                                                    *
 *                                                                      *
```

```
\************************************************************************/

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

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

/*
   NBYTES_KEY is the length of the key in bytes. It must be either 8 or
   16 bytes, for respectively a 56-bit (RIPE-MAC1) or 112-bit key
   (RIPE-MAC3). The rightmost bit of each byte (parity bit) is ignored.
   The functions keyinit() and endes() should be accordingly adapted.
   That is, for NBYTES_KEY == 8 keyinit() should install a single DES-key
   and endes() should perform a single DES encryption with this key.
   For NBYTES_KEY == 16 keyinit() should install two DES-keys and endes()
   should perform consecutively a DES encryption with the first key,
   a DES decryption with the second key and once again a DES encryption
   with the first key.
*/
#define NBYTES_KEY 8

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

/* Data strucure for RIPE-MAC computation */
typedef struct {
   byte  key[NBYTES_KEY];  /* Holds 56 or 112-bit key of MAC computation */
   byte  buffer[8];        /* Holds 64-bit result of MAC computation */
   dword count[2];         /* Holds number of 64-bit blocks processed so far */
   byte  done;             /* Nonzero means computation finished */
} MACstruct, *MACptr;

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

/* prototypes of DES functions */

void keyinit(byte *key);
/*
   installs a single or two new DES key. That is, keyinit() calculates
   the round keys for one or two DES keys. key is a pointer to an
   NBYTES_KEY-byte array. The parity bits are ignored.
*/

void endes(byte *inp, byte *outp);
/*
   a single DES encryption or a triple DES encryption-decryption-encryption
   operation with the key(s) installed by keyinit(). inp points to an
   8-byte array containing the plaintext, outp points to an 8-byte array
   that will contain the ciphertext.
```

```
*/

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

/* prototypes of RIPE-MAC functions */
void MACinit(MACptr MACp);
void MACupdate(MACptr MACp, byte *X, dword nrofblocks);
void MACfinal(MACptr MACp, byte *X, word count);
byte *RIPEMAC(byte *message, byte *key);
byte *RIPEMACfile(char *fname, byte *key);

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

A.2 C Source Code for RIPE-MAC

```
/**********************************************************************\
 *                                                                    *
 *     ripe-mac.c                                                     *
 *                                                                    *
 *     A sample C-implementation of the RIPE-MAC message              *
 *     authentication code.                                           *
 *                                                                    *
 *     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                                                  *
 *                                                                    *
\**********************************************************************/

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

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

void MACinit(MACptr MACp)
/*
   Initialize MAC computation.
*/
{
   int i;

   keyinit(MACp->key);
   for (i=0; i<8; i++)
      MACp->buffer[i] = 0;
   for (i=0; i<2; i++)
      MACp->count[i] = 0;
   MACp->done = 0;
}

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

void MACupdate(MACptr MACp, byte *X, dword nrofblocks)
/*
   compresses nrofblocks 8-byte message blocks contained in X.
   The result is returned in MACp->buffer.
   The MAC calculation should be finished up with a call to MACfinal().
```

```
*/
{
   register int j;
   dword        i;
   byte         H[8];

   /* Initialize 8-byte buffer H[8] */
   for (i=0; i<8; i++)
      H[i] = MACp->buffer[i];

   for (i=0; i<nrofblocks; i++) {

      /* the compression function */
      for (j=0; j<8; j++)
         H[j] ^= X[j];
      endes(H, H);
      for (j=0; j<8; j++)
         H[j] ^= X[j];

      X += 8;

   }

   for (i=0; i<8; i++)
      MACp->buffer[i] = H[i];

   /* Add count to MACp->count */
   if (nrofblocks + MACp->count[0] < MACp->count[0])
      /* overflow to msdw of MACp->count */
      MACp->count[1]++;
   MACp->count[0] += nrofblocks;

}

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

void MACfinal(MACptr MACp, byte *X, word count)
/*
   Put bytes from X into XX and pad out; compress this last block.
   count contains number of message bits in last block (between zero
   63, inclusive).
*/
{
   word  i, cbit, cbyte;
   byte  XX[8], mask;
   dword ls32, ms32;

   if (count == 0 && MACp->done) return;
   if (MACp->done) {
      printf("Error: MACfinal already done.\n");
      return;
   }
```

```
  /* Add count to MACp->count */
  ms32 = (MACp->count[1] << 6) | (MACp->count[0] >> 26);
  ls32 = (MACp->count[0] << 6) + count;
  /* Process data */
  if (count == 64) {
     /* Full block of data to handle */
     printf("Error: MACupdate should be called.\n");
     return;
  } else if (count > 64) {
     /* Check for count too large */
     printf("Error: MACfinal called with illegal count value %d.\n",count);
     return;
  } else {
     /* partial block -- must be last block so finish up */
     cbyte = count >> 3;
     cbit = count & 7;
     for (i=0; i<=cbyte; i++)
        XX[i] = X[i];
     for (i=cbyte+1; i<8; i++)
        XX[i] = 0;
     mask = 1 << (7 - cbit);
     XX[cbyte] = (XX[cbyte] | mask) & ~(mask - 1);
     MACupdate(MACp, XX, 1UL);

     /* final block with length */
     XX[3] = (byte)ms32;
     XX[2] = (byte)(ms32 >> 8);
     XX[1] = (byte)(ms32 >> 16);
     XX[0] = (byte)(ms32 >> 24);
     XX[7] = (byte)ls32;
     XX[6] = (byte)(ls32 >> 8);
     XX[5] = (byte)(ls32 >> 16);
     XX[4] = (byte)(ls32 >> 24);
     MACupdate(MACp, XX, 1UL);

     /* encrypt final block with different key */
     for (i=0; i<NBYTES_KEY; i++)
        MACp->key[i] ^= 0xF0;
     keyinit(MACp->key);
     endes(MACp->buffer, MACp->buffer);

     MACp->done = 1;
  }
}

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

byte *RIPEMAC(byte *message, byte *key)
/*
   computes RIPE-MAC(message,key) and returns the result as
   an array of 8-bytes.
*/
{
   word        i;
   dword       length;
   MACstruct   MAC;
   static byte mac[8];

   length = (dword)strlen((char*)message);
   for (i=0; i<NBYTES_KEY; i++)
      MAC.key[i] = key[i];

   MACinit(&MAC);
   MACupdate(&MAC, message, length >> 3);
   MACfinal(&MAC, message+(length & 0xFFFFFFF8UL), 8*(length & 0x7));

   for (i=0; i<8; i++)
      mac[i] = MAC.buffer[i];
   return mac;
}

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

byte *RIPEMACfile(char *fname, byte *key)
/*
   computes RIPE-MAC(contents of file <fname>,key) and
   returns the result as an array of 8-bytes.
   The contents of the file are interpreted as binary data.
*/
{
   word        i;
   dword       length;
   MACstruct   MAC;
   static byte mac[8];
   byte        data[1024];
   FILE        *f;

   if ( (f = fopen(fname,"rb")) == NULL ) {
      fprintf(stderr, "RIPEMACfile: cannot open file \"%s\".\n",fname);
      exit(1);
   }

   for (i=0; i<NBYTES_KEY; i++)
      MAC.key[i] = key[i];

   MACinit(&MAC);
   do {
      length = fread(data, 1, 1024, f);
      MACupdate(&MAC, data, length >> 3);
   } while ( length && ((length & 0x7) == 0) );
   MACfinal(&MAC, data+(length & 0x7F8UL), 8*(length & 0x7));

   fclose(f);

   for (i=0; i<8; i++)
      mac[i] = MAC.buffer[i];
   return mac;
}

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

A.3 An Example Program

Below an example program is given. It calls both RIPEMAC and RIPEMACfile. By means of command line options several different tests can be performed (see comment to main() function). Test values for both RIPE-MAC1 and RIPE-MAC3 can be found in Appendix B.

```
/********************************************************************\
 *                                                                    *
 *     mactest.c                                                      *
 *                                                                    *
 *     Test file for ripemac.c, a sample C-implementation of the      *
 *     RIPE-MAC message authentication code.                          *
 *                                                                    *
 *     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                                                  *
 *                                                                    *
\********************************************************************/

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

#define TEST_BLOCK_SIZE 4000UL     /* length of buffer, in blocks */
#define TEST_BLOCKS 1000000L       /* maximum number of test blocks */

#ifdef CLOCKS_PER_SEC
#define CLK_TCK CLOCKS_PER_SEC
#endif

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

void MACtimetrial(void)
/*
   A time trial routine, to measure the speed of RIPE-MAC.
   It measures processor time required to authenticate a message
   of nrofblocks (<= TEST_BLOCKS) blocks of 64 bits.
*/
{
   word      i, j, rounds, remainder;
   byte      *X;
```

```
dword     nrofblocks;
MACstruct MAC;
double    time_mac;
clock_t   t1, t2;

do {
   printf("\nEnter number of input blocks (<=%lu): ", TEST_BLOCKS);
   scanf("%lu", &nrofblocks);
} while (nrofblocks > TEST_BLOCKS);
srand(time(NULL));


for (i=0; i<NBYTES_KEY; i++)
   MAC.key[i] = (byte)(rand() >> 7);
MACinit(&MAC);
printf("\nKey: ");
for (i=0; i<NBYTES_KEY; i++)
   printf("%02x", MAC.key[i]);
printf("\n");

rounds = nrofblocks/TEST_BLOCK_SIZE;
remainder = nrofblocks % TEST_BLOCK_SIZE;
i = (rounds ? TEST_BLOCK_SIZE : remainder)*8;
if ( (X = (byte *) malloc(i)) == NULL ) {
   fprintf(stderr,
     "MACtimetrial: Could not allocate %u bytes - aborting\n", i);
   exit(1);
}

time_mac = 0;
for (j=0; j<rounds; j++) {
   for (i=0; i<8*TEST_BLOCK_SIZE; i++)
      X[i] = (byte)(rand() >> 7);
   t1 = clock();
   MACupdate(&MAC, (byte *)X, TEST_BLOCK_SIZE);
   t2 = clock();
   time_mac += (double)(t2-t1);
}
for (i=0; i<8*remainder; i++)
   X[i] = (byte)(rand() >> 7);
t1 = clock();
MACupdate(&MAC, (byte *)X, remainder);
MACfinal(&MAC, (byte *)X, 0);
t2 = clock();
time_mac += (double)(t2-t1);
free(X);

time_mac /= (double)CLK_TCK;
printf("\nRIPE-MAC%1d time trial results:\n", (NBYTES_KEY-4) >> 2);
printf("Test input processed in %g seconds\n", time_mac);
time_mac /= nrofblocks;
printf("Elapsed time per block: %g sec\n", time_mac);
printf("Characters processed per second: %lu.\n", (dword)(8/time_mac));
```

```
   printf("mac: ");
   for (i=0; i<8; i++) printf("%02x", MAC.buffer[i]);
   printf("\n");

}

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

void MACtestsuite(void)
/*
   standard test suite
*/
{
   byte key[NBYTES_KEY], temp;
   byte *mac;
   int  i;

   printf("\nRIPE-MAC%1d test suite results:\n", (NBYTES_KEY-4) >> 2);

   printf("\nkey: ");
   temp = 0x02;
   for (i=0; i<NBYTES_KEY; i++) {
      key[i] = temp | 0x10;
      printf("%02x", key[i]);
      temp += 0x22;
      if (i == 7)
         temp = 0x8A;
   }
   printf("\n");

   mac = RIPEMAC((byte *)"", key);
   printf("\nmessage: \"\" (empty string)\nmac: ");
   for (i=0; i<8; i++)
      printf("%02x", mac[i]);
   printf("\n");

   mac = RIPEMAC((byte *)"a", key);
   printf("\nmessage: \"a\"\nmac: ");
   for (i=0; i<8; i++)
      printf("%02x", mac[i]);
   printf("\n");

   mac = RIPEMAC((byte *)"abc", key);
   printf("\nmessage: \"abc\"\nmac: ");
   for (i=0; i<8; i++)
      printf("%02x", mac[i]);
   printf("\n");

   mac = RIPEMAC((byte *)"message authentication code", key);
   printf("\nmessage: \"message authentication code\"\nmac: ");
   for (i=0; i<8; i++)
      printf("%02x", mac[i]);
   printf("\n");
   mac = RIPEMAC((byte *)"abcdefghijklmnopqrstuvwxyz", key);
   printf("\nmessage: \"abcdefghijklmnopqrstuvwxyz\"\nmac: ");
   for (i=0; i<8; i++)
      printf("%02x", mac[i]);
   printf("\n");

   mac = RIPEMAC((byte *)"ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijkl\
mnopqrstuvwxyz0123456789", key);
   printf("\nmessage: alphabet in uppercase, in lower case and digits\
 0 through 9\nmac: ");
   for (i=0; i<8; i++)
      printf("%02x", mac[i]);
   printf("\n");

   mac = RIPEMAC((byte *)"123456789012345678901234567890123456789\
01234567890123456789012345678901234567890", key);
   printf("\nmessage: 8 times \"1234567890\"\nmac: ");
   for (i=0; i<8; i++)
      printf("%02x", mac[i]);
   printf("\n");
}

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

main(int argc, char *argv[])
/*
   main program. It calls one or more of the test routines depending
   on command line arguments:
         filename  prints filename and mac for key chosen by the user
         -sstring  prints string and mac for key chosen by the user
         -t        performs time trial
         -x        executes a standard suite of test data
*/
{
   word i, j, temp;
   byte *mac, key[NBYTES_KEY];

   if (argc == 1) {
      printf("For each command line argument in turn:\n");
      printf("  filename prints filename and mac\n");
      printf("  -sstring prints string and mac\n");
      printf("  -t       performs time trial\n");
      printf("  -x       executes a standard suite of test data\n");
   } else {
      for (i=1; i<argc; i++)
         if (argv[i][0] == '-' && argv[i][1] == 's') {
            printf("\nEnter key (%d bytes, in hexadecimal): ",
                    NBYTES_KEY);
            for (j=0; j<NBYTES_KEY; j++) {
               scanf("%2x", &temp);
               key[j] = (byte)temp;
            }
         printf("\nkey: ");
         for (j=0; j<NBYTES_KEY; j++)
            printf("%02x", key[j]);
         printf("\nmessage: %s", argv[i]+2);
         mac = RIPEMAC((byte *)(argv[i]+2), key);
         printf("\nmac: ");
         for (j=0; j<8; j++)
            printf("%02x", mac[j]);
      }
      else if (strcmp(argv[i],"-t") == 0)
         MACtimetrial();
      else if (strcmp(argv[i],"-x") == 0)
         MACtestsuite();
      else {
         printf("\nEnter key (%d bytes, in hexadecimal): ",
                NBYTES_KEY);
         for (j=0; j<NBYTES_KEY; j++) {
            scanf("%2x", &temp);
            key[j] = (byte)temp;
         }
         printf("\nkey: ");
         for (j=0; j<NBYTES_KEY; j++)
            printf("%02x", key[j]);
         printf("\nmessagefile (binary): %s", argv[i]);
         mac = RIPEMACfile(argv[i], key);
         printf("\nmac: ");
         for (j=0; j<8; j++)
            printf("%02x", mac[j]);
      }
   }
   return;
}

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

B Test Values

The following test values were obtained by running **`mactest -x`**. The first set is for RIPE-MAC1 (56-bit key, `NBYTES_KEY` = 8), while the second set is for RIPE-MAC3 (112-bit key, `NBYTES_KEY` = 16).

```
RIPE-MAC1 test suite results:

key: 123456789abcdef0

message: "" (empty string)
mac: a5c10317dc5aa355

message: "a"
mac: 598298ba39e8265b

message: "abc"
mac: 121db704b52f71aa

message: "message authentication code"
mac: 6288beba08a21bb9

message: "abcdefghijklmnopqrstuvwxyz"
mac: dd7a2a779098ac52

message: alphabet in uppercase, in lower case and digits 0 through 9
mac: 1ed27286699c3ad5

message: 8 times "1234567890"
mac: ce4620b8fd9da619


RIPE-MAC3 test suite results:

key: 123456789abcdef09abcdef012345678

message: "" (empty string)
mac: 25159cb1ec098e62

message: "a"
mac: 55dc3747024f4fad

message: "abc"
mac: ce6d95e85f723caf

message: "message authentication code"
mac: d6661f40954ed8ed

message: "abcdefghijklmnopqrstuvwxyz"
mac: 693181c24ea085ae

message: alphabet in uppercase, in lower case and digits 0 through 9
mac: a87d471ba312d3fd
```

```
message: 8 times "1234567890"
mac: 6a4c6e7716ef7da3
```

Chapter 5

IBC hash

Contents

1 Introduction

This chapter describes the integrity primitive IBC hash. IBC hash is a keyed hash function that maps messages to hashcodes. This function is *provably secure*: the probability to find the correct hashcode of a message, without knowing the secret key, is proven to be close to optimal (i.e., exponentially small in the hashcode size), even when the hashcode of one other message is known. Furthermore, this probability is independent of the computing power used.

IBC hash can be used for message authentication between parties who share a secret key. The party who wants to send a message first computes the hashcode of this message. It then sends both message and hashcode—a so called tagged message—to the other party. The receiver of this tagged message also computes the hashcode of the message and verifies that the outcome equals the received hashcode. If they are equal, he has good reason to believe that the message is genuine, i.e., the message originated from the party with whom he shares the key and was not modified in transit.

The design of the IBC hash function is such that it is both provably secure and efficient. The basic operation is a simple modular reduction (giving the efficiency) modulo a secret modulus (giving the security). Furthermore, the function is easy to describe and understand.

On the other hand, there are also three disadvantages for IBC hash, the first of which is inherent in provably secure schemes. As every tagged message reduces the number of possible keys, each key can be used a limited number of times, but if you only want cryptographic security you can just generate keys; for IBC hash a key can be used for only one authentication.

Also, once the size of the hashcode is fixed, one has implicitly put a maximum to the message size, although it is much larger than the size of the hashcode. The last disadvantage is in the number n of bits in the hashcode. To achieve a security level of say 2^{-64}, n has to be larger than 64. Ideally the probability to find the correct hashcode of a message is 2^{-n} (i.e., one divided by the number of possible hashcodes). But IBC hash features a key size twice as long as the hashcode size and then, for long messages, this probability is theoretically impossible.

Note that anyone in possession of the key can find messages that hash to a given hashcode; therefore one should trust everyone in possession 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 algebraic evaluation of the primitive. Finally, in Section 6 the performance aspects of IBC hash are considered, and Section 7 gives some guidelines for software implementation.

This chapter has two appendices. Appendix A contains a straightforward software implementation of IBC hash 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 document are fully described in this chapter. 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, ..., 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

A sequence of n bits $b_0, b_1, \ldots, b_{n-1}$ corresponds to a nonnegative integer B as follows:

$$B := \sum_{i=0}^{n-1} b_i \, 2^{n-i-1}. \tag{1}$$

Hence the first bit b_0 of the sequence is the most significant bit of B.

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.

- For two strings X and Y of length $|X| = n$ respectively $|Y| = m$, the $(n+m)$-bit string $W = X\|Y$ is defined as the *concatenation* of the strings X and Y. That is

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

- 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
$$A = QB + R \quad \text{and} \quad 0 \le R < B.$$
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 nonzero integers X and Y we say that X *divides* Y if $Y \bmod X = 0$. That is, if Y is a multiple of X.

- A *prime* is an integer greater than 1 that is divisible only by 1 and by itself.

2.5 Symbols Used by the Primitive

• M	message, input for IBC hash function,
• X	expanded message,
• T	hashcode, output of IBC hash function,
• m	message length in bits ($|M| = m$),
• n	hashcode length in bits ($|T| \le n$),
• $m^\star$	upper bound for m ($m \le m^\star$),
• l	$63 - (m + 63 \bmod 64)$,
• s	upper bound for length of $m^\star$ ($m^\star < 2^s$),
• t	upper bound for n ($n \le t$),
• P	n-bit prime greater than 2^{n-1},
• V	n-bit number,
• $\mathcal{H}_{P,V}$	IBC hash function with key (P, V),
• $\mathcal{P}_0$, $\mathcal{P}_1$	impersonation and substitution probability, respectively.

3 Description of the Primitive

The IBC hash function maps messages of variable size to hashcodes of fixed size. Let m and n denote the length in bits of messages and hashcodes, respectively. For the description of the function also the numbers $m^\star$, s and t are used. The number $m^\star$ is the maximum value of m as allowed by the security evaluation (for given n, see section 5) and s is a fixed number of bits that are used to describe $m^\star$ (thus $m \leq m^\star < 2^s$); t is a fixed upper bound for the number of bits in a hashcode (hence $n \leq t$). The numbers s and t are both required to be multiples of 64 to make programming the function more efficient. In practice one chooses them as small as possible within these constraints: s and t are the least multiple of 64 larger than or equal to $\lceil \log_2 m^\star \rceil$ and n, respectively.

A key for the hash function is a pair of numbers (P, V), where P is a prime such that $2^{n-1} < P < 2^n$ and $0 \leq V < 2^n$. To compute the hashcode T of a message M, M is first expanded to a number X with a length that is a multiple of 64 bits and that contains m and then compressed using the key.

1. Expansion of an m-bit message $M = (b_0, b_1, \ldots, b_{m-1})$ to the number X:

   - Append a string of zero bits to the message such that the expanded message has a length that is a multiple of 64 bits (if the length is already a multiple of 64 bits, then nothing is concatenated).
     Thus, if $l = 63 - (m + 63 \bmod 64) > 0$, then
     $$b_{m+i} := 0 \quad \text{for } i = 0, 1, \ldots, l-1.$$
   - Append s bits containing the length of the original message in bits. Thus, if $m = \sum_{i=0}^{s-1} m_i \, 2^{s-i-1}$ with $0 \leq m_i < 2$, then
     $$b_{m+l+i} := m_i \quad \text{for } i = 0, 1, \ldots, s-1.$$
   - Append t zero bits. Thus
     $$b_{m+l+s+i} := 0 \quad \text{for } i = 0, 1, \ldots, t-1.$$
   - The number X is now defined as the number corresponding to the expanded sequence
     $$(b_0, b_1, \ldots, b_m, \ldots, b_{m+l}, \ldots, b_{m+l+s}, \ldots, b_{m+l+s+t}).$$
     Observe that the length of X is a multiple of 64 bits.

   Note that if Y is the number corresponding to the string M, then the number X is also given by the equation
   $$X := \left(Y \, 2^{l+s} + m\right) \, 2^t.$$

2. Compression: the integer X is first reduced modulo P and the result is added to V modulo 2^n. Thus
   $$\mathcal{H}_{P,V} := [(X \bmod P) + V] \bmod 2^n.$$

Now the n-bit hashcode T of message M is the number $\mathcal{H}_{P,V}$:
$$T := \mathcal{H}_{P,V}.$$

4 Use of the Primitive

The primitive IBC hash has two intended applications:

- to provide both data integrity and data origin authentication,
- to be used in identification schemes.

Before a message is sent, the secret key is used to compute a keyed hashcode (sometimes also called MAC or tag) from it. This value is sent along with the message and can be checked by the legitimate recipient using the common secret key. If the calculated hashcode is equal to the received value, it is reasonable to assume that the original hashcode was computed from the same message. This holds, since computation of another message with the same hashcode is *proven* to be infeasible for someone not in possession of the secret key (see next section). Moreover, it is reasonable to assume that the message is authentic, as it is infeasible for someone who does not know the secret key to produce a hashcode for a given message. The latter furthermore allows linking of the hashcode to the originator of the information. Therefore, the primitive provides both data integrity and data origin authentication.

Each key may be used only once however, because two tagged messages made with the same key may reveal it. Thus each authentication requires its own secret key and before the sender and receiver can communicate with each other, they must agree on the key. Note that the provable security is only guaranteed if such a key exchange is also performed in a provably secure way and not by some way that is only computationally secure. This implies that IBC hash is better suited for few (long) messages rather than for many (short) messages. Furthermore, the key must be kept secret, because anyone in possession of the key can not only compute the hashcode for any message, but can also compute (many) messages that hash to any given hashcode.

Another application for the primitive is identification schemes, which allow parties to establish that their key sharing counterpart is actually communicating with them at a particular moment. One party supplies the other party with a random challenge. The other party applies the primitive to the challenge sent, and returns the result. The challenging party does the same and compares the calculated hashcode with the returned one. If they are equal, it is reasonable to assume that the other party is in possession of the secret key. Of course the roles can be reversed, so that each party is able to challenge the other. For this unilateral and bilateral authentication it is suggested to use, respectively, the RIPE primitives SKID2 and SKID3 (see Chapter 6). However note that in SKID3 the hash function is used twice, and therefore—when using IBC hash—two different keys must be used.

5 Security Evaluation

5.1 Claimed Properties

In the usual model for authentication there are three parties: a sender, a receiver and a tamperer. The sender wants to communicate some message to the receiver, using a public communication channel; the receiver wants to be sure that the message he receives did come from the sender and was not modified in transit. On the other hand, the tamperer wants to deceive them by getting a message of his own accepted by the receiver. Suppose that the tamperer has the ability to insert messages into the channel and/or to modify existing messages. The first ability is called *impersonation* and the second *substitution*. Let $\mathcal{P}_0$ and $\mathcal{P}_1$ denote the probability that the tamperer can deceive the sender/receiver by impersonation and substitution, respectively.

The IBC hash function is provably secure: the probabilities for both abilities are exponentially small in n. That is:

$$\mathcal{P}_0 = \frac{1}{2^n} \quad \text{and}$$

$$\mathcal{P}_1 < 3 \cdot \frac{m+l+s+t}{2^n}.$$

5.2 Algebraic Evaluation

In [CHB92] it is proven that a necessary and sufficient condition for a tamperer to insert or substitute a message is that he knows a non-zero multiple of the prime number in the used key. Furthermore, this multiple must also be less than $2^{8(m+l+s+t)}$ and this gives the equality and inequality of the previous section. Note that the probability to substitute a message is maximal if m is maximal ($m = m^\star$), but that this probability is much less for small messages. By choosing n and $m^\star$ appropriately, we obtain that large messages can be hashed while keeping the impersonation and substitution probabilities as small as one may require. If $m^\star$ is much larger than $l+s+t$, which is usually the case when using hash functions, then a simple upper bound for the substitution probability is $\mathcal{P}_1 < m^\star \cdot 2^{-(n-2)}$.

Suggested parameter values.
If $n = t = 64$, $s = 64$ and $m^\star = 2^{32}$, then we have $\mathcal{P}_0 = 2^{-64}$ and $\mathcal{P}_1 < 3 \cdot \frac{2^{32}+191}{2^{64}} < 2^{-30}$. Thus the substitution probability is less than 2^{-30}, while messages can be up to 2^{32} bits (or $2^9 = 512$ Megabytes) long.

The substitution probability can be reduced by increasing n and/or decreasing $m^\star$. Furthermore, increasing n allows that $m^\star$ can be increased greatly: an upper bound of 2^{-60} for the substitution probability is obtained by choosing n and $m^\star$ such that $n - 2 - \log_2 m^\star = 60$. For example $n = 128$, $m^\star = 2^{66}$ and $n = 96$, $m^\star = 2^{34}$ both have $\mathcal{P}_1 < 2^{-60}$, while authenticating messages of 2^{43} Megabytes and $2^{11} = 2048$ Megabytes, respectively.

These suggestions are recapitulated in the table below.

n	$m^\star$	$\mathcal{P}_0$	$\mathcal{P}_1$
64	2^{32}	2^{-64}	2^{-30}
96	2^{34}	2^{-96}	2^{-60}
128	2^{66}	2^{-128}	2^{-60}

6 Performance Evaluation

6.1 Software Implementations

The basic operation of IBC hash is a modular reduction. In practical systems the modulus will be 64 or 128 bits long. The argument of the modular reduction is the expanded message and will normally be several orders of magnitude larger as the modulus. The most efficient algoiithm will therefore be the classical algorithm as described in [Knu81]. The implementation of this modular reduction can be very compact.

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 general nature of the C version and the ability to use 64-bit integers in Assembly. The C version can be used for moduli of arbitrary length, while its Assembly counterpart is restricted to a 64-bit modulus. The figures in 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. The C version of the compression function uses about 0.5K of memory, while the Assembly version is about half as large. Doubling the modulus will about half the speed.

	C	Assembly
IBC hash (64-bit modulus)	305 Kbit/s	5.27 Mbit/s

Table 1: Software performance of IBC hash with a 64-bit modulus 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.

6.2 Hardware Implementations

The implementation of a modular reduction with a 64-bit or 128-bit modulus can be extremely fast in hardware. It is expected that a speed of up to a 100 Mbit/s is achievable. However it must be borne in mind that at such speeds the critical path is situated in the I/O interface. RSA implementations are existing alternatives, as they normally provide a modular reduction in their instruction set. The speed will however be reduced with several orders of magnitude.

For a 64-bit modulus the modular reduction can be very efficiently implemented on 32-bit architectures. The modulus is kept in two registers. The message is then reduced as follows. First, the next 32-bit message bits are appended to the 64-bit intermediate reduction result. Next, this 96-bit number is divided by the most significant 32 bits of the modulus. The result of this division is checked by a remultiplication and if necessary adapted. The same ideas can be used to implement a 128-bit reduction.

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 basic operation of IBC hash is a modular reduction of the expanded message modulo an n-bit integer. The value of n will be typically 64 or 128, but other values are possible. As the message will normally be several orders of magnitude larger as the modulus, the most efficient algorithm will be the classical algorithm as described in [Knu81].

References

[CHB92] D. Chaum, M. van der Ham and B. den Boer, "A provably secure and efficient message authentication scheme," available from authors, 1992.

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

A C Implementation of the Primitive

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

The IBC-HASH program computes the IBC hashcode of a file. It uses the following values for the parameters s and t of the description given in Section 3:

$$\begin{aligned} s &= 64 \\ t &= 128 \end{aligned}$$

That is, 64 bits are used to represent the length of the original message and 128 zero bits are subsequently appended. As the value of t is an upper bound for the length n of the hashcode, hashcodes up to 128 bits can be produced with this implementation. The program imposes some additional restrictions on the value of n: it should be at least 32 and a multiple of 16.

The IBC-HASH program `ibc-hash.c` consists of a single C source file. The full listing is given in Appendix A.1. The program is written in ANSI-C. It uses only standard library routines and should be easily portable. (Porting the code to a SPARC station was quite simple.) The program accepts the name of the file to compute the hash on as the argument. The key values are read from the file `ibc-hash.key`. The result is printed on the standard output in hexadecimal format.

The file `ibc-hash.key` contains 2 lines of text. The first line is P and the second line is V, both in hexadecimal. Three example key files can be found in Appendix A.3.

A separate program `ibc-test.c` is provided to generate test patterns. The listing is given in Appendix A.2. It takes the length of the required testfile as a command line argument. The testfile is named `ibc-test.dat` and is generated using a pseudorandom generator and a initial seed. Together with the sample key files of Appendix A.3 the testfile can be used to produce the test values of Appendix B.

A.1 C Source Code for IBC hash

```
/***********************************************************************\
 *                                                                     *
 *   IBCHASH.C    Reference implementation for IBC hash                *
 *                                                                     *
 *   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                                      *
 *                                                                     *
 *  This is a flexible and portable implementation. It is reasonably   *
 *  fast, but a machine-specific implementation would be much faster.  *
 *  The size of the prime is variable in steps of 16 bits. The modulo  *
 *  reduction routines are written for clarity, not speed.             *
 *  This implementation was timed at about 150 Kbits per second on a   *
 *  33 MHz 80386DX based IBM compatible.                               *
 *                                                                     *
\***********************************************************************/

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

#define READ_BUF_LEN    (16)        /* prime at most 128 bits */

/*
** All ANSI C implementations have the following properties:
**  short unsigned integers are at least 16 bits
**  long unsigned integers are at least 32 bits
** This program assumes an ANSI compatible C implementation.
*/

typedef unsigned short int digit;
typedef unsigned long int  ddigit;
typedef unsigned char byte;
typedef int bool;

/* on a 32-bit architecture, it might be faster to replace
** the short int for digit by a normal int.
*/

/* Do not alter these constants */
#define RADIX           (0x10000L)
#define DIGIT_MASK      (0xffff)
#define MAX_DIGIT       (0xffff)
#define DIGIT_LENGTH    (16)
#define DIGIT_MSB_MASK  (0x8000)
```

```
#define EFATAL (255)            /* value passed to exit() on fatal error */

/*
** We will store large numbers as an array of digits. The least significant
** digit is stored first
*/

/*============================================================================
**  General routines
**============================================================================
*/

void fatal( char *format, ... )
{
    va_list ap;

    va_start( ap, format );
    fprintf( stderr, "\nFatal Error:\n" );
    vfprintf( stderr, format, ap );
    va_end( ap );

    exit( EFATAL );
}

void *alloc( size_t size )
{
    void *p;

    p = malloc( size );
    if ( p==NULL )
        fatal( "alloc(%d): not enough memory to allocate this", size );
    return( p );
}

/*============================================================================
**  Modulo reduction routines.
**============================================================================
*/

/* This part forms a separate module. For more speed, it can be implemented
** in machine language. All names start with mod_ to prevent name conflicts.
** The interface has been designed so that the implementor of this module
** can choose his own internal data representation.
*/

int   mod_len;                /* must be >= 2 */
digit mod_MSd1;               /* MDdigit of modulus */
digit mod_MSd2;               /* next digit of modulus */
digit *mod_inv = NULL;        /* 2's complement of modulus */

digit *mod_buffer = NULL;

void mod_set_modulus( int len, byte *m )
```

```
/* Specifies the modulus to be used for subsequent modulo operations
** m points to the modulus (MSbyte first), len is the # bytes.
** This routine can be used to do precomputations if a different modulo
** reduction algorithm is used.
*/
{
    digit *p;
    int   i;

    if ( len<4 )
        fatal( "set_modulus( %d, * ): length must be at least 4 bytes", len );

    if ( len&1 )
        fatal( "set_modulus( %d, * ): length must be even", len );

    mod_len  = len/2;
    mod_MSd1 = (m[0]<<8) + m[1];
    mod_MSd2 = (m[2]<<8) + m[3];

    if ( (mod_MSd1 & DIGIT_MSB_MASK) == 0 )
        fatal( "set_modulus( %d, * ): most significant bit is 0", len );

    if ( mod_buffer != NULL )
        free( mod_buffer );
    mod_buffer = alloc( (mod_len+1)*sizeof( digit ) );

    if ( mod_inv != NULL )
        free( mod_inv );
    mod_inv = alloc( mod_len * sizeof( digit ) );

    /* We now compute the 2's complement by complementing and incrementing */
    for ( i=0; i<mod_len; i++ )
        mod_inv[i] = ~( (m[len-2*i-2] << 8) + m[len-2*i-1] );
    p = mod_inv;
    while ( ( *p = (*p+1) & DIGIT_MASK ) == 0 ) p++;
}

void mod_init_buffer( void )
/* Call this routine before any reduction to clear the internal buffer
*/
{
    int   i;
    digit *p;

    /* Clear the buffer to 0 */
    for ( p = mod_buffer, i = 0; i<mod_len; i++ )
        *p++ = 0;
}

int mod_read_buffer( int len, byte *buf )
/* Read the contents of the reduced buffer into buf MSB first
** The return value is the length of the result in bytes.
*/
```

```
{
    int i;

    if ( len < 2*mod_len )
        fatal( "mod_read_buffer( %d, * ): buffer not long enough to contain"
                "result", len );
    for ( i=mod_len-1; i>=0; i-- ) {
        *buf++ = (mod_buffer[i] >> 8) & 0xff;
        *buf++ = mod_buffer[i] & 0xff;
    }
    return( 2*mod_len );
}

void mod_reduce_buffer( void )
/* The buffer contains a len+1 sized number which is less than
** 2^16 times the modulus. This routine reduces the buffer modulo
** the modulus. For the algorithm of this routine, see Knuth, "The
** art of computer programming", volume 2, 'Seminumerical Algorithms',
** paragraph 4.3.1, pages 255..263.
*/
{
    int    i;
    digit  q;
    ddigit t;

    /* Step D3 (page 258) */
    t = ((ddigit) mod_buffer[mod_len] << DIGIT_LENGTH)
        + mod_buffer[mod_len-1];

    if ( mod_buffer[mod_len] == mod_MSd1 )
        q = MAX_DIGIT;
    else
        q = (digit) (t/(ddigit) mod_MSd1 );

    t -= (ddigit)q * mod_MSd1;
    /*
    ** This is the loop that adjusts q. Note the special case when t is
    ** larger then MAX_DIGIT. This can occur if q was set to MAX_DIGIT
    ** instead of t/mod_MSd1. If the second formula had been used, then q
    ** would have had the value MAX_DIGIT+1, which would give an overflow.
    ** This special case occurs very infrequently.
    ** The loop body can also set t to a value larger than MAX_DIGIT.
    */
    while ( t <= MAX_DIGIT &&
       mod_MSd2*(ddigit)q > (t<<DIGIT_LENGTH)+mod_buffer[mod_len-2] ) {
        /* now we decrease q by one and adjust t as well. */
        q--;
        t += mod_MSd1;
    }

    /* Now we add q times mod_inv to mod_buffer */
    t = 0;
    for ( i=0; i<mod_len; i++ ) {
        t += (ddigit) q * mod_inv[i] + mod_buffer[i];
        mod_buffer[i] = (digit) t & DIGIT_MASK;
        t = ( t>>DIGIT_LENGTH ) & DIGIT_MASK;   /* shift can be signed! */
    }
    if ( ((digit)t & DIGIT_MASK) + mod_buffer[mod_len] != q ) {
        /* A carry occured here, so q was one too large.
        ** Therefore, we will substract mod_inv once
        ** For efficiency reasons we reuse t as the carry variable
        */
        q = 0;
        for ( i=0; i<mod_len; i++ ) {
            t = (ddigit) mod_buffer[i] - mod_inv[i] - q;
            mod_buffer[i] = (digit) (t & DIGIT_MASK );
            q = (digit)(t >> DIGIT_LENGTH) & 1;
        }
    }
}

void mod_reduce( size_t len, byte *buffer )
/* takes the data in the buffer as additional data to reduce
** several calls of this routine can be used to reduce more data.
** The size_t is an ANSI type that indicates a length of a memory block
** It is either an unsigned int or long.
** The length of the buffer should be a multiple of 8.
*/
{
    int i;

    if ( (len & 0x7) != 0 )
        fatal( "mod_reduce( %uld, * ): length is not a multiple of 8",
                (unsigned long) len );
    while ( len ) {
        /* First we shift the buffer 1 digit and add the new digit */
        for ( i=mod_len; i>0; i-- )
            mod_buffer[i] = mod_buffer[i-1];
        mod_buffer[0] = (buffer[0] << 8) + buffer[1];

        /* Reduce the buffer modulo the modulus */
        mod_reduce_buffer();

        /* Adjust the loop pointers now */
        buffer += 2;
        len -= 2;
    }
}

/*=========================================================================
**  File interface routines
**=========================================================================
*/

int  read_len;
byte read_buf[READ_BUF_LEN];        /* number, MSbyte first */
```

```
int hexvalue( char c )
/* return the value of hex digit c or -1 if not legal */
{
    if ( c<'0' )  return( -1 );
    if ( c<='9' ) return( c-'0' );
    if ( c<'A' )  return( -1 );
    if ( c<='F' ) return( c-'A'+10 );
    if ( c<'a' )  return( -1 );
    if ( c<='f' ) return( c-'a'+10 );
    return( -1 );
}

bool hexdigit( char c )
{
    return( hexvalue( c ) >= 0 );
}

void read_hexnumber( FILE *f )
/* This routine reads a hexadecimal number from the file f
** into read_buf and sets read_len. All leading white space and the first
** non-hexdigit character are discarded.
*/
{
    int  c;
    char buf[READ_BUF_LEN*2];
    int  len;
    int  i, j;

    c = getc( f );
    while ( c==' ' || c == '\t' || c == '\n' )
        c = getc( f );

    len = 0;
    while ( len<=2*READ_BUF_LEN && c != EOF && hexdigit( c ) ) {
        buf[len++] = c;
        c = getc( f );
    }

    if ( len == 0 )
        fatal( "read_hexnumber( * ): No hex number in file" );

    if ( len > 2*READ_BUF_LEN )
        fatal( "read_hexnumber( * ): Hex number too long" );

    read_len = (len + 1)/2;  /* the number of bytes in the number */

    /* We now convert the ascii to the binary buffer */
    j = 0;
    for ( i=0; i<read_len; i++ ) {
        if ( i==0 && (len & 1) ) {
            read_buf[i] = hexvalue( buf[j++] );
        }
        else {
            read_buf[i] = hexvalue( buf[j++] );
            read_buf[i] = 16 * read_buf[i] + hexvalue( buf[j++] );
        }
    }
}

void read_modulus( FILE *f )
/* This routine reads a hexadecimal number from the file f
** and sets it as the modulus. All leading white space and the first
** non-hexdigit character are discarded.
*/
{
    read_hexnumber( f );
    mod_set_modulus( read_len, read_buf );
}


#define BUF_LEN (1U<<15)        /* The 'U' makes this value unsigned */

void reduce_file( FILE *f )
/* Reads the file and reduces it modulo the modulus.
** It clears the buffer at the start, padds the file with 0's so that
** the length is a multiple of 8 bytes,
** appends a 64-bit length of the file and then 16 0 bytes. The
** whole sequence of bytes is reduced modulo the modulus.
** This routine can be speeded up significantly by using a bigger buffer.
*/
{
    unsigned long fl;
    byte          buf[8];
    byte          *bp;
    unsigned      l;

    bp = alloc( BUF_LEN );
    mod_init_buffer();
    fl = 0;

    while ( (l = fread( bp, 1, BUF_LEN, f )) == BUF_LEN ) {
        mod_reduce( BUF_LEN, bp );
        fl += l;
    }
    fl += l;
    /* Read the last block, padd to multiple of 8 */
    while ( l&7 ) {
        *(bp+l++) = 0;
    }
    mod_reduce( l, bp );

    buf[0] = buf[1] = buf[2] = buf[3] = 0;
    buf[7] = (fl >>  0) & 0xff;
    buf[6] = (fl >>  8) & 0xff;
    buf[5] = (fl >> 16) & 0xff;
    buf[4] = (fl >> 24) & 0xff;
    mod_reduce( 8, buf );

    buf[4] = buf[5] = buf[6] = buf[7] = 0;
    mod_reduce( 8, buf );
    mod_reduce( 8, buf );
    free( bp );
}

void ibc_hash( FILE *f_mess, FILE *f_key, FILE *f_result )
/* Computes ibc_hash of file f_mess using key in file f_key
** result is written in hex to file f_result
*/
{
    byte     buf[16];
    int      i;
    int      l;
    unsigned c;

    read_modulus( f_key );
    reduce_file( f_mess );
    read_hexnumber( f_key );
    l = mod_read_buffer( 16, buf );

    if ( l != read_len )
        fatal( "ibc_hash( *, *, * ): prime length != blinding term length" );

    /* Add the blinding term */
    c = 0;
    for ( i=l-1; i>=0; i-- ) {
        c += buf[i] + read_buf[i];
        buf[i] = c % 256;
        c /= 256;
    }

    for ( i=0; i<l; i++ )
        fprintf( f_result, "%02X", buf[i] );
    fprintf( f_result, "\n" );
}

/*============================================================================
**  Main routine
**============================================================================
*/

int main( int argc, char *argv[] )
{
    FILE *fi, *fk;

    printf( "IBC-HASH computation program version 0.0\n" );

    if ( argc != 2 ) {
        printf( "Usage: IBC-HASH <filename>\n"
                "Computes hash for file using keys in \"ibc-hash.key\"\n"
                );
        exit( 1 );
    }
    fi = fopen( argv[1], "rb" );
    if ( fi==NULL ) {
        printf( "Sorry, I cannot find the file \"%s\"\n", argv[1] );
        exit( 2 );
    }
    fk = fopen( "ibc-hash.key", "rt" );
    if ( fk==NULL ) {
        printf( "Sorry, I cannot find the file \"ibc-hash.key\"\n" );
        exit( 3 );
    }
    ibc_hash( fi, fk, stdout );
    return( 0 );
}

/*======================== end of IBCHASH.C ===============================*/
```

A.2 Program for Generating Test Programs

```
/***********************************************************************\
 *                                                                     *
 *   ibc-test.c   A program to generate test files to be used with     *
 *                 ibc-hash.c                                          *
 *                                                                     *
 *   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                                      *
 *                                                                     *
\***********************************************************************/

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

typedef unsigned char byte;


byte buf[55]= "IBC-HASH test. Copyright (c) 1992 by RIPE consortium   ";
int  bp= 0;

int write_testfile( unsigned long l, FILE *f )
{
    while ( l-- ) {
        if ( putc( buf[bp], f ) != buf[bp] ) {
            fprintf( stderr, "\nError while writing output file" );
            return( 1 );
        }
        buf[bp] = ( buf[bp] + buf[(bp+55-24)%55] ) % 256;
        bp = (bp+1)%55;
    }
    return( 0 );
}

void main( int argc, char *argv[] )
{
    FILE *f;

    printf( "IBC-TEST program to generate IBC-HASH test files\n" );

    f = fopen( "ibc-test.dat", "wb" );
    if ( f == NULL )
        fprintf( stderr, "\nCould not create file \"ibc-test.dat\"\n" );
    else
        if ( argc != 2 ) {
            fprintf( stderr, "Usage: IBC-TEST <size>" );
        }
        else {
            long l = atol( argv[1] );
            if ( l<0 || l>16777216L )
                fprintf( stderr, "\nIllegal size paramter\n" );
            else
                write_testfile( l, f );
        }
}
```

A.3 Sample Key Files

The following three files are sample key files. They all contain 2 lines. The first line is P and the second is V, both in hexadecimal.

```
IBC-HASH.KE1:

8537366B9856CCE7
7589024781647928


IBC-HASH.KE2:

AEC5E2D5F4BAA261
FF0055AAFF0055AA


IBC-HASH.KE3:

FFAEF21A73E83E3F
4782901478657483
```

B Test Values

The following table gives the hashcodes of 4 different files produced by the program of Appendix A.2 for the 3 keys given in Appendix A.3.

Size	Key1	Key2	Key3
0	7589024781647928	FF0055AAFF0055AA	4782901478657483
55	C8612196C379F9EA	2099C76293A7BE70	DB10CD3111A31F47
100000	DA3E02BD71232B61	6BB6E2714051B6F4	D1A85FDF233EA996
1048575	E40FFB3AD650EB92	51E2A0B8245FB653	B12B20E9E1D6A2E4

Chapter 6

SKID

Contents

1 Introduction

The secret key identification protocols SKID2 and SKID3 provide entity authentication.

SKID2 provides unidirectional authentication after two passes. That means after having sent out a challenge to her/his communication partner and having received the correct reply, a user of this protocol has good reason to assume that she/he is communicating with whom she/he thinks.

Using SKID2, the authentication is unidirectional, the authenticity of the other partner is not checked. However, an extension of the protocol, SKID3, which provides authentication of the other partner as well by an additional pass, is also described.

Both SKID2 and SKID3 are based on a secret key only known to the two parties involved, and the application of a keyed hash function using this key. The keyed hash function to use is not specified, but it is suggested to use RIPE-MAC described in Chapter 4 of this report. Random numbers are used so that with high probability in different invocations of the protocol different values are exchanged.

It should be noted that entity authentication can also be achieved by using asymmetric cryptographic algorithms. Examples are the primitives RSA and COMSET described in Chapter 7 and Chapter 8 of this report.

The structure of this description of SKID2 and SKID3 is as follows. In order to avoid any ambiguities in the description, the notation and definitions used are fixed in section 2. In section 3 the primitive is described and in section 4 its purpose is explained. In section 5 security aspects of the primitive are discussed. This includes claimed properties and the results of the algebraic evaluation of the primitive. Finally, in section 6 the performance of the protocol is considered.

2 Definitions and Notation

2.1 Introduction

In order to obtain a clear description of the primitive, the notation and definitions used in this document are fully described in this chapter. 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.

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 document 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 $2^{64} - 1$.

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,

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

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

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

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.
- For two strings $X = x_0, x_1, \ldots, x_{n-2}, x_{n-1}$ of length $|X| = n$ and $Y = y_0, y_1, \ldots, y_{n-2}, y_{m-1}$ of length $|Y| = m$, the $(n + m)$-bit string $W = w_0, w_1, \ldots, w_{n+m-1} = X\|Y$ is defined as the *concatenation* of the strings X and Y. That is

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

  If the operation $\|$ is applied to operands which are words, the corresponding representation as a sequence of bits is to be used.

3 Description of the Primitive

SKID2 and SKID3 are secret key identification protocols. In the participants in the protocol are denoted by $\mathcal{A}$ (Alice) and $\mathcal{B}$ (Bob). SKID2 provides entity authentication of $\mathcal{B}$, SKID3 provides mutual entity authentication of $\mathcal{A}$ and $\mathcal{B}$.

The SKID2 protocol for participants $\mathcal{A}$ and $\mathcal{B}$ is based on the following requirements:

1. The secret key K is only known to $\mathcal{A}$ and $\mathcal{B}$.
2. $\mathcal{A}$ wants to communicate with $\mathcal{B}$ and knows his distinguished name B. This distinguished name must identify $\mathcal{B}$ uniquely.
3. The keyed hash function $\mathcal{H}_K()$ to use must be specified. This concept is defined in Part II of this report. The requirements for this function are discussed in section 5.2 . It is suggested to use the primitive RIPE-MAC described in chapter 4 of this report.

The protocol SKID2 consists of the following steps:

- $\mathcal{A}$ chooses a random 64-bit word $R_{\mathcal{A}}$. Each of the possible 2^{64} words should be selected with equal probability. This random word is also the first message $M_1 := R_{\mathcal{A}}$. $\mathcal{A}$ sends M_1 to $\mathcal{B}$.
- $\mathcal{B}$ chooses a random 64-bit word $R_{\mathcal{B}}$. Each of the possible 2^{64} words should be selected with equal probability. The message he sends to $\mathcal{A}$ is defined by $M_2 := R_{\mathcal{B}} \| \mathcal{H}_K(R_{\mathcal{A}} \| R_{\mathcal{B}} \| B)$.
- $\mathcal{A}$ extract the 64-bit word $R_{\mathcal{B}}$ from M_2, computes $\mathcal{H}_K(R_{\mathcal{A}} \| R_{\mathcal{B}} \| B)$ and checks whether the result is the same as the corresponding part of the message M_2 she received. If it is, she has good reason to assume that she is communicating with $\mathcal{B}$, otherwise the authentication failed.

These are also the first steps of the protocol SKID3, but, if everything went well so far, SKID3 consists of two more steps:

- $\mathcal{A}$ computes her message M_3 defined by $M_3 := \mathcal{H}_K(R_{\mathcal{B}} \| A)$ and sends it to $\mathcal{B}$.
- $\mathcal{B}$ computes $\mathcal{H}_K(R_{\mathcal{B}} \| A)$ and checks whether the result is equal to the received M_3. If it is he has good reason to assume that he is communicating with $\mathcal{A}$, otherwise the authentication failed.

For SKID3, in addition to the requirements given for SKID2, $\mathcal{B}$ should want to communicate with $\mathcal{A}$ and know her distinguished name A.

The descriptions of the protocols can be summarized in the following bird's eye views.

$\mathcal{A}$		$\mathcal{B}$
choose random word $R_\mathcal{A}$		
$M_1 := R_\mathcal{A}$		
	$\xrightarrow{M_1}$	
		choose random word $R_\mathcal{B}$
		compute $M_2 := R_\mathcal{B} \Vert \mathcal{H}_K(R_\mathcal{A} \Vert R_\mathcal{B} \Vert B)$
	$\xleftarrow{M_2}$	
verify $\mathcal{H}_K(R_\mathcal{A} \Vert R_\mathcal{B} \Vert B)$		

Figure 1. The SKID2 protocol

$\mathcal{A}$		$\mathcal{B}$
choose random word $R_\mathcal{A}$		
$M_1 := R_\mathcal{A}$		
	$\xrightarrow{M_1}$	
		choose random word $R_\mathcal{B}$
		compute $M_2 := R_\mathcal{B} \Vert \mathcal{H}_K(R_\mathcal{A} \Vert R_\mathcal{B} \Vert B)$
	$\xleftarrow{M_2}$	
verify $\mathcal{H}_K(R_\mathcal{A} \Vert R_\mathcal{B} \Vert B)$		
compute $M_3 := \mathcal{H}_K(R_\mathcal{B} \Vert A)$		
	$\xrightarrow{M_3}$	
		verify $\mathcal{H}_K(R_\mathcal{B} \Vert A)$

Figure 2. The SKID3 protocol

4 Use of the Protocol

SKID2 provides entity authentication of $\mathcal{B}$. This means that $\mathcal{A}$ has, after having completed the protocol successfully, good reason to believe that she is communicating with $\mathcal{B}$.

Note that for SKID2 $\mathcal{B}$ has no reason to believe that he is communicating with $\mathcal{A}$. If this is desired as well, SKID3 can be used which additionally provides entity authentication of $\mathcal{A}$.

5 Security

5.1 Claimed Properties

If initially, the key K is known to users $\mathcal{A}$ and $\mathcal{B}$ only, the SKID2 and SKID3 protocols are expected to satisfy the following:

- If the users share the same key, and both follow the instructions, both SKID2 and SKID3 will always complete successfully.
- Even after watching a large number of conversations between $\mathcal{A}$ and $\mathcal{B}$, or having taken part in a number of conversations with $\mathcal{B}$, it is infeasible for any third party on his own to execute SKID2 successfully with $\mathcal{A}$. This remains true, even if different instances of the protocol are allowed to take place simultaneously.
- Even after watching a large number of conversations between $\mathcal{A}$ and $\mathcal{B}$, or having taken part in a number of conversations with $\mathcal{A}$ and $\mathcal{B}$, it is infeasible for any third party on his own to execute SKID3 successfully with $\mathcal{A}$ or with $\mathcal{B}$. This remains true, even if different instances of the protocol are allowed to take place simultaneously.

Here, and in what follows, by "infeasible" we mean computationally far out of reach of current technology.

5.2 Algebraic Evaluation

The properties claimed rely on the following property, which we assume is satisfied by $\mathcal{H}_K()$:

- If the key K is not known, the following problem is infeasible to solve: first choose some inputs $R_1, R_2, \ldots$ and receive $\mathcal{H}_K(R'_1), \mathcal{H}_K(R'_2), ..$, where R_i is a prefix of R'_i. This is precisely the situation an attacker will find himself in by eavesdropping previous SKID protocols. Now compute some R_0 and $\mathcal{H}_K(R_0)$, such that R_0 is not equal to any of $R'_1, R'_2, \ldots.$

Consider an enemy trying to impersonate $\mathcal{B}$ when talking to $\mathcal{A}$ in the SKID2 protocol. If the inputs to $\mathcal{H}_K()$ are chosen at random, there is only a negligible probability that the current random input received from $\mathcal{A}$ has been used before. The assumption above on $\mathcal{H}_K()$ therefore implies that the enemy cannot on his own compute the $\mathcal{H}_K()$ value he needs to complete the protocol.

Hence the only remaining possibility is to fool a user that knows K (i.e., $\mathcal{A}$ or $\mathcal{B}$) into computing the value needed. This must be done while the current instance of the protocol is still running.

The only way to get answers from $\mathcal{A}$ or $\mathcal{B}$ is of course to start an instance of the protocol with them. Doing this with $\mathcal{B}$ would mean that the enemy would merely act as a relay between $\mathcal{A}$ and $\mathcal{B}$. However this is not a successful attack for the enemy:

the purpose of the protocol is to prove to $\mathcal{A}$ that $\mathcal{B}$ is active in the current protocol-instance, and this would in fact be the case here. The only remaining possibility is to start a simultaneous protocol instance with $\mathcal{A}$, set up such that $\mathcal{A}$ is the party supplying the $\mathcal{H}_K()$-output. However, any $\mathcal{H}_K()$-input used here will have the name of party $\mathcal{A}$ inserted into it. Therefore the output will be useless to the enemy because his original purpose was to impersonate $\mathcal{B}$.

The only remaining attack to consider is an intermediate person, who simply forwards the different parts of the protocol, acting as a relay so to speak. However, this attacker would never be able to learn the secret key. In fact this type of attack, known as the mafia attack, will only work in one of the following instances in general, completely independent of the actual schemes:

1. Key exchange without authentication.
2. An identification scheme with no connection to the subsequent communication (access control).

A similar argument can be made for SKID3: the enemy cannot impersonate $\mathcal{B}$ for the same reasons as above (the first part of SKID3 equals SKID2). If the enemy is trying to impersonate $\mathcal{A}$, starting a parallel session with $\mathcal{A}$ as above does not make sense. Likewise, a parallel session with $\mathcal{B}$ will be useless because of the names inserted on the $\mathcal{H}_K()$-inputs, and by the property of $\mathcal{H}_K()$, the value he receives from $\mathcal{B}$ in the second protocol message will not help him either.

6 Performance Evaluation

SKID2 requires 2 passes of communication. The computations required in total are: the generation of two 64-bit random integers, and two applications of the keyed one-way function. SKID3 requires 3 passes of communication. The computations required in total are: the generation of two 64-bit random integers, and four applications of the keyed one-way function. The computational effort is equally balanced between $\mathcal{A}$ and $\mathcal{B}$. The actual performance depends on the choice and implementation of the function $\mathcal{H}_K()$.

Chapter 7

RSA

Contents

1 Introduction

This chapter describes the integrity primitive RSA.

RSA in its basic form is a so called public key system, where each user has a private key, known only to him. Corresponding to this private key, there exists a public one, which may be known to anyone. Data processed by one key can be recovered using the other one. Yet, there is no feasible way known by which the private key can be found from the public one. In the case of RSA, the difficulty of finding the secret key is based on the difficulty of factoring large numbers.

Using proper modes of use, RSA can be used for digital signatures, and for distributing keys for authentication systems.

The name RSA is derived from the inventors of the algorithm: Rivest, Shamir and Adleman, who presented the algorithm in [RSA78].

In Section 2 of this chapter, we present the definitions and notation used, in order to avoid ambiguities in the description. Section 3 describes the primitive proper, Section 4 gives the recommended modes of use. Section 5 contains results of the algebraic evaluation of the primitive, while Section 6 contains the results of the performance evaluation. Finally, Section 7 gives some guidelines to software implementation of the primitive, and there is an appendix containing some test values.

2 Definitions and Notation

2.1 Introduction

In order to obtain a clear description of the primitive, the notation and definitions used in this document 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.

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 document a *byte* is defined as an 8-bit quantity. A byte is considered to be a nonnegative integer. That is, it can take on the values 0 through $2^8 - 1 = 255$.

A sequence of n bits b_0, b_1, ..., b_{n-1} is interpreted as an nonnegative integer B in the following way. The bits are considered as the binary representation of B, the more significant bits being first in the sequence. That is,

$$B := \sum_{i=0}^{n-1} b_i 2^{n-i-1}.$$

Conversely, an interpretation of a number as a sequence of bits is defined by this equation, if a bit length for this number is fixed.

A sequence of $8n$ bits b_0, b_1, ..., 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,

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

2.4 Definitions and Basic Operations

- A *string* is a sequence of bits.
- 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.
- For an integer N, the *length* of N is defined as the length of the shortest binary representation of N. The length of N is denoted as $|N|$.
- For two strings $X = x_0, x_1, \ldots, x_{n-1}$ and $Y = y_0, y_1, \ldots, y_{m-1}$, the $(n+m)$-bit string $W = X\|Y$ is defined as the *concatenation* of the strings X and Y. That is,

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

- 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.

- For two strings $X = x_0, x_1, \ldots, x_{n-1}$ and $Y = y_0, y_1, \ldots, y_{m-1}$, the string $U = X \oplus Y$, is defined as the bitwise *XOR* of X and Y, where, if the strings are of different length, the shorter of the two is preceeded by 0-bits in order to make the lengths equal. For two bits x and y the *XOR* is defined as $(x + y)$ mod 2. This notation is also used for nonnegative integers. In this case its shortest binary representation as described above is used.

- The notation "$X \equiv Y \pmod{N}$" (X is equivalent to Y modulo N) is used to indicate that $X \bmod N = Y \bmod N$.
- For two nonzero integers X and Y we say that X *divides* Y if $Y \bmod X = 0$. That is, if Y is a multiple of X.
- For two nonnegative integers X and Y, not both zero, the *greatest common divisor* $\gcd(X, Y)$ is defined as the greatest positive integer that divides both X and Y.
- An integer X is *invertible modulo* N if $\gcd(X, N) = 1$.
- An integer X that is invertible modulo N is said to be a *quadratic residue modulo* N if there is an integer Y such that $X \equiv Y^2 \pmod{N}$.
- A *prime* is an integer greater than 1 that is divisible only by 1 and by itself.
- A *composite* is an integer greater than 1 that is not a prime. A composite can uniquely be written as the product of at least two (not necessarily different) primes.
- When a is invertible modulo n, we let $(\frac{a}{n})$ denote the *Jacobi symbol* of a modulo n (see [Kob87]). When $n = pq$, where p and q are primes, the Jacobi symbol of a modulo n can be defined as

$$\left(\frac{a}{n}\right) = (a^{(p-1)/2} \bmod p) \cdot (a^{(q-1)/2} \bmod q).$$

Note that this number is always 1 or -1. See [Kob87] for the mathematical background.

2.5 Symbols

In what follows we also use lower case letters for numbers to facillitate the reading of formulas.

- p, q will denote prime numbers.
- n will be the *modulus*, the product of p and q.
- k will denote $|n|$, the bit length of n.
- e, d will denote the public, respectively the secret exponent.
- P, S denote the operation with the public, respectively the secret key in the RSA system.
- h denotes a hash function.

- $RR(x)$ denotes the representative element for the hashcode x, computed as defined below in this section.
- $SIG_S(M)$ is the digital signature of M, computed using secret key S.
- l denotes the length of an authentication key to be exchanged.

2.6 The Redundancy Function *RR*

The ISO/IEC standard 9796 ([ISO91]) describes a redundancy function RR. The scheme described in the standard works for any length of input, in this document only the special case of input lengths which are multiples of 8 bits is given. In our case, the input will always be a hashcode.

Let the hashcode H be the concatenation of z bytes h_i

$$H = h_0 \parallel h_1 \parallel \cdots \parallel h_{z-1}.$$

The redundancy scheme requires that the inequality $16z \leq k+2$ holds. This should be no problem for the value 16 of z, as suggested in this report, since RSA moduli should be larger anyway.

A number t is determined from k by

$$t := (k + 14) \text{ div } 16.$$

Now t bytes w_i $(0 \leq i < t)$ are determined from the hashcode by

$$w_i := h_{z-1-((t-1-i) \bmod z))}.$$

Basically this just means that the bytes of H are inserted at the beginning of the string multiply until a length of t bytes is reached.

For $0 \leq i < 2t$ the byte u_i is determined by

$$u_i := \begin{cases} w_{(i-1)\text{div}2} & \text{if } i \text{ is odd} \\ Sh(w_{(i-1)\text{div}2}) & \text{if } i \text{ is even,} \end{cases}$$

where for nonnegative integers a and b less than 16

$$Sh(16a + b) := 16\Pi(a) + \Pi(b)$$

and the values of the function $\Pi()$ are determined by the following table:

x	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\Pi(x)$	14	3	5	8	9	4	2	15	0	13	11	6	7	10	12	1

The bytes u_i are concatenated to form a string

$$U = u_0 \parallel u_1 \parallel \cdots \parallel u_{2t-2} \parallel u_{2t-1}.$$

The bits of U are denoted by b_i

$$U = b_0, b_1, \ldots, b_{16t-2}, b_{16t-1}.$$

The $k-1$ bit number I is defined by

$$I := \sum_{i=0}^{k-2} v_{k-2-i}\, 2^i,$$

where the bits v_i are

$$v_i = \begin{cases} 1 & \text{if } i = 0 \\ b_i & \text{if } 0 < i < k+6-16z \\ 1-b_i & \text{if } i = k+6-16z \\ b_i & \text{if } k+6-16z < i \leq k-10 \\ b_{i-4} & \text{if } k-10 < i \leq k-6 \\ 0 & \text{if } i = k-5 \\ 1 & \text{if } i = k-4 \\ 1 & \text{if } i = k-3 \\ 0 & \text{if } i = k-2. \end{cases}$$

In the case of plain RSA (see section 3.1) the redundancy scheme is finished, so we define

$$RR(H) = I.$$

In the case of the Rabin variant of RSA (see section 3.2) the result $RR(H)$ of the redundancy function is defined by

$$RR(H) = \begin{cases} I & \text{if } \left(\frac{I}{n}\right) = +1 \\ I/2 & \text{if } \left(\frac{I}{n}\right) = -1. \end{cases}$$

For the Rabin variant we therefore need to compute the Jacobi symbol, or equivalent information. To this end, any of the following 3 methods can be used:

1. If the prime factors of n are available (which will often be the case due to optimization of arithmetic modulo n), one can simply use the definition

   $$\left(\frac{I}{n}\right) = (I^{(p-1)/2} \bmod p) \cdot (I^{(q-2)/2} \bmod q).$$

   However this is not very effective, so the following two methods are preferable.

2. Without knowledge of the factors of n one can calculate the value of the Jacobi symbol as described in Chapter 10 of this report.

3. This final method does not require explicit computation of the Jacobi symbol, and is particularly efficient if the public exponent e is small. For maximum efficiency, it requires that the number $T := 2^{-d} \bmod n$ has been precomputed and stored as part of the secret key.

   The goal is to compute directly $S(RR(H))$. To do this, we first compute $V := I^d \bmod n$ and $V^e \bmod n$. Then,

$$S(RR(H)) = \begin{cases} V & \text{if } V^e \bmod n = I \text{ or } V^e \bmod n = n - I \\ \\ VT \bmod n & \text{otherwise.} \end{cases}$$

3 Description of the Primitive

3.1 Plain RSA

Any user of the RSA system must generate a pair of keys, a secret and a public one. This is done by choosing at random two large primes p, q, and an odd public exponent e, which must satisfy that

$$\gcd(e, (p-1)(q-1)) = 1.$$

The secret exponent d is defined to be the smallest non-negative integer satisfying

$$ed \bmod (p-1)(q-1) = 1.$$

See Chapter 9 for detailed information on how to generate these numbers and for possible refinements.

The number n is defined to be $n = pq$ and is called the modulus. It is recommended to choose p, q such that k, the bit length of n is between 512 and 1024. See Chapter 9 for details.

We can now define the public and secret keys:

- Public key P: $n = pq$ and e.
- Secret key S: n, d, and optionally p, q.

The operations with the public and secret key can take any number m as input, that satisfies $0 \leq m < n$. They produce as output a number in the same range. The public key operation is defined as follows (for notational convenience we shall reuse the symbols P and S. It will be clear from the context exactly which meaning is intended):

$$P(m) := m^e \bmod n$$

and the secret key operation is defined by:

$$S(m) := m^d \bmod n.$$

It is shown in [RSA78] that P and S are inverses of each other, i.e.,

$$P(S(m)) = S(P(m)) = m$$

for all m in the range.

3.2 The Rabin Variant

A variant of RSA is known as the Rabin system. In this variant, p, q are chosen such that

$$p \bmod 8 = 3 \quad \text{and} \quad q \bmod 8 = 7.$$

Chapter 9 contains guidelines for generating p, q to satisfy this. The public exponent e is always 2, while the secret exponent d is defined to be

$$d := \frac{(p-1)(q-1)+4}{8}.$$

The definitions of P and S are the same as for RSA. Using similar methods as in [RSA78] it can be shown that $P(S(m)) = S(P(m)) = m$ if $0 \leq m < n$ and m is a quadratic residue modulo n.

4 Use of the Primitive

4.1 Digital Signatures

We describe here a mode of use for RSA that allows generation of a digital signature on a binary string M of arbitrary length. Let h denote one of the hash functions recommended in this report. Then $h(M)$, i.e., the hashcode of M, is a binary string of length 128 bits, independently of the length of M.

The other ingredient we need is the ISO/IEC standard 9796 [ISO91], which contains the description of a *redundancy function* which takes a binary string as input and produces an output at least twice as long. We denote the final result of these operations by $RR(X)$ where X was the original input to the redundancy function. This number is known in the terminology of the standard as the *representative element*. In the section 2.6, we give details on how to compute the RR function. We now define the digital signature on M using secret key operation S as:

$$SIG_S(M) := \text{the smallest of } S(RR(h(M))) \text{ and } n - S(RR(h(M))).$$

A digital signature may be checked as follows: given the message M, the signature $SIG_S(M)$, and the public key operation P corresponding to S, we define the signature to be valid, if and only if

$$P(SIG_S(M)) = RR(h(M)) \quad \text{or} \quad P(SIG_S(M)) = n - RR(h(M)).$$

This holds for both plain RSA and for the Rabin variant (e=2).

4.2 Forwarding of Authentication Keys

This mode of use enables user B to send a message to user A, such that after A has processed the message, the two users share a key to a system for conventional message authentication, such as RIPE-MAC described in Chapter 4 of this report.

We assume that both A and B have generated a pair of RSA keys, $(P_A, S_A), (P_B, S_B)$, and that each knows the public key of the other. This can be accomplished using a public key directory or public key certificates. Let n_A be the modulus used by A, of length k_A, and let n_B be the modulus used by B, of length k_B.

We let l denote the length in bits of the key to be shared. In general, l will depend on the message authentication system where the key will be used. But we require that l is at most $k_A/2$.

B will now execute the following:

1. Choose at random a number r, such that $0 \leq r < n_A$ and the length of r is at least $k_A/2$ (see Chapter 9 for a discussion of random numbers). If P_A is a key for the Rabin variant of RSA, i.e., public exponent 2, then put $x := r^2 \bmod n_A$. Otherwise, put $x := r$.

2. Compute $y := P_A(x)$, and let the key K be the least significant l bits of y.

3. Compute the digital signature $SIG_{S_B}(x)$. Put $c := SIG_{S_B}(x) \oplus RR(h(x))$. Also compute $z := P_A(y)$. Send c, z to A.

After receiving c, z, A does the following:

1. Put $x := S_A(S_A(z))$.

2. Put $\sigma := c \oplus RR(h(x))$. Using P_B, test whether $\sigma = SIG_{S_B}(x)$. If not, reject the message and stop (in practice, more may have to be done, depending on the application — such as notifying B that the message was rejected). Otherwise continue.

3. Compute the key K as the least significant l bits of $P_A(x)$.

The above description of A's algorithm has been optimized for the case where a small public exponent is used, so that the P-operation is much faster than the S-operation: if $d^2 \bmod (p-1)(q-1)$ has been precomputed and stored as part of the secret key, x can be computed in one exponentiation as $x = z^{d^2 \bmod (p-1)(q-1)} \bmod n$. The subsequent cost of computing $P_A(x)$ is negligible if e is small. If e is as large as d, the intermediate result $S_A(z) = P_A(x)$ of step 1 should be saved.

5 Security

5.1 Claimed Properties

The signature mode is expected to have the following property:

- given a public key P, and valid signatures on messages in a set M, it is infeasible to come up with a message m not in M and a valid signature on m. This remains true, even if the attacker gets to choose the messages in M.

The forward authentication key mode is expected to satisfy the following:

- If A accepts the message he gets, the owners of S_A and S_B are both capable of computing the same K, and it is infeasible for any third party to compute K.

- For large key lengths (which in practice means $l \geq 56$), the key resulting from an execution of B's algorithm is not easily controllable. More precisely, let K_0 be an l-bit key selected by some efficient method. Then the following is infeasible: given K_0, choose the initial input x such that the resulting message c, z will be accepted by A, and such that the key associated with c, z equals K_0 with probability significantly larger than 2^{-l}.

Remark: one reason for considering the second property is that it gives some extra protection against the case where B's source of random bits is fallacious, and outputs for example a highly patterned string of bits (the all-1 string is an extreme example). This situation will always be problematic because the number of possible keys will be limited. However, direct use of such a bitstring as a key in subsequent authentication could be particularly dangerous, because some authentication algorithms are weaker when such keys are used. The second property ensures that no such pattern is likely to show up in the key produced.

5.2 Algebraic Evaluation

The security of the use of RSA is based on the belief that factoring is a hard problem, i.e., that computing p, q from $n = pq$ is infeasible for large n. This is the problem one needs to solve to find a secret RSA key from the corresponding public one. From current state of the art in factoring algorithms, and the previous development, there is no indication that factoring will become feasible in general. At present, factorization of 512-bit numbers with two primes factors of comparable size is out of reach, even with the most efficient methods. From what is known today, the hardness of the problem increases quickly with increasing bit size, and it will therefore be easy to defend against developments in computing equipment that are unknown at present. Factoring of 1024-bit numbers is generally thought to be totally out of the question in any forseeable future. More details on this can be found in Chapter 9.

Breaking RSA is at most as hard as factoring the modulus used, and is in fact precisely as hard as finding the secret key from the public one. Whether inverting the P-function is also equivalent to factoring is an open problem, however, as far as ordinary RSA is concerned. The P-function of the Rabin variant, however, is provably equivalent to factoring, as long as one uses the algorithm in its pure form, i.e., without a mode of use. This, on the other hand, means that the system is open to a chosen message attack when used for signatures. Using the mode of use recommended here solves this problem, but also means that the equivalence to factoring is not provable anymore. Hence, ordinary RSA and Rabin are on equal footing as far as this aspect is concerned.

5.2.1 Security of the Signature Mode

If RSA is used directly for digitial signatures, it is possible to use knowledge of legitimately signed messages to generate new, signed messages, without knowing the secret key. For example, it always holds that

$$S(M_1) \cdot S(M_2) \bmod n = S(M_1 \cdot M_2 \bmod n),$$

and so if signatures on M_1 and M_2 are known, an enemy can always sign the "message" $M_1 \cdot M_2 \bmod n$. Another attack puts $M := P(R)$ for some number R which then becomes a valid signature of M. Although such new messages are unlikely to be meaningful, the security of this system is clearly not optimal.

It is the purpose of the redundancy function RR to defend against these attacks. The properties of RR were analyzed in [GQWL91], and the evaluation in RIPE has revealed nothing that goes against the results obtained there. Using the hash function before RR serves as an extra precaution, and adds the benefit that messages can be of arbitrary lengths.

Note, however, that the hash function must be secure, i.e., it should be hard to find different messages with the same hashcode. It is therefore recommended to use a carefully analyzed hash function, such as RIPEMD or MDC-4 described in Chapter 2 and Chapter 3 of this report.

5.2.2 Security of the Forward Authentication Key Mode

For the first property of this mode, note that given a message c, z, it is clear that A can compute a key with the right connection to z. On the other hand, no one else can do this efficiently, since first z is an RSA ciphertext that requires knowledge of the secret key for decryption, and second c's only connection to x is that it depends on $h(x)$ — security of the hash function means that x cannot be computed efficiently from $h(x)$.

Assume now that A accepts the message c, z as having originated from B, i.e., the signature he checks turns out to be valid. We claim that this means that B must have generated the message, and is the only other party capable of computing the key. Since the processes leading from x to z and from x to c are both one-way functions, assuming security of RSA, it is a reasonable conjecture (supported in part by the conjecture from [MiSc91]) that a valid pair c, z can only be generated by first choosing x and then computing the pair. Hence the fact that A accepts the pair is evidence that *some* other user knows x and therefore the key. Moreover, because $RR(h(x))$ is XORed onto the signature to produce c, $h(x)$ is difficult to compute from c. Therefore, once the pair is generated, the signature hidden in c cannot be easily replaced by the signature of another party, and so A can conclude that the other party capable of computing the key must be B.

For the second property, note that the key K is extracted from the output of a one-way function that B cannot easily invert, namely P_A, and that B must know the input value x in order to be able to sign it (A would not accept the message without the signature). So to control the value of K, only three possibilities seem to be open to B:

- Invert P_A on a random output value producing the desired key. This is infeasible because B does not know S_A.
- Choose carefully an input value such that the intermediate results and the output can be controlled. The obvious way to do this is to choose a small input value so that no modular reductions will take place during the computation of P_A. But this is prevented by the conditions on the choice of x.
- Generate many keys with randomly chosen input, and hope that one will equal the desired key value. With current state of the art, this is infeasible if the key has length at least 56.

It therefore seems reasonable to conjecture that B cannot control the value of K.

6 Performance Evaluation

6.1 Software Implementations

Both the P and S operations in plain RSA (see Section 3.1) are modular exponentiations, where the modulus is the product of two large primes. In practical systems both primes are either 256, 384 or 512 bits long, resulting in, respectively a 512, 768 or 1024-bit modulus. The basic multiprecision operations needed to implement such a modular exponentiation are the multiplication of two integers, the squaring of an integer and the modular reduction of a integer (see Chapter 10 for an algorithm). For the multiplication and squaring optimized versions of the classical algorithms are used, as described in [Knu81]. The modular reduction is implemented according a method due to P.L. Montgomery, allowing a reduction in almost the same time as a multiplication [Mon85].

Using knowledge of the prime factors of the modulus, the S operation can be speeded up with a factor 3 to 4. That is, using the so-called Chinese Remainder Theorem (CRT) the S operation is basically reduced to two exponentiations modulo the two prime factors, being only half the length of the modulus, see [QuCo82]. To improve the performance of the P operation of plain RSA a small public exponent can be chosen in the key generation, such as $2^{16} + 1$. Since the public exponent of the Rabin variant is always 2, its P operation is just a modular squaring.

Both a C and a 80386 Assembly language implementation are considered. The C version has the advantage of being portable. The considerable difference in performance between the C and the Assembly language version is due to the general nature of the C code: it can be used for integers of arbitrary length, whereas the Assembly versions use different code for each length, which has been optimized for that particular length. All figures of Table 1 are for an IBM-compatible 33 MHz 80386DX based PC with 64K cache memory. The C version was compiled with the WATCOM C/386 9.0 and run with the DOS/4GW DOS extender (i.e., in protected mode). The Assembly language implementation was assembled with Turbo Assembler 2.5 and run in real mode.

	speed in bit/s					
	C			Assembly		
	512	768	1024	512	768	1024
General exponentiation	191	90	52	1045	485	280
P (plain RSA, $2^{16}+1$)	4447	3126	2405	34K	24K	18K
P (Rabin variant)	120K	85K	66K	662K	473K	356K
S (with CRT)	653	322	190	3460	1688	1042

Table 1: Software performance of the public key and secret key operations in plain RSA and its Rabin variant 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.

An interesting alternative to custom hardware is the implementation of RSA on a digital signal processor (DSP) providing hardware speed yet software flexibility. The figures of Table 2 were obtained on a 20 MHz Motorola DSP56001.

	speed in bit/s
	512 bits
General exponentiation	5K
P (plain RSA, $2^{16}+1$)	184K
S (with CRT)	15K

Table 2: Software performance of the public key and secret key operations in plain RSA on a 20 MHz Motorola DSP56001.

6.2 Hardware Implementations

The figures of Table 3 are for a general exponentiation on the fastest RSA chip yet available [Pij92]. It uses a 25 MHz clock frequency. The speed of the S operation can be improved as in the software case, but here the factor of improvement is only about 1.5.

	speed in bit/s		
	512	768	1024
General exponentiation	40K	30K	25K
S (with CRT)	60K	45K	40K

Table 3: Hardware performance of a general modular exponentiation on the PCC200 RSA Encryption Device.

7 Guidelines to Software Implementation

For generating keys for RSA, we refer to Chapter 9. For implementation of the multi-precision arithmetic needed for RSA itself, we refer to Chapter 10. Implementing the modes of use is straightforward, given an implementation of RSA.

References

[FiSh86] A. Fiat and A. Shamir, "How to prove yourself: practical solutions of identification and signature problems," in: *Advances in Cryptology - CRYPTO'86*, A.M. Odlyzko ed., Lecture Notes in Computer Science no. 263, pp. 186-194, 1987.

[GQWL91] L. Guillou, J.-J. Quisquater, M. Walker, P. Landrock and C. Shaer, "Precautions taken against various potential attacks in ISO/IEC 9796," in: *Advances in Cryptology - EUROCRYPT'90*, I.B. Damgård ed., Lecture Notes in Computer Science no. 473, Springer-Verlag, Berlin-Heidelberg-New York, pp. 465-473, 1991.

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

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

[Kob87] N. Koblitz, *A Course in Number Theory and Cryptography*, Springer-Verlag, Berlin-Heidelberg-New York, 1987.

[MiSc91] S. Micali and C.P. Schnorr, "Efficient perfect polynomial random number generators," *Journal of Cryptology*, vol. 3, no. 3, pp.157-172, 1991.

[Mon85] P.L. Montgomery, "Modular multiplication without trial division," *Mathematics of Computation*, vol. 44, pp. 519-521, 1985.

[Pij92] Pijnenburg micro-electronics & software: *PCC200 RSA Encryption Device*, 1992.

[QuCo82] J.-J. Quisquater, C. Couvreur, "Fast decipherment algorithm for RSA public-key cryptosystems," *Electronic Letters*, vol. 18, pp. 905-907, 1982.

[Rab79] M.O. Rabin, *Digital Signatures and Public Key Functions as Intractable as Factoring*, Technical Memo TM-212, Laboratory of Computer Science, Massachusetts Inst. of Technology, 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.

A Test Values for Signature Mode

The following shows some an example of the computation of a signature. The example starts from the hashcode, which has size 128 bits in this example. To facilitate reference to ISO/IEC 9796, we use the terminology from that standard but also refer to the notation from section 2.6.

```
9796 TEST
Data in HEX notation and ordinary number format:
most significant digit first

modulus:
9FFC F9B7 B211 3D86 B214 4A07 6047 D24F EB78 2369 63F6 3653
3D24 4BA1 6AEF 4812 8D3A FE8B 1626 3E62 E02F 1260 1BC5 176F
F0C6 01E4 744D 000E 13C8 0178 46C9 98B1 9F

public exponent:
10001

secret exponent:
232C 329A 3803 A24B 228B 635B 0BC7 BE6C E38F 9DF9 6588 9398
E930 3990 5B3A 0FD6 3EA9 B7DB B667 B8DF 86C8 CA96 3991 2438
B5A0 A1E6 2F89 50ED CFF9 A505 B347 56EB 81

hashcode:
FFEEDDCCBBAA99887766554433221100

Padded and extended message
(only bytes in non-redundant positions shown)
00.. FF.. EE.. DD.. CC.. BB.. AA.. 99.. 88.. 77.. 66.. 55..
44.. 33.. 22.. 11.. 00.. FF.. EE.. DD.. CC.. BB.. AA.. 99..
88.. 77.. 66.. 55.. 44.. 33.. 22.. 11.. 00

Message with redundancy bytes (U)
0011 FFCC EEAA DD77 CC66 BBBB AADD 9900 88FF 7722 6644 5599
4488 3355 2233 11EE 0010 FFCC EEAA DD77 CC66 BBBB AADD 9900
88FF 7722 6644 5599 4488 3355 2233 11EE 00

After truncate and force least sig. byte (I)
4011 FFCC EEAA DD77 CC66 BBBB AADD 9900 88FF 7722 6644 5599
4488 3355 2233 11EE 0010 FFCC EEAA DD77 CC66 BBBB AADD 9900
88FF 7722 6644 5599 4488 3355 2233 11EE 06
```

```
Representative element (RR)
4011 FFCC EEAA DD77 CC66 BBBB AADD 9900 88FF 7722 6644 5599
4488 3355 2233 11EE 0010 FFCC EEAA DD77 CC66 BBBB AADD 9900
88FF 7722 6644 5599 4488 3355 2233 11EE 06

Signature
3F84 1031 166A 4EA9 E956 C80A 6B57 CF9A F3B0 ADC8 AD9B 6E7F
AD93 313C 844D B12F F8C2 26DB 00B4 0F28 A0DA 7E9B B559 40EB
8FCA 25BA 7C20 2804 1521 8605 59D0 3FC8 15
```

Chapter 8

COMSET

Contents

1 Introduction

COMSET is a cryptographic protocol that allows any two users to identify themselves to each other, and also to exchange a secret key that they use for subsequent data origin authentication. Such interaction will typically occur at the beginning of a session, and this led to the acronym

COMSET for COMmunication SETup.

The essential idea of the protocol is that in order to identify a user in a system with public key cryptography, it is sufficient to establish authenticity of his public key and subsequently to be convinced that the user is in possession of the corresponding secret key. The first task can be solved by a public key directory, or by certificates (cf. [CCI89]); this is not part of COMSET. COMSET concentrates on the second task: how can $\mathcal{A}$ (Alice) convince $\mathcal{B}$ (Bob) that she possesses the corresponding secret key, if $\mathcal{B}$ knows the public key of $\mathcal{A}$? Further, COMSET provides the two users with a secret key know only to them. In many situations this is desirable for subsequent data origin authentication using a symmetric cryptosystem. These concepts are explained in Part II of this report.

The underlying mathematical principle of COMSET is the Rabin variant (cf. [Rab79]) of the RSA scheme (cf. [RSA78]), which is also a primitive described in Chapter 7 of this report.

Originally COMSET was suggested by J. Brandt, I. Damgård, P. Landrock and T. Pedersen in [BDLP88].

The structure of this description of COMSET is as follows. In order to avoid any ambiguities in the description, the notation and definitions used are fixed in section 2. In section 3 the primitive is described and in section 4 its use is explained. In section 5 security aspects of the primitive are discussed. This includes claimed properties and the results of the algebraic evaluation of the primitive. Finally, in section 6 the performance of the protocol is considered. Section 7 refers the reader to the Appendix B "Implementation Guidelines for Arithmetic Computations".

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 as well as 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.

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

A sequence of n bits $b_0, b_1, \ldots, b_{n-1}$ is interpreted as an nonnegative integer B in the following way. The bits are considered as the binary representation of B, the more significant bits being first in the sequence. That is,

$$B := \sum_{i=0}^{n-1} b_i 2^{n-i-1}.$$

Conversly, an interpretation of a number as a sequence of bits is defined by this equation, if a bit length for this number is fixed. In this chapter only the bit length L of the modulus is used.

2.4 Definitions and Basic Operations

- A string is a sequence of bits.
- For a bit 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.
- For an nonnegative 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 string $X = x_0, x_1, \ldots, x_{n-1}$, the string $x_i, x_{i+1}, \ldots, x_j$ with $0 \le i \le j < |X|$, is denoted as $X_{i,j}$. This notation is also used for nonnegative integers. In this case its binary representation as an L-bit number is used to take the substring from.
- 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

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

  That is, A mod B is the *remainder*, and A div B is the *quotient* of an integer division of A by B.
- The notation "$X \equiv Y \pmod{N}$" (X is equivalent to Y modulo N) is used to indicate that $X \bmod N = Y \bmod N$.

- For two nonzero integers X and Y we say that X *divides* Y if $Y \bmod X = 0$. That is, if Y is a multiple of X.
- For two nonnegative integers X and Y, not both zero, the *greatest common divisor* $\gcd(X, Y)$ is defined as the greatest positive integer that divides both X and Y.
- An integer X is *invertible modulo* N if $\gcd(X, N) = 1$, see [Kob87].
- An integer X that is invertible modulo N is said to be a *quadratic residue modulo* N if there is an integer Y such that $X \equiv Y^2 \pmod{N}$.
- A *prime* is an integer greater than 1 that is divisible only by 1 and by itself.

2.5 Functions and Symbols Used by the Primitive

- The users of the protocol are denoted by $\mathcal{A}$ (for Alice) and $\mathcal{B}$ (Bob).
- $p_{\mathcal{A}}$ and $q_{\mathcal{B}}$ are the secret prime factors of the modulus $n_{\mathcal{A}}$ of $\mathcal{A}$.
- L is the bit length of L.
- The public encryption function of $\mathcal{A}$ is denoted by $P_{\mathcal{A}}()$. It is defined by $P_{\mathcal{A}}(m) = m^2 \bmod n_{\mathcal{A}}$
- The secret decryption function of $\mathcal{A}$ is denoted by $S_{\mathcal{A}}()$. It is defined by $S_{\mathcal{A}}(c) = c^{d_{\mathcal{A}}} \bmod n_{\mathcal{A}}$, where
$$d_{\mathcal{A}} = \frac{(p_{\mathcal{A}} - 1)(q_{\mathcal{A}} - 1) + 4}{8}.$$
- m denotes the message in the protocol.
- c denotes the challenge in the protocol.
- v denotes the challenge validator in the protocol.
- The secret key exchanged in the protocol is denoted by k.

3 Description of the Primitive

COMSET is a public key identification protocol. It provides entity authentication of its user $\mathcal{A}$. It also provides exchange of secret keys.

Each potential participant $\mathcal{A}$ of the protocol must have secretly selected two primes $p_{\mathcal{A}}$ and $q_{\mathcal{A}}$ of $L/2$ bits each with $p_{\mathcal{A}} \equiv 3 \pmod{8}$ and $q_{\mathcal{A}} \equiv 7 \pmod{8}$. L is a security parameter whose meaning is discussed in Appendix A of this report, it must be at least 512. Note that for security reasons $p_{\mathcal{A}}$ and $q_{\mathcal{A}}$ have to meet additional conditions which are also described in Appendix A.

$\mathcal{A}$ publishes the product $n_{\mathcal{A}} = p_{\mathcal{A}} q_{\mathcal{A}}$ as her public key $\mathcal{P}_{\mathcal{A}}$. She must keep secret the prime factors $p_{\mathcal{A}}$ and $q_{\mathcal{A}}$.

The public encryption function of $\mathcal{A}$ is given by $P_{\mathcal{A}}(m) = m^2 \bmod n_{\mathcal{A}}$. It is only used for quadratic residues m modulo $n_{\mathcal{A}}$.

Only $\mathcal{A}$ knows her secret decryption function $S_{\mathcal{A}}(c) = c^{d_{\mathcal{A}}} \bmod n_{\mathcal{A}}$, where

$$d_{\mathcal{A}} = \frac{(p_{\mathcal{A}} - 1)(q_{\mathcal{A}} - 1) + 4}{8}.$$

An explanation of why this Rabin variant (cf. [Rab79]) of the RSA scheme (cf. [RSA78] and Chapter 7 of this report) works may be found in [Kob87].

The COMSET protocol for participants $\mathcal{A}$ and $\mathcal{B}$ is as follows.

- $\mathcal{B}$ chooses a random integer x with $2 \leq x < n_{\mathcal{A}}$. Each number in this interval should be selected with equal probability. The message m is determined by the equation $m := x^2 \bmod n_{\mathcal{A}}$. He encrypts m to the *challenge* $c := P_{\mathcal{A}}(m)$ using the public key $\mathcal{P}_{\mathcal{A}}$ of $\mathcal{A}$. He sends the challenge c together with a *challenge validator* $v := m_{L-64,L-1}$ i. e., the 64 least significant bits of m to $\mathcal{A}$.

- $\mathcal{A}$ decrypts c into $m' := S_{\mathcal{A}}(c)$ and compares $m'_{L-64,L-1}$ with v: if equality holds, she sends the *answer* $w := m'_{L-128,L-65}$ to $\mathcal{B}$. If $m'_{L-64,L-1}$ and v are not equal, she compares $(n_{\mathcal{A}} - m')_{L-64,L-1}$ with v. If equality holds, she sends the *answer* $w := (n_{\mathcal{A}} - m')_{L-128,L-65}$ to $\mathcal{B}$. If this check also fails, $\mathcal{A}$ sends the message "stop" to $\mathcal{B}$, the execution of the protocol failed.

- $\mathcal{B}$ compares w with $m_{L-128,L-65}$; if equality holds, the authentication has been successful.

- Finally, both $\mathcal{A}$ and $\mathcal{B}$ determine their shared secret key. Let l be the bit length of the key to use later for symmetrical message authentication. $\mathcal{B}$ takes as secret key $k := m_{L-128-l,L-129}$. $\mathcal{A}$ finds the secret key by using the equation $k = m'_{L-128-l,L-129}$, if $m'_{L-64,L-1} = v$ holds, and $k = (n_{\mathcal{A}} - m')_{L-128-l,L-129}$ if $(n_{\mathcal{A}} - m')_{L-64,L-1} = v$.

The description can be summarized in the following bird's eye view of the protocol.

$\mathcal{A}$		$\mathcal{B}$
		choose random square m compute $c := P_{\mathcal{A}}(m)$ define $v := m_{L-64,L-1}$
	$\xleftarrow{c,v}$	

compute $m' := S_{\mathcal{A}}(c)$
if $v = m'_{L-64,L-1}$
then define $w := m'_{L-128,L-65}$
else if $(n_{\mathcal{A}} - m')_{L-64,L-1} = v$
then define $w := (n_{\mathcal{A}} - m')_{L-128,L-65}$
else failure
	$\xrightarrow{w}$	

		verify that $w = m_{L-128,L-65}$

if $v = m'_{L-64,L-1}$
then compute $k := m'_{L-128-l,L-129}$
if $(n_{\mathcal{A}} - m')_{L-64,L-1} = v$
then compute $k := (n_{\mathcal{A}} - m')_{L-128-l,L-129}$
		compute $k := m_{L-128-l,L-129}$

Figure 1. The COMSET protocol

4 Use of the Primitive

COMSET provides entity authentication of $\mathcal{A}$, where the entity $\mathcal{A}$ is identified by her knowledge of the factorization of an L-bit public number, combined with the exchange of a secret key that can be used for subsequent message authentication. This means that $\mathcal{B}$ has good reason to believe that he is communicating with $\mathcal{A}$, and that the secret key exchanged is only known to him and $\mathcal{A}$.

Note that $\mathcal{A}$ has no reason to believe that she is communicating with $\mathcal{B}$, or that the one she shares the secret key with is indeed $\mathcal{B}$. If this is desired as well, the protocol has also to be executed with the roles of $\mathcal{A}$ and $\mathcal{B}$ interchanged. Both authentication processes can be executed at essentially the same time. This means that while $\mathcal{B}$ is computing in the first step of the protocol message and challenge validator, $\mathcal{A}$ executes the first step of the protocol as well (of course she uses the public key of $\mathcal{B}$ to determine the challenge validator). Then $\mathcal{A}$ and $\mathcal{B}$ exchange their c and v values. After that both can execute the second step of the protocol at essentially the same time, and so on. At the end, each user has two keys. The bitwise exclusive or of these is used as the common secret key of $\mathcal{A}$ and $\mathcal{B}$.

Note that $\mathcal{B}$ must establish the authenticity of the public key of $\mathcal{A}$. This is not achieved by the COMSET protocol, but can be realized using a public key directory or certificates (cf. [CCI89]).

5 Security

5.1 Claimed Properties

The COMSET protocol is claimed to satisfy the following:

- If both parties follow the protocol, all checks will be satisfied, and the protocol will complete successfully.
- It is infeasible for any user $\mathcal{X}$ to claim he is $\mathcal{A}$ and on his own complete the protocol successfully with $\mathcal{B}$, unless $\mathcal{X}$ possesses the secret key of $\mathcal{A}$.
- COMSET is a zero-knowledge protocol (see [GMR89] for a formal definition), i.e., after completing the protocol with $\mathcal{A}$, $\mathcal{B}$ has obtained no information that he could not have computed efficiently by himself. In particular, he has learnt nothing that could help him to impersonate $\mathcal{A}$ later.
- If $\mathcal{A}$ and $\mathcal{B}$ complete the protocol successfully, $\mathcal{B}$ can assume that the key k has been received by $\mathcal{A}$. Moreover, it is infeasible for any third party observing the communication to guess the value of k essentially better than at random.

Here, and in what follows, by "infeasible" we mean computationally far out of reach of current technology.

5.2 Algebraic Evaluation

The properties claimed rely on the following two assumptions:

1. Given only $m^2 \bmod n$ and n, the least significant half of the bits of m are simultaneously secure, i.e., even if some of the bits are revealed, it is infeasible to guess any of the remaining bits essentially better than at random. In particular, this means that from only n and (c, v) it is infeasible to guess m or w, and also that from n, (c, v) and w it is infeasible to guess any bit of k.
2. Given only n, it is infeasible to compute a pair (c, v) that will pass $\mathcal{A}$'s check without also being able to compute m; in other words it is infeasible for $\mathcal{B}$ to come up with a quadratic residue and many bits of a square root without in fact being able to compute the entire square root.

The first assumption made is a special case of the conjecture made by Micali and Schnorr in [MiSc91], this conjecture is in turn based on the assumption that modular squaring is a one-way function, in other words that RSA and Rabin's variant of RSA are secure. Therefore the first assumption is reasonable.

The second assumption is also supported in part by [MiSc91]: their conjecture says that the least significant bits of a modular square root are simultaneously secure (see explanation above). This implies that the most obvious way of trying to break the assumption, namely by selecting a random number and trying to compute the least significant bits of a square root, will not be feasible. It is therefore conjectured that essentially the only feasible way of computing a valid pair (c, v) is by first selecting m and then computing the pair, and this is just the content of the second assumption.

Assuming 1. and 2., the only remaining attack to consider is an intermediate person, who simply forwards the different parts of the protocol, acting as a relay so to speak. However, this attacker would never be able to learn the common key, as this requires knowledge of the secret key. In fact this type of attack, known as the mafia attack, will only work in one of the following instances in general, completely independent of the actual schemes:

1. Key exchange without authentication.
2. An identification scheme with no connection to the subsequent communication (access control).

6 Performance Evaluation

In the protocol $\mathcal{B}$ has to perform only two modular squarings, i.e., one squaring to choose m and one squaring to compute c. This is negligible compared to the general modular exponentiation $\mathcal{A}$ has to perform: the length of the exponent e is about the same as that of the modulus n. Therefore, the performance of COMSET is completely determined by the time needed to perform *one* general modular exponentiation. This still holds if both $\mathcal{A}$ and $\mathcal{B}$ have to prove themselves to each other, because in that case the two general exponentiations can be done simultaneously.

6.1 Software Implementations

In practical systems both primes of the modulus are either 256, 384 or 512 bits long, resulting in, respectively a 512, 768 or 1024-bit modulus. The basic multiprecision operations needed to implement such a modular exponentiation are the multiplication of two integers, the squaring of an integer and the modular reduction of a integer (see Appendix B for an algorithm). For the multiplication and squaring optimized versions of the classical algorithms are used, as described in [Knu81]. The modular reduction is implemented according a method due to P.L. Montgomery, allowing a reduction in almost the same time as a multiplication [Mon85].

Using knowledge of the prime factors of the modulus, the $S_{\mathcal{A}}$ operation can be speeded up with a factor 3 to 4. That is, using the so-called Chinese Remainder Theorem (CRT) the $S_{\mathcal{A}}$ operation can be reduced to basically two general exponentiations modulo the two prime factors, being only half the length of the modulus, see [QuCo82].

Both a C and a 80386 Assembly language implementation are considered. The C version has the advantage of being portable. The considerable difference in performance between the C and the Assembly language version is due to the general nature of the C code: it can be used for integers of arbitrary length, whereas the Assembly versions use different code for each length, which has been optimized for that particular length.

All figures of Table 1 are for an IBM-compatible 33 MHz 80386DX based PC with 64K cache memory. The C version was compiled with the WATCOM C/386 9.0 and run with the DOS/4GW DOS extender (i.e., in protected mode). The Assembly language implementation was assembled with Turbo Assembler 2.5 and run in real mode.

An interesting alternative to custom hardware is the implementation of a modular exponentiation on a digital signal processor (DSP) providing hardware speed yet software flexibility. The figures of Table 2 were obtained on a 20 MHz Motorola DSP56001.

	speed in bit/s					
	C			Assembly		
	512	768	1024	512	768	1024
General exponentiation	191	90	52	1045	485	280
$S_{\mathcal{A}}$ (with CRT)	653	322	190	3460	1688	1042

Table 1: Software performance of the secret key operation $S_{\mathcal{A}}$ 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.

	speed in bit/s
	512 bits
General exponentiation	5K
$S_{\mathcal{A}}$ (with CRT)	15K

Table 2: Software performance of the secret key operation $S_{\mathcal{A}}$ on a 20 MHz Motorola DSP56001.

6.2 Hardware Implementations

The figures of Table 3 are for a general exponentiation on the fastest RSA chip yet available [Pij92]. It uses a 25 MHz clock frequency. The speed of the $S_{\mathcal{A}}$ operation can be improved as in the software case, but here the factor of improvement is only about 1.5.

7 Guidelines for Implementation

The implementation of the protocol is straightforward. For multiprecision arithmetic which is needed for the computations required by the protocol we refer to Appendix B.

	speed in bit/s		
	512	768	1024
General exponentiation	40K	30K	25K
$S_{\mathcal{A}}$ (with CRT)	60K	45K	40K

Table 3: Hardware performance of a general modular exponentiation on the PCC200 RSA Encryption Device.

References

[BDLP88] J. Brandt, I.B. Damgård, P. Landrock and T. Pedersen, "Zero-knowledge authentication scheme with secret key exchange," in: *Advances in Cryptology - CRYPTO'88*, S. Goldwasser ed., Lecture Notes in Computer Science no. 403, Springer-Verlag, Berlin-Heidelberg-New York, pp. 583-588, 1990.

[CCI89] CCITT Recommendation X.509, *The Directory-Authentication Framework*, 1989.

[GMR89] S. Goldwasser, S. Micali and C. Rackoff, "The knowledge complexity of interactive proof systems," *SIAM Journal on Computing*, vol. 18, no. 1, pp. 186-208, 1989.

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

[Kob87] N. Koblitz, *A Course in Number Theory and Cryptography*, Springer-Verlag, Berlin-Heidelberg-New York, 1987.

[MiSc91] S. Micali and C.P. Schnorr, "Efficient perfect polynomial random number generators," *Journal of Cryptology*, vol. 3, no. 3, pp. 157-172, 1991.

[Mon85] P.L. Montgomery, "Modular multiplication without trial division," *Mathematics of Computation*, vol. 44, pp. 519-521, 1985.

[Pij92] Pijnenburg micro-electronics & software: *PCC200 RSA Encryption Device*, 1992.

[QuCo82] J.-J. Quisquater, C. Couvreur, "Fast decipherment algorithm for RSA public-key cryptosystems," *Electronic Letters*, vol. 18, pp. 905-907, 1982.

[Rab79] M.O. Rabin, *Digital Signatures and Public Key Functions as Intractable as Factoring*, Technical Memo TM-212, Laboratory of Computer Science, Massachusetts Institute of Technology, 1979.

[RSA78] R. 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.

Chapter 9

RSA Key Generation

Contents

1 Introduction

This appendix gives an overview of state-of-the-art of key generation for the RSA public key cryptosystem (see Chapter 7). It contains a minimum of theoretical background, and a maximum of practically oriented information.

The appendix treats RSA key generation by a top-down approach, and ends with a conclusion on which methods are preferable in various situations.

All the definitions and notation from the RSA-chapter 7 of this report are also used in this appendix. Let us recall the most important symbols: RSA uses the following key data:

Public Key: Modulus n, Public exponent e.

Secret Key: Prime numbers p, q such that $pq = n$. Secret exponent d, such that $ed \equiv 1 \bmod (p-1)(q-1)$ [1].

The public key operation P is defined by $P(m) = m^e \bmod n$ for any $0 \leq m < n$, while the secret key operation S is defined by $S(c) = c^d \bmod n$ for any $0 \leq c < n$. The choice of e, d ensures that P and S are the inverses of each other.

The primes are not really needed to execute the S-operation, except for some optimized versions of it. Nevertheless, they need to be computed during the key generation phase, in order to correctly generate the rest of the key data.

The Chapter is structured as follows: Section 2 describes a generic algorithm for key generation, and Section 3 discusses the security constraints needed to ensure that the RSA key is sufficiently protected against the known attacks. Section 4 then explains different ways of generating primes for use in RSA, probable as well as provable. Section 5 briefly mentions the problem of randomness, and finally Section 6 discusses the cryptographic aspects of the choice of algorithm.

2 High Level Algorithm

This section describes a generic algorithm *keygen* for RSA key generation. We will assume that it is given a public exponent e as input, such that the goal is to generate the rest of the key material to fit e. This is because some applications of RSA need the public exponent to be a constant which is the same for all users. Moreover, one may wish to choose a small public exponent since this makes the P-operation more efficient. We will also assume that *keygen* is given an exact bit length k for the modulus produced, and random seeds s_p, s_q from which the two primes are to be generated.

keygen assumes the existence of two procedures:

- *generateprime*(s, I, e), which returns a random prime p in the interval I, generated from random seed (bit string) s, such that e and $p-1$ have greatest common

[1] The least common multiple of $(p-1)$ and $(q-1)$ could be used instead and results in slightly shorter exponents

divisor 1. The last condition is necessary to ensure that we can find a suitable secret exponent. Section 4.1 shows one concrete implementation, *strongprime* and some simpler methods *probprime* and *probprimeinc.* Another alternative is the procedure *hybridprime* from Section 4.3 or methods derived from Section 4.2. Section 6 gives a discussion on the choice of method.

- *inverse*(a, m), which returns a number b, such that $ab \equiv 1 \bmod m$. A standard algorithm for implementing it can be found in any textbook on numerical algorithms (e.g., [Knu81]).

The input parameters to *keygen* leave open a choice for the interval in which to generate the primes. The only constraint is that they must multiply together to a number of the right length. For security reasons (see Section 3.1.1 for details) the primes should be of approximately the same bit length. We take a simple approach, and let the lengths of the primes be as close as possible, and ensure simultaneously that the modulus will have exact bit length k.

An example: suppose $k = 512$. Then we will let each prime be of length 256, and choose the interval such that both p and q have the two most significant bits set, i.e., p, q are in the interval $]2^{254}+2^{255} \ldots 2^{256}[$. This will ensure that $n = pq$ is in the interval $]2^{511} \ldots 2^{512}[$, i.e., has bit length exactly 512. Generalizing this, we have the following notation:

$$I_p(k) = \begin{cases}]2^{k/2-1} + 2^{k/2-2} \ldots 2^{k/2}[& \text{if } k \text{ is even} \\ ]2^{(k+1)/2-1} + 2^{(k+1)/2-2} \ldots 2^{(k+1)/2}[& \text{otherwise} \end{cases}$$

$$I_q(k) = \begin{cases}]2^{k/2-1} + 2^{k/2-2} \ldots 2^{k/2}[& \text{if } k \text{ is even} \\ ]2^{(k-1)/2-1} + 2^{(k-1)/2-2} \ldots 2^{(k-1)/2}[& \text{otherwise} \end{cases}$$

With this in mind, we can describe the algorithm itself. Below follow subsections explaining how to choose the input parameters:

```
PROCEDURE KEYGEN(seed $s_p, s_q$, integer $k$, exponent $e$ )

output: RSA key set (with public exponent e) consisting of
    $n$ ($k$ bits),
    $p, q$ (about $k/2$ bits each),
    $d$ ($k$ bits)

  1. $p :=$ generateprime$(s_p, I_p(k), e)$
  2. $q :=$ generateprime$(s_q, I_q(k), e)$
  3. $d :=$ inverse$(e, (p-1)(q-1))$
  4. $n := pq$
  5. return $n, p, q, d$
```

2.1 Management of Secret Data

Although physical security is not a main subject of this appendix, we point out that some of the data handled by *keygen* are of course to be kept secret, and should be treated accordingly by the implementation.

It is important to realize that this is true, not only for the secret key, i.e., p, q, d, but also for the random seeds s_p, s_q: with knowledge of the seeds, the entire RSA key can be reconstructed. All the numbers s_p, s_q, p, q, d should therefore be treated at the same level of security.

2.2 Choice of Public Exponent e

The definition of the secret key implies that e must be an odd number (there is a variant of RSA that uses $e = 2$, the so called Rabin system — see Section 2.5 for a description of the changes needed in the key generation to support this system).

Hence *keygen* should be called with an odd number e as input. It should have at most k bits, but as mentioned above, a small e gives a much more efficient P-operation (but does not affect the time for the S-operation). In general, the time to compute $m^e \bmod n$ is directly proportional to the bit length of e.

There is nothing known to suggest that there is any difference in security between a small e and a randomly chosen one, although it should be mentioned that when using RSA for secrecy with very small values of e, one should not send messages that are numerically extremely small, or send exactly the same message to many different users. No such problems occur with the modes of use suggested in this report.

The smallest possible value of e is of course 3. Another popular value is $e = 2^{16} + 1$. It has only two 1's in its binary representation which makes the square-and-multiply algorithm very efficient for this exponent compared to other 17 bit numbers (see Appendix B).

2.3 Choice of the Bitlength k

The parameter k controls the size of the modulus generated. To attack the key generated, one may try to find p, q from n, i.e., factor the modulus. It is necessary that k is large enough to make this a difficult task. State of the art suggests that $k = 512$ is the absolutely smallest value one should consider. At the other extreme, all experts agree that it is not even remotely possible to factor a 1024-bit number in any forseeable future. Already 600 bits is way out of reach currently. See Section 3.1.1 for details.

There is a trade-off here between security and performance. In practice, doubling the length of the modulus will make the P and S operations 3-4 times slower in software, and about 2 times slower in hardware.

2.4 Choice of the Seeds s_p, s_q

As mentioned above, the random seeds must be treated as secret data. This also means that they must be hard to predict for an outsider. Therefore one should not rely only on approaches that are good enough in other contexts, like e.g., taking the system time, or using the standard random number generator of the programming language used. It is advisable to take at least some random input from the user, and of course a hardware source of really random bits is preferable, if available.

To prevent an outsider from simply guessing the seeds, they should be at least 64 bits each, but $k/2$ bits each are required to ensure maximal diversity of the primes generated.

2.5 Rabin's RSA Variant

This system works essentially like RSA, except that we use $e = 2$. This means that the E operation maps 4 different inputs to one output, while the D operation reconstructs exactly 1 of these inputs. The problems caused by this can be solved in various ways, and we will not give any details here, but refer to for instance [Kob87] for more details.

The most commonly used version of the Rabin system can be described as follows:

Public Key: Modulus n.

Secret Key: Prime numbers p, q such that $pq = n$ and p is congruent to 3 modulo 8, and q is congruent to 7 modulo 8. Secret exponent $d = ((p-1)(q-1)+4)/8$.

The procedures described in the sequel can be made to generate primes suitable for this system, by calling *generateprime* with $e = 1$, and inside this procedure, whenever a number is considered as a candidate prime, discard it immediately, if it is not congruent to 3 modulo 8, respectively 7 modulo 8.

3 Security Constraints

Before we go into the actual algorithms for generating primes, we have to describe some constraints that are necessary in order to ensure that the RSA key we generate will in fact be hard to break with the known algorithms. It should be noted that the example parameters given represent acceptable security today, but that higher values might be needed in the future.

3.1 Precautions against Factoring Algorithms

It is clear that an enemy should not be able to find the prime factors p, q from the RSA modulus n, i.e., it should be hard to factor n. There are a large number of factoring algorithms known, out of which some are particularly efficient against numbers of various special forms. We should make sure that the numbers we generate are not of this kind. The subsections below list the relevant factoring algorithms, and the constraints they imply.

3.1.1 General Factoring Algorithms

These are algorithms that can be used against numbers that do not have properties to make them vulnerable to special methods (see below). The best algorithm of this type is currently the quadratic sieve algorithm. It may eventually be outperformed by a variant known as the number field sieve [LLMP90]. The largest numbers without special properties that can be factored using this type of algorithm are of length about 110-120 decimal digits. The factorization can be found in about 60 days real time, using a large number of computers in parallel [LeMa90].

It is generally accepted that the hardest input for such algorithms are randomly chosen numbers with two prime factors of approximately the same size. This is the motivation for letting p and q have the same bit length. It should be mentioned that if $p - q$ is relatively small (less than 2^{75}, say), there is an easy and elementary way of factoring $n = pq$. However, for all the methods described below for prime number generation, the diversity of the primes generated is large enough to make the probability of having such a small $p - q$ completely negligible.

3.1.2 The $p - 1$ and the Elliptic Curve Method

The $p - 1$ method is a factoring method suggested by Pollard, which will work if $p - 1$ has only small prime factors (which is called a"smooth" number), where p is some prime factor in n. Lenstra found a generalization of this method that will work, if one can find a so called elliptic curve over p, whose order is a smooth number. The orders that are possible can be expected to be in the interval $[p - \sqrt{p} \ldots p + \sqrt{p}]$.

Although it is completely infeasible to check all possible orders of elliptic curves for smoothness, it may still be a good idea to pay special attention to smoothness of $p - 1$ since if $p - 1$ is indeed smooth, Pollards algorithm will be much more efficient than the elliptic curve method, even if we get a smooth order curve for free.

Thus, we should ensure that the p, q generated for RSA are such that none of $p-1$, $q - 1$ are smooth. More concretely, with the current state of the art, this means that $p - 1$ and $q - 1$ should have at least one prime factor of at least 75 bits. If we look for large enough numbers (about 350-400 bits), a randomly chosen prime will satisfy the condition with very large probability. For smaller numbers, however, a special algorithm is needed.

3.1.3 The Cyclotomic Polynomial Method

This is a factoring method suggested by Bach and Shallit [BaSh89]. It will be efficient if n has a prime factor p, such that a particular function of p produces a smooth number. There are several choices for this function. One possibility is $p + 1$, others are $p - 1$ and $p^2 + p + 1$, and there are many others involving p^2 or larger powers of p.

Out of these possibilities, $p - 1$ has already been considered, and those involving p^2 or larger powers can be neglected, because the function values will be much larger than p (at least 512 bits) and will therefore have negligible probability of being smooth.

As far as $p+1$ is concerned, it can be as small as 256 bits in practice, and a random number this size has a small, but non-negligible probability of being smooth. Therefore the RSA key generation should make sure that $p+1$ and $q+1$ are not smooth, at least if the modulus is less than 600-700 bits.

3.2 Precautions against Iterated Encryption

One potential way of breaking RSA without factoring is by repeated encryption, i.e., given ciphertext C, one encrypts C m times to get $P(P(\cdots P(C)\cdots)) = P^m(C)$, for increasing values of m, until we get to a point where for some m, $C = P^m(C)$. Then the corresponding plaintext will in fact be $P^{m-1}(C)$.

For any instance of the RSA system, there is an m with this property. However, it will almost always be the case that m is so large that the attack is infeasible.

If one knows a relatively large prime factor t_p of $r_p - 1$, where r_p is a prime that divides $p-1$ and similarly primes t_q, r_q for q, it is possible to check that the value of m is not too small: if the public exponent e satisfies that $e^{(r_p-1)/t_p} \neq 1 \bmod r_p$ and $e^{(r_q-1)/t_q} \neq 1 \bmod r_q$, then m is divisible by both t_p and t_q, so it is at least $t_p t_q$.

3.3 Summary of Constraints

In summary, we have the following demands to RSA key material of good quality:

1. p, q should be of approximately the same bit length, but $p - q$ must not be less than 2^{75}.

2. $p-1, q-1, p+1, q+1$ should have prime factors respectively r_p, r_q, s_p, s_q, all of which should be of length at least 75 bits.

3. The multiplicative order of e modulo $(p-1)(q-1)$ must be large. This is satisfied if $r_p - 1$ and $r_q - 1$ have prime factors t_p, resp. t_q such that $e^{(r_p-1)/t_p} \neq 1 \bmod r_p$ and $e^{(r_q-1)/t_q} \neq 1 \bmod r_q$; and such that $t_p t_q$ is of length at least 75 bits.

Of these constraints, the first part of 1 has been taken care of by the construction of *keygen* above. The second part is satisfied with overwhelming probability, if one uses the methods described below for generating the primes. Of course, one may also check it directly, if absolute certainty is desired.

Conditions 2 and 3 may be taken care of by integrating them in the method for generating p and q. Details will be given below.

4 Generation of Primes

This section explains in detail ways of generating primes for use in RSA-keys. There are several possibilities: One can use numbers generated such that it can only be asserted with some (large) probability that they are primes, one can use numbers generated in such a way that it can be proven that they are primes, or one can combine the two methods in various ways. The following subsections contain technical considerations dealing with these aspects. For a conclusion see Section 6.

4.1 Probable Primes

Probable primes are numbers generated such that we can only assert with some (large) probability that they are primes. Later we will look at other methods that provide absolute certainty, and produce so called provable primes. In practice, however, probable primes are sometimes preferable because they often can be generated more efficiently, and nearly always lead to smaller program sizes than provable primes.

4.1.1 The Rabin Test

The Rabin test is a procedure that is called with an integer n as input. It will test whether n is a prime and will accordingly return "fail" or "pass" as output. As we shall see, the answer is not always correct, but we can gain larger certainty by repeating the test.

rabintest works as follows:

```
PROCEDURE RABINTEST(integer $n$)

Output: "fail" or "pass"

  1. define $h, a$ by: $n - 1 = a2^h$, and $a$ is odd.
  2. choose $b$ uniformly at random from the interval $]1 \ldots n-1[$
  3. $b := b^a \bmod n$
  4. if $b = 1$, return(pass)
  5. if there is an $i$, such that $0 <= i < h$ and $b^{2^i} \bmod n = n - 1$ then
     return(pass)
     else
     return(fail)
```

The number b is called the "base". Note that the numbers $b^{2^i} \bmod n$ can be computed easily by repeated squaring of b.

The basic facts about this test are:

- If n is a prime, then *rabintest*(n) = *"pass"* always.
- If n is a composite, then *rabintest*(n) = *"pass"* with probability at most 1/4.

Note that these basic facts do not necessarily imply that a method for prime number generation using t iterations of the test will have an error probability less than $(1/4)^t$. The error probability depends on the distribution with which candidate primes are chosen. More details are given below.

$k\backslash t$	1	2	3	4	5	6	7	8	9	10
100	1	9	15	21	26	30	33	36	39	41
150	2	12	21	28	34	39	43	47	51	54
200	3	15	26	34	41	47	52	57	61	65
250	8	19	30	40	47	54	60	65	70	75
300	16	26	35	45	53	60	67	73	78	84
400	35	42	50	58	66	73	80	87	93	99
500	53	60	67	74	81	88	94	101	107	114
600	72	79	85	92	98	104	110	116	122	128

Figure 1: Bounds on error probability of *probprime*, with $e = v = 1$.

4.1.2 Uniform Probable Primes

We describe here one possible procedure, *probprime*, for generating a random prime number in an interval. To facilitate the description of procedures later in this report, we also give *probprime* the ability to ensure that the prime p generated satisfies that some given integer v divides $p-1$ and that p can be used with a given public exponent e, i.e., $gcd(p-1, e) = 1$. A totally random prime is obtained by setting $e = v = 1$.

It assumes the existence of the procedure *randomchoice*, which is called with an interval I as input and returns a random odd number chosen from I. We will return later to how this procedure will be realized in practice (see Section 5).

Also, a table is assumed that contains all odd primes less than some upper limit r, where r is a constant chosen once and for all. Since dividing a candidate number by a small prime is much faster than doing a Rabin test, the overall time to find a prime will be smaller, if we subject candidates to the Rabin test only if they are not divisible by the small primes in the table. Maurer [Mau89] has shown that the optimal value of r is R/D, where R is the time needed to do one Rabin test, and D is the time needed to divide a candidate prime number by one prime less than r.

Finally, we assume a function *gcd* for computing the greatest common divisor of two integers. The algorithm can be found in e.g., [Knu81] and in Appendix A.

If I is the interval $[a \dots b]$, we let cI denote the interval $[ca \dots cb]$.

The procedure guarantees that if *randomchoice* returns uniformly distributed numbers, and $v = e = 1$ then the primes produced by *probprime* will be uniformly distributed in the interval specified as input.

PROCEDURE PROBPRIME(interval I, divisor v, exponent e)

Output: a probable prime number p chosen at random from I, such that $p-1$ is divisible by v and $gcd(p-1, e) = 1$.

```
1. $n := 2v \cdot randomchoice(1/(2v)I) + 1$

2. if $n$ is divisible by a prime less than $r$, or $gcd(n-1, e) \neq 1$, go to 1.
```

$k\backslash t$	1	2	3	4	5	6	7	8	9	10
100	0	2	9	15	19	23	27	30	33	36
150	0	5	14	21	27	32	36	40	44	48
200	0	8	18	27	34	39	45	49	54	58
250	0	11	22	32	40	46	52	58	62	67
300	0	13	26	36	45	52	59	65	70	75
400	1	18	33	45	55	63	71	78	85	91
500	2	22	39	52	63	73	82	90	97	104
600	4	25	44	59	71	82	92	101	109	116

Figure 2: Bounds on error probability of *probprimeinc*, when $e = v = 1$.

```
3. for $count = 1 \ldots t$ do: if $rabintest(n) = $ "fail" go to 1
4. return($n$)
```

In [DLP], the probability that this procedure outputs a composite number is analyzed, in the cases where the interval specified is of the form $[2^{k-1} \ldots 2^k]$ for some k, and $e = v = 1$. Upper bounds for the probability are given in table 1. The numbers given are $-\log_2$ of the bound for the probability. For example, if we look for 250-bit primes and use $t = 6$, the error probability is less than 2^{-54}.

4.1.3 Incremental Search

An alternative to *probprime*, which is more economical in its use of random bits, is to choose at random only some odd starting point n_0, and then do an incremental search for the smallest prime larger than n_0, i.e. we look at $n_0, n_0 + 2, \ldots$.

If we want the same enhancement as for *probprime*, i.e., ensuring that the result minus 1 is divisible by a given v, we make sure that $n_0 - 1$ is divisible by v, and examine $n_0, n_0 + 2v, \ldots$.

One advantage of this approach is that the testdivision by small primes can be done much more efficiently: first compute the residue of n_0 modulo each small prime in the table. Each time we add $2v$ to the current current candidate, add $2v$ to each residue modulo the small primes, and test that no residue becomes 0. In [BDL91] it is shown that the optimal limit r for the small primes in this case is $r = \frac{R}{D \cdot log(R/D)}$.

We have the following implementation of this idea:

PROCEDURE PROBPRIMEINC(interval I, divisor v, exponent e)

Output: a probable prime number p chosen at random from I by incremental search, such that v divides $p - 1$ and $gcd(p - 1, e) = 1$.

```
1. $n = 2v \cdot randomchoice(1/(2v)I) + 1$, initialize testdivision.
```

2. $n = n + 2v$, if n is now not in I, go to 1.
3. if n is divisible by a prime less than r (use optimized test division), or $gcd(n-1, e) \neq 1$, go to 2.
4. for $count = 1 \ldots t$ do if $rabintest(n) =$ "$fail$" go to 2.
5. return(n).

In [BDL91], it is shown that if one accepts an upper limit on the number of candidates to be examined (and therefore a small probability that the algorithm fails altogether), one can estimate the error probability of *probprimeinc* in the case where $v = e = 1$. One takes the numbers in table 2 as the point of departure. If the maximal number of candidates is $c \cdot log(2^k)$, then the numbers in the table should be multiplied by c^2. The algorithm will fail with probability $exp(-2c)$.

4.1.4 Satisfying Security Constraints

In this section, we will discuss ways to ensure that the primes we produce will satisfy constraints 2 and 3 mentioned in Section 3.3.

To fix some notation, let p be the prime to be produced, such that r divides $p-1$, s divides $p+1$, and t divides $r-1$, where r, s, t are prime numbers.

We show here a variant of an algorithm of Gordon [Gor84], which works by first constructing t, s from scratch, then r from t, and finally p from r, s.

If I is the interval $[a \ldots b]$, we let $\sqrt{I}$ denote the interval $[\sqrt{a} \ldots \sqrt{b}]$.

We assume the existence of a procedure *initrand*, which will initialize a scheme for generation of random or nearly random bits (see Section 5), using a random seed which is passed as a parameter. All subsequent calls to *randomchoice* will refer to the seed used with *initrand*.

A concrete implementation will have to choose fixed functions c_1, c_2 which are used in the procedure *strongprime* to control the size of the prime factors generated for $p \pm 1$ and $r-1$. We discuss below how to choose these functions.

PROCEDURE STRONGPRIME(seed se, interval I, exponent e)

Output: a prime p chosen at random from interval I based on seed s, such that it can be used in RSA with public exponent e. Security conditions from Section 3.3 are satisfied.

1. $initrand(se)$
2. $t := probprime(c_1(I)\sqrt{I}, 1, 1)$
3. $s := probprime(c_2(I)\sqrt{I}, 1, 1)$
4. $r := probprime(c_2(I)\sqrt{I}, t, 1)$

```
5. $p_0 := s^{r-1} - r^{s-1} \bmod rs$
6. if $p_0$ is even then $p_0 = p_0 + rs$
7. return $probprime(I, p_0, e)$
```

When the interval I is $[a \dots b]$, one possible choice for the functions c_1, c_2 is

$$c_1(I) = \frac{1}{2 \cdot bitlength(a)}, \quad c_2 = \frac{1}{2 \cdot \sqrt{bitlength(a)}}$$

This choice allows the maximum possible size of r, s, t such that we still have a good chance of finding a prime in the interval I with the right properties. At the other extreme, one can replace the constant 2 in the formulas for c_1, c_2 by a larger number, chosen such that r, s, t will be of the minimum required size (see Section 3.3). This will give a larger number of primes to choose from in the interval I.

An even more advanced idea is to choose c_1, c_2 at random between the two extremes each time these values are needed. This will make it possible to generate virtually every existing prime that satisfies the security constraints.

The calls to *probprime* in the procedure may of course be replaced by calls to *probprimeinc*, which will give an efficiency improvement.

4.2 Provable Primes

Provable primes are numbers generated in such a way that one can prove with certainty that they are primes. Even though primality tests that always give correct answers are quite complicated and inefficient, it is possible to generate provable primes quite efficiently. The reason for this is that when we generate a number from scratch, we may know some side information which can help us in proving the number to be prime.

4.2.1 Maurer's Algorithm

Maurer [Mau89] has proposed a recursive algorithm for generating provable primes, based on the following number theoretic result by Pocklington:

Let $n - 1 := FR$, and let $q_1, \dots q_t$ be the distinct prime factors of F. Suppose there exists a number a such that

$$a^{n-1} \equiv 1 \bmod n$$

and for all $i = 1 \dots t$,

$$gcd(a^{(n-1)/q_i} - 1, n) = 1,$$

then if $F > \sqrt{n}$, n is a prime.

This suggests a straightforward algorithm for generating a random prime in some interval $[low \dots high]$: first generate recursively $q_1, q_2, \dots$, where $q_1 \geq q_2 \geq \dots$. This goes on until F, the product of the q's is larger than $\sqrt{high}$. Then choose random even R-values such that $n = FR + 1$ is in $[low \dots high]$, until an n-value can be proven prime.

α	$\rho(\alpha)$
1.5	0.59453 48919
2.0	0.30685 28194
2.5	0.13031 95618
3.0	0.04860 83883
3.5	0.01622 95932
4.0	0.00491 09256
4.5	0.00137 01177
5.0	0.00035 47247
6.0	0.00001 96497
7.0	0.00000 08746
8.0	0.00000 00323
9.0	0.00000 00010

Table 1: Distribution of the largest prime factor.

Maurer shows that if the q's are large, nearly any choice of a will suffice for proving primality of n (provided n really is prime!), so we are not likely to miss any primes, even if we only try once for each candidate. Furthermore, it is shown that if the number e is used to prove primality of the prime factors of $p-1$ and $q-1$, then the resulting RSA system with e as public exponent will not be easy to break by repeated encryption.

If the goal is to generate a prime uniformly chosen from the interval, then we should know something about the distribution of the prime factors of $n-1$, in particular the distribution of their sizes is necessary. Fortunately, the distribution of the size of the largest prime factor of a number is well known. More precisely, for large N, one can compute $\rho(\alpha)$, the fraction of numbers x less than N whose largest prime factor is less than $x^{1/\alpha}$. In Table 1, sample values of this function are given. From heuristic arguments, this distribution function seems to be also applicable if we add the condition that the number we are looking at is a prime minus 1.

Finally, we note that by the recursive nature of the algorithm, it is of course necessary to have some lower limit for the primes generated, below which one generates a prime, simply by exhaustive search and test division.

4.2.2 Various Tricks for Optimization

What can be done to speed up this algorithm? First of all, we should of course use test division by small primes on a candidate before going into expensive exponentiations. Maurer suggests that since all candidates are of the form $n = FR + 1$ for fixed F, one can translate the condition that none of the small primes divide n into a condition on R. This will be faster to check, since R is usually much smaller than n (and certainly less than $\sqrt{n}$). More concretely, if $n = FR + 1 \equiv 0 \bmod p$, then $R \equiv -F^{-1} \bmod p$. So we can precompute $-F^{-1}$ modulo each small prime used for test division, and for every candidate check for each p if R has residue $-F^{-1} \bmod p$.

Furthermore, even if a candidate passes the test division, there is no need to try immediately proving that it is prime. A better approach is to do a Rabin test with base 2 (see Section 4.1.1). Base 2 gives the most efficient Rabin test possible. Like any other base, it will exclude no prime, and from practical experience, it will exclude virtually all composites (this is also supported by theoretical results [Pom81]). If n passes this test, we have implicitly checked that $2^{n-1} \equiv 1 \bmod n$. It is therefore advantageous to use Pocklingtons result with $a = 2$, since we have then already checked the first condition.

A final optimization concerns Pocklingtons result, which has been improved by Brillhart, Lehmer and Selfridge [BLS75]:

Given $n = FR+1$, suppose we have an a that satisfies the conditions of Pocklington. Let R' be the odd part of R, and $F' = (n-1)/R'$. Let r, s be defined by $R' = 2F's+r$, where $1 \leq r < 2F'$. Suppose $F' > \sqrt[3]{n}$. Then n is prime if and only if $s = 0$ or $r^2 - 8s$ is not a square.

This refined condition is slightly more computationally costly to verify. However, this makes little difference in practice, at least if we look at the variation using the Rabin test. For this variation, experience shows that the above result will only be used on the final candidate, and the extra computation required is only some trivial manipulations to find F', R', s, r, and perhaps a square root computation, which takes time negligible compared to the exponentiations.

Furthermore, the distribution of the largest prime factor shows that only 5% of the numbers x are expected to have all prime factors less than $x^{1/3}$. We suggest that we can easily live without these 5%, in which case we never have to generate more than one prime factor of $n - 1$. This will simplify the code and save time compared to Maurer's original version for the approximately 30% of the numbers that have their largest prime factor less than $x^{1/2}$. It will, however, bias the distribution of the primes generated slightly, compared to the uniform distribution over the primes. This is not a problem for application to RSA, though, since the modification will tend to generate primes p with larger prime factors of $p - 1$.

4.2.3 Satisfying Security Constraints

Of the conditions in Section 3.3, the ones on $p - 1$ and $r_p - 1$ are very easy to ensure with Maurer's algorithm: one simply sets up a lower limit for the size of q_1, the largest prime factor of the candidate prime minus 1. This limit may be set to, e.g., 75 and 40 bits on the first, respectivly second level of recursion. In addition, one should use $a = e$ when proving primality of r_p.

The condition on $p + 1$, can be solved similarly as for probable primes (once again, our target is a prime in the interval $[low \ldots high]$):

1. Using Maurer's algorithm, generate primes r, s of at least 75 bits, such that $r > \sqrt[3]{high}$.

2. Using the same method as in the *strongprime* procedure, find an odd p_0, such that $p_0 \equiv 1 \mod r$ and $p_0 \equiv -1 \mod s$.

3. Choose random values of L in some appropriate interval, until a number of the form $p = 2Lrs + p_0$ can be proved prime by Maurer's method (or Theorem 3).

p_0 is likely to have about the same bit length as rs, so since s must be of length at least 75 bits, this means that r can be of length at most $length(p_0) - 75$. This introduces a slight deviation from the uniformity of primes otherwise produced by Maurer's method. Table 1 indicates that for 256 bit primes and a 75 bit s, we loose at most 20% of the primes this way.

4.3 A Hybrid Method

It is possible to combine the provable and the probable method. Using the notation from the *strongprime* procedure, this works roughly as follows:

PROCEDURE HYBRIDPRIME(seed se, interval I, exponent e)

Output: a prime p chosen at random from interval I based on seed s, such that it can be used in RSA with public exponent e. Security conditions from Section 3.3 are satisfied.

1. $initrand(se)$
2. $t := probprime(c_1(I)\sqrt{I}, 1, 1)$
3. $s := probprime(c_2(I)\sqrt{I}, 1, 1)$
4. Search through numbers of the form $r = 2kt + 1$, k chosen such that r is in $c_2(I)\sqrt{I}$, until r can be proved prime using Pocklingtons result, with $F = t$ and $a = e$.
5. $p_0 = s^{r-1} - r^{s-1} \bmod rs$
6. if p_0 is even then $p_0 = p_0 + rs$
7. Search through numbers of the form $p = 2krs + p_0$, k chosen such that p is in I and $gcd(p-1, e) = 1$, until p can be proved prime using (the improvement of) Pocklingtons result, with $F = r$. Return p.

This procedure is constructed such that IF t is prime then r and p MUST be primes. Thus, we only have to worry about the error probability when generating t and s. Since these primes are generated "from scratch", the estimates for the error probability given earlier will apply directly. Moreover, this method will be faster than Maurer's method, since we can get rid of all of the recursion below the level of t. It is also faster than *strongprime* because we do not have to do many Rabin tests on r and p, in particular all the tricks for speedup of Maurer's algorithm apply here.

Finally, note that by using $a = e$ when proving primality of r, we have implicitly checked that the condition on iterated encryption is satisfied: the multiplicative order of e, i.e., the number of encryptions needed to reconstruct the plaintext, is at least divisible by t.

5 Generation of Pseudorandom Bits

If a hardware source of randomness is not available (which will be the case in many environments), it is likely that only a very limited number of random bits will be at our disposal: for example, there is a limit to how many random characters we can ask a user to type.

What is needed in this situation is a method that will take a short random bit string and stretch it to a much longer string that is SEEMINGLY random, e.g. for any practical purpose, it is as good as a really random string.

An example: suppose we have a strong encryption algorithm E, where $E_K(M)$ denotes encryption of M under key K. Then the procedure *initrand* would interpret its input as a pair of plaintext, key M, K and store this in a fixed memory location. Later, the procedure *randomchoice* could obtain a seemingly random bit string of any length by computing $E_K(M), E_K(E_K(M)), \ldots$.

Many variations on this theme are possible. Also, good methods exist that use modular arithmetic [MiSc91].

6 Conclusion, Choice of Algorithm

From a cryptographic point of view, there is not much practical reason for using provable primes rather than probable ones. Any application will rely on secrecy of a number of keys. There is always a non-zero probability that these keys are guessed by an enemy, so removing error-probability from the prime generation will never remove all error probabilities from the system.

Hence the question rather is whether one can efficiently bring down the error probability to an acceptable level. We have seen that for the Rabin test, only a small number (less than 5) tests are enough to get a probability that is comparable to the probability of guessing e.g., a random DES key.

Taking this to be an acceptable error probability, probable primes tend to be a bit faster than provable ones. Moreover, in applications where only a small amount of storage for program and data is available, probable primes have a distinct advantage: the Rabin test is simple enough to make a very compact implementation possible.

However, as we have seen in Section 4.3, the methods that provable primes are based on can still be very useful.

Finally, we discuss the security constraints: the (rather complicated) methods described by *strongprime* and *hybridprime* are necessary if we want to check with certainty that the constraints are satisfied. This is motivated by the fact that for primes of less than 300-400 bits, there is some nonnegligible (but small) probability that a random prime will not satisfy the constraints (see table 1). With increasing size of primes, this probability rapidly becomes completely negligible, however. Therefore, a much simpler solution than *strongprime*, for example a single call to *probprime*, can safely be used for primes above 300-400 bits. For smaller primes, the simple solution may still be used, if one is prepared to take a small risk that one of the primes does not quite satisfy the demands. Depending on the application, this may be acceptable.

However, it should be noted that, independently of the security considerations, there is an efficiency benefit in building a large prime p from factor(s) of $p - 1$, similarly to what is done in *hybridprime*.

References

[BaSh89] E. Bach and J. Shallit, "Factoring with cyclotomic polynomials," *Mathematics of Computation*, vol. 52, pp. 201-219, 1989.

[BDL91] J. Brandt, I.B. Damgård and P. Landrock, "Speeding up prime number generation," *Abstracts of ASIACRYPT'91*, Fujiyoshida, Japan.

[BLS75] J.Brillhart, D.H.Lehmer and J.L.Selfridge, "New primality criteria and factorizations of $2^m \pm 1$, *Mathematics of Computation*, vol. 29, pp. 620-647, 1975.

[DLP] I.B. Damgård, P. Landrock and C. Pomerance, "Improved bounds for the Rabin primality test," to appear in: *Mathematics of Computation.*

[Gor84] J. Gordon, "Strong primes are easy to find," in: *Advances in Cryptology - EUROCRYPT'84*, T. Beth, N. Cot and I. Ingemarsson eds., Lecture Notes in Computer Science no. 209, Springer-Verlag, Berlin-Heidelberg-New York, pp. 216-223, 1985.

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

[Kob87] N. Koblitz, *A Course in Number Theory and Cryptography*, Springer-Verlag, Berlin-Heidelberg-New York, 1987.

[LeMa90] A.K. Lenstra and M.S. Manasse, "Factoring with two large primes," in: *Advances in Cryptology - EUROCRYPT'90*, I.B. Damgård ed., Lecture Notes in Computer Science no. 473, Springer-Verlag, Berlin-Heidelberg-New York, pp. 72-82, 1991.

[LLMP90] A.K. Lenstra, H.W. Lenstra Jr., M.S. Manasse and J.M. Pollard, "The number field sieve," *Proceedings of STOC'90,*, 1990.

[Mau89] U.M. Maurer, "Fast generation of secure RSA-products with almost maximal diversity," 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. 636-647, 1990.

[MiSc91] S. Micali and C.P. Schnorr, "Efficient perfect polynomial random number generators," *Journal of Cryptology*, vol. 3, no. 3, pp. 157-172, 1991.

[Pom81] C.Pomerance, "On the distribution of pseudoprimes," *Mathematics of Computation*, vol. 37, pp. 587-593, 1981.

Chapter 10

Implementation Guidelines for Arithmetic Computation

Contents

1 Introduction

The integrity primitives IBC-hash (Chapter 5), RSA (Chapter 7) and COMSET (Chapter 8) are based on calculations with large integers. The implementation of such calculations is non-trivial, as the size of the numbers used far exceeds the word-size of computers. Therefore this appendix gives some guidelines for software implementation of large number arithmetic. For more detailed information, [Knu81] is a good reference.

2 Elementary Operations

The addition, subtraction, multiplication and division of large integers can be implemented according to the classical algorithms familiar from pencil and paper calculations.

For software implementations, one should use 2 to the power of the word-size of the computer as a base, rather than 10.

Whereas the performance of the classical algorithms is good in the case of addition and subtraction, one can do much better for multiplication and division. Such fast algorithms are described in [Knu81].

Below an algorithm for the division of the large integer u by the large integer v is given. Let $u = (u_1u_2\ldots u_{m+n})_b$ and $v = (v_1v_2\ldots v_n)_b$ be nonnegative integers in radix-b notation. That is,

$$\begin{aligned} u &= \sum_{i=1}^{m+n} u_i b^{m+n-i} \\ v &= \sum_{i=1}^{n} u_i b^{n-i}. \end{aligned}$$

Let $v_1 \neq 0$ and $n > 1$. The radix-b quotient $\lfloor u/v \rfloor = (q_0q_1\ldots q_m)_b$ and the remainder $u \bmod v = (r_1r_2\ldots r_n)_b$ is calculated with the following algorithm (Algorithm D, pp. 257-258 of [Knu81]).

D1. [Normalize] Set $d := \lfloor b/(v_1{+}1)\rfloor$. Then set $(u_0u_1u_2\ldots u_{m+n})_b$ equal to $(u_1u_2\ldots u_{m+n})_b$ times d, and set $(v_1v_2\ldots v_n)_b$ equal to $(v_1v_2\ldots v_n)_b$ times d.

D2. [Initialize j] Set $j := 0$.

D3. [Calculate $\hat{q}$] If $u_j = v_1$, set $\hat{q} := b-1$; otherwise set $\hat{q} := \lfloor (u_jb+u_{j+1})/v_1/rfloor.$ Now test if $v_2\hat{q} > (u_jb + u_{j+1} - \hat{q}v_1)b + u_{j+2}$; if so, decrease $\hat{q}$ by 1 and repeat this test.

D4. [Multiply and subtract] Replace $(u_ju_{j+1}\ldots u_{j+n})_b$ by $(u_ju_{j+1}\ldots u_{j+n})_b$ minus $\hat{q}$ times $(v_1v_2\ldots v_n)_b$. This step consists of a simple multiplication by a one-place number, combined with a subtraction. The digits $(u_ju_{j+1}\ldots u_{j+n})_b$ should be kept positive. If this is not the case, decrease $\hat{q}$ by 1 before replacing $(u_ju_{j+1}\ldots u_{j+n})_b$ by its new value.

D5. [Loop on j] Set $q_j := \hat{q}$. Increase j by one. If $j \leq m$ go back to D3.

D6. [Unnormalize] Now $(q_0 q_1 \ldots q_m)_b$ is the desired quotient, and the desired remainder may be obtained by dividing $(u_{m+1} \ldots u_{m+n})_b$ by d.

3 Modular Computations

Modular addition, subtraction and multiplication can be implemented using the non-modular operations desribed above. The modulo reduction consists of adding or subtracting the modulus for the modular addition and subtraction. The result of a modular multiplication can be determined as the remainder when dividing the non-modular product of the multiplicands by the modulus.

However more efficient algorithms to do modular calculations are known. A software library for digital signal processors is described in [DuKa90] . In [Bar86] a faster algorithm for the modulo reduction is given. One could also consider to do multiplication and modulo reduction not subsequently, but to start the modular reduction already on intermediate results of the multiplication. An example of a fast modular multiplication algorithm is [Mon85].

The following algorithm makes modular exponentiation feasible for huge exponents: to compute $a^e \bmod n$ it is not necessary to do e multiplications. If s is the length of e in bits, $2s$ multiplications are enough. For that, the *square and multiply algorithm* (see [Knu81]) is used. In pseudocode it works like this:

```
$Y := 1$;
$Z := a$;
for $i := 0$ to $(s-1)$ do {
    if $(e_{s-1-i} = 1)$
        then $Y := Z * Y \bmod n$;
    $Z := Z^2 \bmod n$;
    }
```

The bits of e are denoted by $e_0, e_1, \ldots, e_{s-1}$, where e_0 is the most significant bit. At the end of the algorithm Y is equal to $a^e \bmod n$.

This is the most elementary, but quite efficient, way to do modular exponentiation. Some possible improvements are desribed in [BoCo89].

4 Determination of the greatest common divisor and modular inverses

The extended Euclidian algorithm ([Knu81]) can be used in order to determine the greatest common divisor of two nonnegative integers as well as modular inverses. Given two nonnegative integers u and v this algorithm determines values u_1, u_2, and u_3 such that u_3 is the greatest common divisor of u and v and that $u_3 = uu_1 + vu_2$ holds.

Extended Euclidian algorithm:

```
$u_1 := 1$;
$u_2 := 0$;
$u_3 := u$;
$v_1 := 0$;
$v_2 := 1$;
$v_3 := v$;
while ($v_3 > 0$) {
    $q := u_3$ div $v_3$;
    $t_1 := u_1 - v_1 q$;
    $t_2 := u_2 - v_2 q$;
    $t_3 := u_3 - v_3 q$;
    $u_1 := v_1$;
    $u_2 := v_2$;
    $u_3 := v_3$;
    $v_1 := t_1$;
    $v_2 := t_2$;
    $v_3 := t_3$;
    }
```

u is invertible modulo v if and only if $u_3 = 1$ holds. In this case $1 \equiv uu_1 \pmod{v}$ holds, u_1 is the inverse of u modulo v .

5 Jacobi symbol

The following algorithm in pseudo-code determines the value of the Jacobi symbol. This is needed for the Rabin variant of RSA described in Chapter 7. The mathematical background may e. g. be found in [Kob87].

We evaluate $J = \left(\frac{a}{b}\right)$ as follows:

```
$J = 1$;
while ($a > 1$) do {
    if ($a \bmod 2 == 0$) {
        if ($(b^2 - 1)/8 \bmod 2 == 1$)
            $J = -J$;
        $a = a/2$;
    }
    else {
        if ($(a-1) \cdot (b-1)/4 \bmod 2 == 1$)
            $J = -J$;
        $s = b \bmod a$;
        $b = a$;
```

```
            a = s;
        }
    }
    if (a == 0)
        J = 0;
```

Here s represents an auxiliary storage variable.

References

[BoCo89] J. Bos and M. Coster, "Addition chain heuristics," in: *Advances in Cryptology - CRYPTO'89*, G. Brassard ed., Lecture Notes in Computer Science no. 435, Springer-Verlag, Berlin-Heidelberg-New York, pp. 400-407, 1990.

[Bar86] P. Barrett, "Implementing the Rivest Shamir Adleman public key encryption algorithm on a standard digital signal processor," in: *Advances in Cryptology - CRYPTO'86*, A.M. Odlyzko ed., Lecture Notes in Computer Science no. 263, Springer-Verlag, Berlin-Heidelberg-New York, pp. 311-323, 1987.

[DuKa90] S.R. Dussé and B.R. Kaliski Jr., " A cryptographic library for the Motorola DSP 56000," in: *Advances in Cryptology - EUROCRYPT'90*, I.B. Damgård ed., Lecture notes in Computer Science no. 473, Springer-Verlag, Berlin-Heidelberg-New York, pp. 230-244, 1991.

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

[Kob87] N. Koblitz, *A Course in Number Theory and Cryptography*, Springer-Verlag, Berlin-Heidelberg-New York, 1987.

[Mon85] P.L. Montgomery, "Modular multiplication without trial division," *Mathematics of Computation*, vol. 44, pp. 519-521, 1985.