emSecure-RSA

Digital signature suite

User Guide & Reference Manual

Document: UM12002
Software Version: 2.40
Revision: 0
Date: October 10, 2018

A product of SEGGER Microcontroller GmbH

www.segger.com

Disclaimer

Copyright notice

Trademarks

Contact address

SEGGER Microcontroller GmbH

Ecolab-Allee 5
D-40789 Monheim am Rhein

Germany

Tel. +49 2173-99312-0
Fax. +49 2173-99312-28
E-mail: support@segger.com
Internet: www.segger.com

Manual versions

This manual describes the current software version. If you find an error in the manual or a problem in the software, please inform us and we will try to assist you as soon as possible. Contact us for further information on topics or functions that are not yet documented.

Print date: October 10, 2018

Software	Revision	Date	By	Description
2.40	0	181010	PC	Update to latest software version.
2.38	0	180621	PC	Update to latest software version.
2.36	0	171116	PC	Update to latest software version.
2.34	0	170823	PC	Chapter “API Reference” • `SECURE_RSA_VERSION` added. • `SECURE_RSA_SignEx()` added. • `SECURE_RSA_VerifyEx()` added. • `SECURE_RSA_GetVersionText()` added. • `SECURE_RSA_GetCopyrightText()` added.
2.32	0	170728	PC	Update to latest software version.
2.30	0	170518	PC	Chapter “Performance” • Updated for latest Embedded Studio and algorithms. • Added benchmark performance output. • Added complete listing of benchmark application. Chapter “Configuring emSecure-RSA” • Updated for using shared emCrypt component. Chapter “API Reference” • `SECURE_RSA_Init()` added. • Return values documented for additional failures.
2.20	0	151202	PC	Chapter “API Reference” • `SECURE_RSA_SignDigest()` added. • `SECURE_RSA_VerifyDigest()` added. • `SECURE_RSA_HASH_Add()` added. • `SECURE_RSA_HASH_Init()` added. • `SECURE_RSA_HASH_Sign()` added. • `SECURE_RSA_HASH_Verify()` added. Chapter “emSecure-RSA walkthrough” • Section “Incremental sign and verify” added. Chapter “Using OpenSSL with emSecure-RSA” • Added. Chapter “API Reference” • Section “Configuring emSecure-RSA” updated with selectable hash. Index “Index of functions” • Added.
2.10	0	150713	JL	Documentation converted to SEGGER emDoc format. • Minor documentation fixes. • Documentation in manual automatically derived from source code.
1.02	0	150522	JL	Chapter “Introduction” • “Package content” updated. Chapter “Working with emSecure-RSA” • Utility description updated. Chapter “emSecure API” • Section “Configuring emSecure-RSA” updated. • Compile-time configuration updated.
1.00	1	141016	JL	Chapter “Support” • Added. Chapter “Appendix” • Flowchart “emSecuring a product” added.
1.00	0	140827	JL	Initial Release.

About this document

Assumptions

This document assumes that you already have a solid knowledge of the following:

- The software tools used for building your application (assembler, linker, C compiler).
- The C programming language.
- The target processor.
- DOS command line.

If you feel that your knowledge of C is not sufficient, we recommend *C: A Reference Manual* by Harbison and Steele (ISBN 0--13--089592X). This book provides a complete description of the C language, the run-time libraries, and a style of C programming that emphasizes correctness, portability, and maintainability.

How to use this manual

This manual explains all the functions and macros that the product offers. It assumes you have a working knowledge of the C language. Knowledge of assembly programming is not required.

Typographic conventions for syntax

This manual uses the following typographic conventions:

Style	Used for
Body	Body text.
`Parameter`	Parameters in API functions.
`Sample`	Sample code in program examples.
`Sample comment`	Comments in program examples.
`User Input`	Text entered at the keyboard by a user in a session transcript.
`Secret Input`	Text entered at the keyboard by a user, but not echoed (e.g. password entry), in a session transcript.
Reference	Reference to chapters, sections, tables and figures.
Emphasis	Very important sections.
SEGGER home page	A hyperlink to an external document or web site.

Table of contents

Chapter 1

Introduction to emSecure-RSA

This section presents an overview of emSecure-RSA, its structure, and its capabilities.

1.1 What is emSecure-RSA?

emSecure-RSA is a SEGGER software package that allows creation and verification of digital signatures. One important feature is that emSecure-RSA can make it impossible to create a clone of an embedded device by simply copying hardware and firmware.

And it can do much more, such as securing firmware updates distributed to embedded devices and authenticating licenses, serial numbers, and sensitive data.

emSecure-RSA offers 100% protection against hacking. It is not just nice to have, but in fact a must-have, not only for critical devices such as election machines, financial applications, or sensors.

Compromised devices are dangerous in several ways, not just from a commercial point of view. They hamper manufacturers' reputation and might entail severe legal disputes. Not addressing the issue of hacking and cloning is irresponsible.

Based on asymmetric encryption algorithms with two keys, emSecure-RSA signatures cannot be forged by reverse engineering of the firmware. A secure, private key is used to generate the digital signature, whereas a second, public key is used to authenticate data by its signature. There is neither a way to get the private key from the public key, nor is it possible to generate a valid signature without the private key.

The emSecure-RSA source code has been created from scratch for embedded systems, to achieve highest portability with a small memory footprint and high performance. However, usage is not restricted to embedded systems.

With its easy usage, it takes less than one day to add and integrate emSecure-RSA into an existing product. emSecure-RSA is a very complete package, including ready-to-run tools and functionality for generation of keys and signatures.

1.2 Why should I use emSecure-RSA?

emSecure-RSA offers a fast and easy way to prevent hacking and cloning of products. Not addressing the issue of hacking and cloning would be irresponsible.

Security consideration

If you want to check the integrity of your data, for instance the firmware running on your product, you would normally include a checksum or hash value into it, generated by a CRC or SHA function. Hashes are excellent at ensuring a critical data transmission, such as a firmware download, has worked flawlessly and to verify that an image, stored in memory, has not changed. However they do not add much security, as an attacker can easily compute the hash value of modified data or images.

Digital signatures can do more. In addition to the integrity check, which is provided by hash functions, a digital signature assures the authenticity of the provider of the signed data, as only he can create a valid signature. emSecure-RSA creates digital signatures using the RSA cryptosystem that has proven robust against decades of attacks on the algorithms. For the default of 2048-bit key sizes, it is considered well beyond the capability of governments, with all their computing power and using the very latest number-theoretic methods, to recover a properly generated RSA private key before 2030, and most probably well beyond that.

emSecure-RSA can be used for two security approaches:

- *Anti-hacking* on page 17: Prevent tampering or exchange of data, for example the firmware running on a product, with non-authorized data.
- *Anti-cloning* on page 18: Prevent a firmware to be run on a cloned hardware device.

1.3 Features

emSecure-RSA is written in standard ANSI C and can run on virtually any CPU. Here’s a list summarizing the main features of emSecure-RSA:

- Dual keys, private and public make it 100% safe
- Hardware-independent, any CPU, no additional hardware needed
- High performance, small memory footprint
- Simple API, easy to integrate
- Applicable for new and existing products
- Complete package, key generator and tools included
- Drag-and-drop Sign And Verify application included
- Full source code

1.4 Recommended project structure

We recommend keeping emSecure-RSA separate from your application files. It is good practice to keep all the source files (including the header files) together in the subdirectories of your project's root directory as they are shipped. This practice has the advantage of being very easy to update to newer versions of emSecure-RSA by simply replacing the directories. Your application files can be stored anywhere.

Note

When updating to a newer emSecure-RSA version: as files may have been added, moved or deleted, the project directories may need to be updated accordingly.

1.5 Package content

emSecure-RSA is provided in source code and contains everything needed. The following table shows the content of the emSecure-RSA Package:

Files	Description
`Application`	emSSL sample applications for bare metal and embOS.
`Config`	Configuration header files.
`Doc`	emSecure-RSA documentation.
`CRYPTO`	Shared cryptographic library source code.
`SECURE`	emSecure-RSA implementation code.
`SEGGER`	SEGGER software component source code used in emSecure-RSA.
`Sample/Config`	Example emSecure-RSA configuration.
`Sample/Keys`	Example emSecure-RSA key pairs.
`Windows`	Supporting applications in binary and source form.

1.5.1 Include directories

You should make sure that the include path contains the following directories (the order of inclusion is of no importance):

- `Config`
- `CRYPTO`
- `SECURE`
- `SEGGER`
- `SYS`

Note

Always make sure that you have only one version of each file!

It is frequently a major problem when updating to a new version of emSecure-RSA if you have old files included and therefore mix different versions. If you keep emSecure-RSA in the directories as suggested (and only in these), this type of problem cannot occur. When updating to a newer version, you should be able to keep your configuration files and leave them unchanged. For safety reasons, we recommend backing up (or at least renaming) the `SECURE` directories before to updating.

Chapter 2

Working with emSecure-RSA

This chapter gives some recommendations on how to use emSecure-RSA in your applications. It explains the steps of “emSecuring” a product.

2.1 Introduction

emSecure-RSA is created to be simple but powerful, and easy to integrate. It can be used in new products and even extend existing ones as emSecure-RSA is a software solution and no additional hardware is required. The code is completely written in ANSI C and can be used platform- and controller-independent.

emSecure-RSA has been created from scratch to achieve highest portability and performance with a very small memory footprint. It enables you to profit from the security of digital signatures in embedded applications, even on small single-chip microcontrollers without the need of additional hardware such as external security devices or external memory.

emSecure-RSA is a complete package. It includes ready-to-run tools to generate keys and signatures, to sign and verify data and to convert the keys and signatures into compilable formats.

The required key pairs can be generated with the included tool. The generated keys can be exported into different formats to be stored on the application code or loaded from a key file. This allows portability and exchangeability between different platforms.

Signing data, for instance firmware images, can be done with the included tool. It is also possible to integrate the signing process directly into a production application running on any PC or even on a microcontroller.

Once a signature is generated, the signed data can be verified by its signature in an embedded application or on an external application communicating with the device. Verifying data takes less than 40 ms on a Cortex-M4, running at 200 MHz, which is not significantly more time for a bootloader to start a firmware.

emSecure-RSA includes all required source code to integrate signature generation directly into your production process and data verification into your application or firmware.

emSecure-RSA incorporates proven security algorithms as proposed by NIST. The algorithms are proven to be cryptographically strong and can provide a maximum of security to your applications.

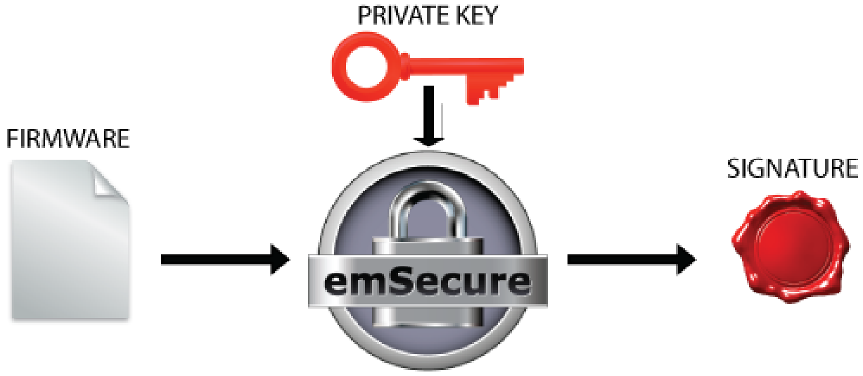
emSecure-RSA is licensed in the same way as other SEGGER middleware products and not covered by an open-source or required-attribution license. It can be integrated in any commercial or proprietary product without the obligation to disclose the combined source. It can be used royalty-free in your product.

2.2 Anti-hacking

Authentication of firmware

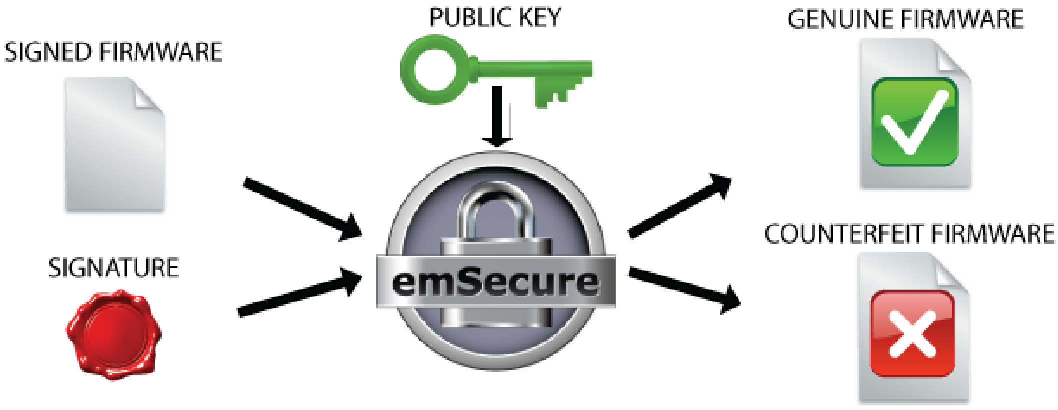
To make sure only authorized firmware images are run on a product the firmware image will be signed with emSecure-RSA. To do this an emSecure-RSA key pair is generated one time.

The private key will be included in the production process of the firmware. Once a firmware is created and ready to be shipped or included into a product it will be signed with this private key. The signature will be transferred and stored in the product alongside the firmware.



The public key will be included in the bootloader of the product, which manages firmware updates and starts the firmware.

On a firmware update and when starting the product, the bootloader will verify the firmware by its signature. If they match, the firmware is started, otherwise the application will stay in the bootloader or even erase the firmware.

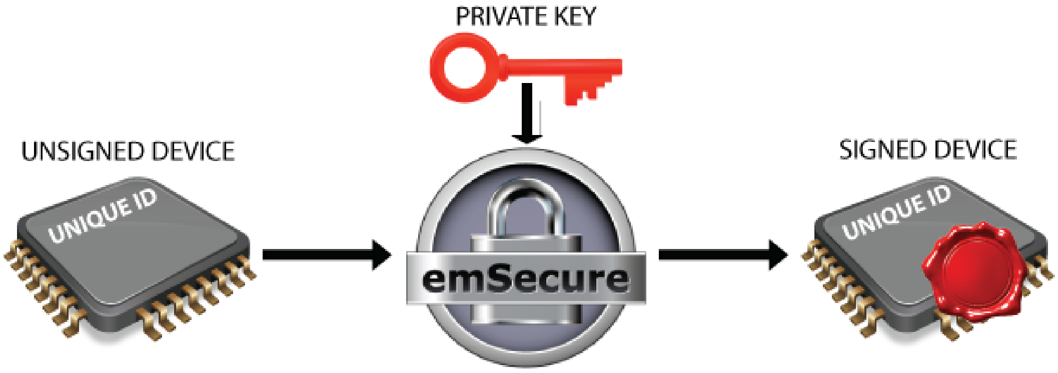


2.3 Anti-cloning

Authentication of hardware

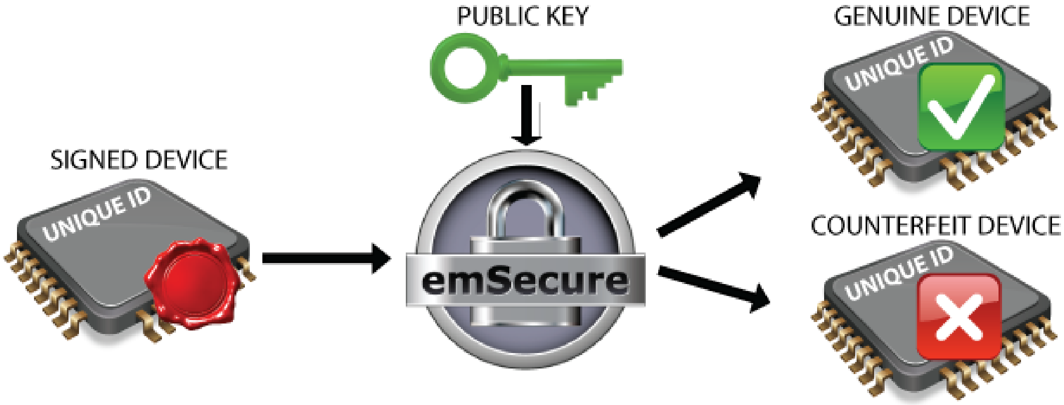
To make sure a product cannot be re-produced by non-authorized manufacturers, by simply copying the hardware, emSecure-RSA will be used to sign each genuine product unit.

First an emSecure-RSA key pair is generated one time. This is likely done at the production site.



The private key will be included in the production process of the product. At the end of the production process, after the unit is assembled and tested, some hardware-specific, fixed, and unique data, like the unique id of the microcontroller is read from the unit. This data is signed by emSecure-RSA with the private key and the signature is written back to the unit into an OTP area or a specified location on memory.

The public key will be included in the firmware which will run on the product. When the firmware is running it will read the unique data from the unit and verify it with the signature. When the signature does not match, for example, when it was simply copied to a counterfeit unit with other unique data, the firmware will refuse to run.



2.4 Additional measures to keep the system secure

When it comes to the degree of security emSecure-RSA offers, there is an easy answer: It is unbreakable because no one can generate a valid signature without knowledge of the private key.

Putting enough effort into getting the bootloader or firmware image, disassembling and analyzing it and modifying the application to bypass the security measures, hackers might be able to clone a product or use alternative firmware images. However this will only work until a firmware update is done.

There are additional ways to increase overall system security:

- The private key has to be kept private.
- Private keys should best be generated on a dedicated machine that has no connection to a network and controlled access.
- Private keys can be generated from a pass-phrase, which means the pass-phrase should not be too easy to guess. The pass-phrase length is not limited. The company name is not a good choice. The pass-phrase has to be kept private, too.
- The private key might also be encrypted and is only decrypted while it is used in production.
- The bootloader should should be stored in a memory which can be protected against read- and write-access.
- The firmware should be protected against external read-access.
- The verification process can be done in multiple places of the application. A tool communicating with the product, like a PC application, might carry our additional checks.

Chapter 3

Included applications

This chapter describes the applications which are part of the emSecure-RSA package. The executables can be found at `/Windows/SECURE`. The source code of the basic utilities is included.

emSecure-RSA includes all basic applications required for securing a product. The applications' source-code is included and provides an easy to use starting point for modifications and integration into other applications.

The source code for the key generator is an add-on: please contact SEGGER for more information.

3.1 emSecure-RSA Key Generator

emSecure-RSA KeyGen generates a public and a private key. The generation parameters can be set via command line options, by default a random 2048 bit key is generated. The keys are saved in a common key file format and can be published and exchanged.

Usage

`emKeyGenRSA.exe` [<Options>]

Command line options

emSecure-RSA KeyGen accepts the following command line options.

Option	Description
`-h`	Print usage information and available command line options.
`-q`	Operate silently and do not print log output.
`-v`	Increase verbosity of log output.
`-k` *string*	Set the key file prefix to *string*. Default is “`emSecure`”.
`-l` *n*	Set the modulus length to *n* bits. Default is *2048*. For modulus lengths that other than 2048 and 3072, *-nf* is required.
`-seed` *n*	Set the initial seed for random number generation to *n*.
`-pw` *string*	Generate the initial seed from the pass phrase *string*.
`-f`	Generate proven primes for the key pair according to FIPS algorithms. This option is default and recommended to be used with a key length of 2048 bits.
`-nf`	Generate probabilistic primes for the key pair.
`-pem`	Write key files in PEM format.

3.2 emSecure-RSA Sign

emSignyRSA digitally signs file content, usually the data to be secured, with a given (private) key file and creates a signature file.

Usage

emSignRSA.exe [<Options>] <InputFile>

Command line options

Option	Description
-h	Print usage information and available command line options.
-q	Operate silently and do not print log output.
-v	Increase verbosity of log output.
-k *string*	Set the key file prefix to *string*. Default is "emSecure".
-sha1	Select SHA-1 as the hash function when generating the digest to be signed.
-sha256	Select SHA-256 as the hash function when generating the digest to be signed.
-sha512	Select SHA-256 as the hash function when generating the digest to be signed.
-pss	Select RSASSA-PSS as the signature scheme to use when creating the signature.
-pkcs	Select RSASSA-PKCS1-v1.5 as the signature scheme to use when creating the signature.

3.3 emSecure-RSA Verify

`emVerifyRSA` accepts a signature file and verifies if the corresponding data file matches the signature.

Usage

`emVerifyRSA.exe` [<Options>] <InputFile>

Command line options

Option	Description
`-h`	Print usage information and available command line options.
`-q`	Operate silently and do not print log output.
`-v`	Increase verbosity of log output.
`-b`	Signature file is pure binary, do not probe the signature file to discover its format.
`-s` *n*	Set the salt length to *n* bytes. For nonempty salts on signing, correct verification requires that only the *salt length* be provided to the signature verifier, not the original salt content.
`-k` *string*	Set the key file prefix to *string*. Default is "`emSecure`".
`-sha1`	Select SHA-1 as the hash function when computing the message digest.
`-sha256`	Select SHA-256 as the hash function when computing the message digest.
`-sha512`	Select SHA-512 as the hash function when computing the message digest.
`-pss`	Select RSASSA-PSS as the signature scheme to use when verifying the signature.
`-pkcs`	Select RSASSA-PKCS1-v1.5 as the signature scheme to use when verifying the signature.

3.4 emSecure-RSA Print Key

`emPrintKeyRSA` exports key and signature files into a format suitable for compilation by a standard C compiler. The output can be linked into your application, so there is no need to load them from a file at runtime. This is especially useful for embedded applications.

Usage

`emPrintKeyRSA.exe` [<Options>] <Input-File>

Command line options

emSecure-RSA PrintKey accepts the following command line options.

Option	Description
`-v`	Increase verbosity of log output.
`-x`	Define objects with external linkage.
`-p` *string*	Prefix object names with *string*. Default is "_".

Chapter 4

emSecure-RSA walkthrough

This section will walk you through generating keys, installing those in an application, and then signing some data and verifying that the signature operation succeeded.

4.1 Generating the keys

Before signing anything, you must generate a set of keys that can sign data and verify the generated signatures. The tool emKeyGenRSA will construct the keys for you: they are generated using secure random numbers such that they are strong.

To use emKeyGenRSA you must choose the key length to use for RSA. The default key length is 2048 bit, which is considered to be secure.

emSecure-RSA keys can be generated randomly or from a pass-phrase. Generating them from a pass-phrase allows you to always re-generate the same keys.

In this example we will choose a 2048 bit key generated from the pass-phrase “SEGGER - The Embedded Experts”

Generating the keys is a matter of running emKeyGenRSA:

```
C:> emKeyGenRSA -l 2048 -pw "SEGGER - The Embedded Experts"

(c) 2014-2018 SEGGER Microcontroller GmbH & Co. KG    www.segger.com
emSecure-RSA KeyGen V2.38 compiled May 17 2018 10:43:55

Generating proven prime key pair with public modulus of 2048 bits
Public encryption exponent is set to 65537
Initial seed is 0xADBE961296F573AD2FA65468E1A8837D
Checking keys are consistent: OK
Writing public key file emSecure.pub.
Writing private key file emSecure.prv.

C:> _
```

The two files that are written contain the public key and the private key, together making a matched key pair. The private key is required when signing some data and must be kept private and secure. The public key is required when verifying some signed data and can be distributed without concern for privacy.

4.2 Testing the keys

You can test out the keys by signing and verifying a small text file:

```
C:> dir >test.txt
C:> emSignRSA test.txt

(c) 2014-2018 SEGGER Microcontroller GmbH & Co. KG    www.segger.com
emSecure-RSA Sign V2.38 compiled May 17 2018 10:43:55

Loading private key from emSecure.prv
Probing file to determine type of key: RSA key detected
  Public modulus is 2048 bits
Loading content from test.txt
  Loaded content is 790 bytes
Writing signature file test.txt.sig

C:> _
```

Once signed, you can verify the signature:

```
C:> emVerifyRSA test.txt

(c) 2014-2018 SEGGER Microcontroller GmbH & Co. KG    www.segger.com
emSecure-RSA Verify V2.38 compiled May 17 2018 10:43:55

Loading public key from emSecure.pub
Probing file to determine type of key: RSA key detected
  Public modulus is 2048 bits
Loading RSA signature from test.txt.sig
  Loading content from test.txt
  Loaded content is 790 bytes
Signature OK.

C:> _
```

If you tamper with the signature or alter the original file, even by *one bit*, the alteration is detected and the signature is not verified:

```
C:> edit test.txt
C:> emVerifyRSA test.txt

(c) 2014-2018 SEGGER Microcontroller GmbH & Co. KG    www.segger.com
emSecure-RSA Verify V2.38 compiled May 17 2018 10:43:55

Loading public key from emSecure.pub
Probing file to determine type of key: RSA key detected
  Public modulus is 2048 bits
Loading RSA signature from test.txt.sig
  Loading content from test.txt
  Loaded content is 790 bytes
Signature NOT VERIFIED!

C:> _
```


4.3 Signing and verifying in your application

Now that you have a pair of keys, it's time to integrate them into your program. You can do that by using `emPrintKeyRSA` which converts the textual form of the key into something that a C program can use.

First we convert the private and public keys into C declarations:

```
C:> emPrintKeyRSA SECURE_RSA_Expert_Key.prv -p _SECURE_RSA_PrivateKey_Expert \
      >SECURE_RSA_PrivateKey_Expert.h

(c) 2014-2018 SEGGER Microcontroller GmbH & Co. KG    www.segger.com
emSecure-RSA PrintKey V2.38 compiled May 17 2018 10:43:55

Probing file to determine type of key

C:> emPrintKeyRSA SECURE_RSA_Expert_Key.pub -p _SECURE_RSA_PublicKey_Expert \
      >SECURE_RSA_PublicKey_Expert.h

(c) 2014-2018 SEGGER Microcontroller GmbH & Co. KG    www.segger.com
emSecure-RSA PrintKey V2.38 compiled May 17 2017 10:43:55

Probing file to determine type of key

C:> _
```

The generated output contains declarations in a format suitable for direct inclusion into C program. The `-p` option sets the prefix for the names of the generated C identifiers.

Now we have the keys, we can write a program that uses the private key to sign an message and a public key to verify it:

```
/*********************************************************************
*                   (c) SEGGER Microcontroller GmbH                  *
*                        The Embedded Experts                        *
*                           www.segger.com                           *
**********************************************************************

-------------------------- END-OF-HEADER -----------------------------

File        : CRYPTO_RSA_Example1.c
Purpose     : Sign and verify a message.

*/

/*********************************************************************
*
*       #include section
*
**********************************************************************
*/

#include "SECURE_RSA.h"
#include "SECURE_RSA_PrivateKey_Expert.h"
#include "SECURE_RSA_PublicKey_Expert.h"
#include <stdio.h>

/*********************************************************************
*
*       Static const data
*
**********************************************************************
*/

static const U8 _aMessage[] = { "This is a message, sign me." };
```

```
/*********************************************************************
*
*       Static data
*
**********************************************************************
*/

static U8 _aSignature[SECURE_RSA_MAX_KEY_LENGTH / 8];

/*********************************************************************
*
*       Public code
*
**********************************************************************
*/

int main(void) {
  int SigLen;
  int Status;
  //
  SigLen = SECURE_RSA_Sign(&_SECURE_RSA_PrivateKey_Expert,
                           0, 0,
                           &_aMessage[0], sizeof(_aMessage),
                           &_aSignature[0], sizeof(_aSignature));
  if (SigLen > 0) {
    printf("Signed message...SUCCESS!\n");
    Status = SECURE_RSA_Verify(&_SECURE_RSA_PublicKey_Expert,
                               0, 0,
                               &_aMessage[0], sizeof(_aMessage),
                               &_aSignature[0], SigLen);
    if (Status > 0) {
      printf("Verified message is correctly signed...SUCCESS!\n");
    } else {
      printf("Correctly signed message did not verify...ERROR!\n");
    }
  } else {
    printf("Failed to sign message...ERROR!\n");
  }
  return 0;
}

/*************************** End of file ****************************/
```

The private key is defined as a constant `SECURE_RSA_PrivateKey_Expert` in `SECURE_RSA_PrivateKey_Expert.h` The public key is defined as a constant `SECURE_RSA_PublicKey_Expert` in `SECURE_RSA_PublicKey_Expert.h`

If you compile and run this application, you will see this output:

```
C:> SECURE_RSA_Example1
Signed message...SUCCESS!
Verified message is correctly signed...SUCCESS!

C:> _
```

If you tamper with the signature or the message, the signature is detected as invalid. For instance, altering the program like this...

```
SigLen = SECURE_RSA_Sign(&_SECURE_RSA_PrivateKey_Expert,
                         0, 0,
                         &_aMessage[0], sizeof(_aMessage),
                         &_aSignature[0], sizeof(_aSignature));
if (SigLen > 0) {
  printf("Signed message...SUCCESS!\n");
  _aMessage[0]++;
  Status = SECURE_RSA_Verify(&_SECURE_RSA_PublicKey_Expert,
```


…causes verification to fail:

```
C:> SECURE_RSA_Example1
Signed message...SUCCESS!
Correctly signed message did not verify...ERROR!

C:> _
```

4.4 Incremental sign and verify

The previous example shows how to sign and verify a complete message. However, it may well be that the message to sign or verify is not able to entirely fit into the microcontroller's memory. If this is the case, emSecure-RSA offers the ability to compute the message signature and verify that signature incrementally.

Incrementally signing and verifying a message is very straightforward:

- Initialize a hash context using `SECURE_RSA_HASH_Init()`.
- Repeatedly add the message content to sign or verify to the hash context using `SECURE_RSA_HASH_Add`.
- Finally, sign or verify the message using `SECURE_RSA_HASH_Sign()` or `SECURE_RSA_HASH_Verify()`.

The following example shows how to do this:

```
/*********************************************************************
*                   (c) SEGGER Microcontroller GmbH                  *
*                        The Embedded Experts                        *
*                           www.segger.com                           *
**********************************************************************

-------------------------- END-OF-HEADER -----------------------------

File        : CRYPTO_RSA_Example2.c
Purpose     : Incrementally sign and verify a message.

*/

/*********************************************************************
*
*       #include section
*
**********************************************************************
*/

#include "SECURE_RSA.h"
#include "SECURE_RSA_PrivateKey_Expert.h"
#include "SECURE_RSA_PublicKey_Expert.h"
#include <stdio.h>

/*********************************************************************
*
*       Static const data
*
**********************************************************************
*/

static const U8 _aMessagePart1[] = { "This is a message, " };
static const U8 _aMessagePart2[] = { "sign me." };

/*********************************************************************
*
*       Static data
*
**********************************************************************
*/

static U8 _aSignature[SECURE_RSA_MAX_KEY_LENGTH / 8];

/*********************************************************************
*
*       Static code
*
**********************************************************************
*/
```



```
static int _Sign(void) {
  SECURE_RSA_HASH_CONTEXT Context;
  //
  // Sign message incrementally.
  //
  SECURE_RSA_HASH_Init(&Context);
  SECURE_RSA_HASH_Add (&Context, &_aMessagePart1[0], sizeof(_aMessagePart1));
  SECURE_RSA_HASH_Add (&Context, &_aMessagePart2[0], sizeof(_aMessagePart2));
  //
  return SECURE_RSA_HASH_Sign(&Context,
                              &_SECURE_RSA_PrivateKey_Expert,
                              NULL, 0,
                              &_aSignature[0], sizeof(_aSignature));
}

static int _Verify(int SigLen) {
  SECURE_RSA_HASH_CONTEXT Context;
  //
  // Verify message incrementally.
  //
  SECURE_RSA_HASH_Init(&Context);
  SECURE_RSA_HASH_Add (&Context, &_aMessagePart1[0], sizeof(_aMessagePart1));
  SECURE_RSA_HASH_Add (&Context, &_aMessagePart2[0], sizeof(_aMessagePart2));
  //
  return SECURE_RSA_HASH_Verify(&Context,
                                &_SECURE_RSA_PublicKey_Expert,
                                NULL, 0,
                                &_aSignature[0], SigLen);
}

/*********************************************************************
*
*       Public code
*
**********************************************************************
*/

int main(void) {
  int SigLen;
  //
  SigLen = _Sign();
  if (SigLen > 0) {
    printf("Signed message...SUCCESS!\n");
    if (_Verify(SigLen) > 0) {
      printf("Verified message is correctly signed...SUCCESS!\n");
    } else {
      printf("Correctly signed message did not verify...ERROR!\n");
    }
  } else {
    printf("Failed to sign message...ERROR!\n");
  }
  return 0;
}

/*************************** End of file ****************************/
```

Chapter 5

Using OpenSSL with emSecure-RSA

emSecure-RSA will accept RSA public and private keys that are stored in *Privacy Enhanced Mail* (PEM) format and validate RSA signatures in PKCS#1 v1.5 format. This section describes how to combine emSecure-RSA and OpenSSL into a sign and verify workflow.

5.1 What to expect with OpenSSL

emSecure-RSA supports two signature schemes, RSASSA-PSS and RSASSA-PKCS1-v1.5. The most recent releases of OpenSSL that ship with Mac OS X and most Linux hosts only support older PKCS #1 signing. Therefore, to use signatures that are compatible with both emSecure-RSA and OpenSSL, you must select the RSASSA-PKCS1-v1.5 signature scheme if using the included applications, and you must select that signature scheme when configuring emSecure-RSA on the target equipment.

5.2 Using OpenSSL to generate keys

emSecure-RSA will accept *unencrypted* PEM private keys to sign and PEM public keys to verify.

The following command generates a PEM-formatted RSA keypair whose modulus size is 2048 bits:

```
$ openssl genrsa -out private.pem 2048
Generating RSA private key, 2048 bit long modulus
..+++
..........+++
e is 65537 (0x10001)
$ _
```

The private key file contains both private and public key parts. The following command extracts the public key used for verification of signatures, eliminating the private key information:

```
$ openssl rsa -in private.pem -out public.pem -outform PEM -pubout
writing RSA key
$ _
```

These keys are automatically detected as PEM format when provided to emSecure-RSA utilities.

5.3 Signing and verifying using PEM keys

Signing

emSecure-RSA probes the provided key file to determine its format, which can be a native emSecure-RSA key file or a PEM-formatted key file generated, for instance, by OpenSSL. Because the aim of this section is to demonstrate how to use emSecure-RSA with OpenSSL, we must be mindful to select the RSASSA-PKCS1-v1.5 scheme:

```
C:> echo "Hello, world" >test.txt
C:> emSignRSA -k private.pem -pkcs test.txt

(c) 2014-2018 SEGGER Microcontroller GmbH & Co. KG    www.segger.com
emSecure-RSA Sign V2.38 compiled May 17 2017 10:43:55

Loading private key from private.pem
  Probing file: Key file accepted
  Modulus length is 2048 bits
Loading content from test.txt
  Loaded content is 17 bytes
Writing signature file test.txt.sig

C:> _
```

Verifying

emSecure-RSA probes the provided key file to determine its format, which can be a native emSecure-RSA key file or a PEM-formatted key file. When verifying, the signature will only correctly verify if the same signature scheme is used when signing and verifying:

```
C:> emVerifyRSA -k public.pem -pkcs test.txt

(c) 2014-2018 SEGGER Microcontroller GmbH & Co. KG    www.segger.com
emSecure-RSA Verify V2.38 compiled May 17 2017 10:43:55

Loading public key from public.pem
 Probing file: Key file accepted
  Modulus length is 2048 bits
Loading signature from test.txt.sig
Loading content from test.txt
  Loaded content is 17 bytes
Signature OK.

C:> _
```


5.4 Replicating the signature process with OpenSSL

The preceding section signed a candidate file with an OpenSSL-generated private key and verified the signature of the file with the OpenSSL-generated public key. We now need to show that we can verify an OpenSSL computed signature with emSecure-RSA on the host and on the target.

5.4.1 Using a modern OpenSSL

We start by computing the message digest of the input file using OpenSSL's `dgst` command:

```
$ openssl dgst -sha1 -binary test.txt >test.sha1
$ _
```

This creates a 20-byte SHA-1 message digest in binary format in `test.sha1`.

```
$ ls -l test1.sha1
-rw-r--r--  1 plc  staff  20  1 Feb 19:57 test.sha1
$ hexdump -C test.sha1
00000000  cc fb dc d2 24 d8 60 a8  d7 b5 af 4f 4f 2f 79 ad  |....$.`....OO/y.|
00000010  c2 0d 1a d9                                      |....|
00000014
$ _
```

If your installation of OpenSSL understands the `pkeyutl` command (e.g. as shipped with Ubuntu Linux 14.04 and later), you can sign the file in RSASSA-PKCS1-v1.5 format with a single command:

```
$ openssl pkeyutl -sign -in test.sha1 -inkey private.pem -out test.txt.sig \
    -pkeyopt digest:sha1
$ _
```

5.4.2 Using an older OpenSSL

If your installation open OpenSSL does not understand the `pkeyutl` command (e.g. as shipped with OS X up to and including El Capitan), some additional steps are needed to manually sign the file.

In this case and before signing, we need to DER-encode the plain digest to conform with RSASSA-PKCS1-v1.5's encoding algorithm, EMSA-PKCS1-v1.5, to form the *encoded message*, EM. For a SHA-1 signature, this means we need to prefix the message with some magic:

```
$ printf "\x30\x21\x30\x09\x06\x05\x2b\x0e\x03\x02\x1a\x05\x00\x04\x14" >prefix
$ cat prefix test.sha1 >test.em
$ hexdump -C test.em
00000000  30 21 30 09 06 05 2b 0e  03 02 1a 05 00 04 14 cc  |0!0...+.........|
00000010  fb dc d2 24 d8 60 a8 d7  b5 af 4f 4f 2f 79 ad c2  |...$.`....OO/y..|
00000020  0d 1a d9                                         |...|
00000023
$ _
```

Now we sign this correctly-formatted data with our private key, applying PKCS #1 padding, using OpenSSL's `rsautl` command:

```
$ openssl rsautl -sign -inkey private.pem -in test.em >test.txt.sig
$ _
```

We now have a binary signature in `test.txt.sig` that we can instruct emSecure-RSA to verify:


```
C:> emVerifyRSA.exe -b -pkcs -k public.pem test.txt

(c) 2014-2018 SEGGER Microcontroller GmbH & Co. KG    www.segger.com
emSecure-RSA Verify V2.38 compiled May 17 2017 10:43:55

Loading public key from public.pem
 Probing file: Key file accepted
  Modulus length is 2048 bits
Loading binary signature from test.txt.sig
Loading content from test.txt
  Loaded content is 17 bytes
Signature OK.

C:> _
```

If you need to use SHA-256, you can wrap the digest according to section 9.2 of PKCS #1 v2.2. In practice, this simply means that you replace the prefix above with the prefix constructed for a SHA-256 message digest algorithm.

Chapter 6

API reference

This section describes the public API for emSecure-RSA. Any functions or data structures that are not described here but are exposed through inclusion of the `SECURE.h` header file must be considered private and subject to change.

6.1 Preprocessor symbols

6.1.1 Version number

Description

Symbol expands to a number that identifies the specific emSecure-RSA release.

Definition

```
#define SECURE_RSA_VERSION    24000
```

Symbols

Definition	Description
SECURE_RSA_VERSION	Format is “Mmmrr” so, for example, 23600 corresponds to version 2.38.

6.2 API functions

Function	Description
Initialization	
SECURE_RSA_Init()	Initialize emSecure-RSA.
Sign and verify an entire message	
SECURE_RSA_Sign()	Signs message.
SECURE_RSA_SignEx()	Signs message, external allocation.
SECURE_RSA_Verify()	Verify message.
SECURE_RSA_VerifyEx()	Verify message, external allocation.
Sign and verify a precomputed message digest	
SECURE_RSA_SignDigest()	Sign message digest.
SECURE_RSA_VerifyDigest()	Verify message digest.
Incrementally sign and verify a message	
SECURE_RSA_HASH_Init()	Initialize, incremental sign or verify.
SECURE_RSA_HASH_Add()	Add message data, incremental.
SECURE_RSA_HASH_Sign()	Sign message, incremental.
SECURE_RSA_HASH_Verify()	Verify message, incremental.
Key setup functions	
SECURE_RSA_InitPrivateKey()	Initialize private key.
SECURE_RSA_InitPublicKey()	Initialize public key.
Version information	
SECURE_RSA_GetCopyrightText()	Get copyright as printable string.
SECURE_RSA_GetVersionText()	Get version as printable string.

6.2.1 SECURE_RSA_Init()

Description

Initialize emSecure-RSA.

Prototype

```
void SECURE_RSA_Init(void);
```

Additional information

If not already installed as part of CRYPTO initialization, install the software-implemented hash function selected by emSecure-RSA configuration and basic fast modular exponentiation.

6.2.2 SECURE_RSA_Sign()

Description

Signs message.

Prototype

```
int SECURE_RSA_Sign(const SECURE_RSA_PRIVATE_KEY * pPrivate,
                    const U8                     * pSalt,
                          int                      SaltLen,
                    const U8                     * pMessage,
                          int                      MessageLen,
                          U8                     * pSignature,
                          int                      SignatureLen);
```

Parameters

Parameter	Description
pPrivate	Pointer to private key to sign with.
pSalt	Pointer to salt value to embed.
SaltLen	Octet length of the salt.
pMessage	Pointer to message to sign.
MessageLen	Octet length of message to sign.
pSignature	Pointer to object that receives the generated signature.
SignatureLen	Octet length of the signature.

Return value

≤ 0 Signature failure (signature buffer too small).
> 0 Success, number of bytes written to the the signature buffer that constitutes the signature.

Example

```
static const U8 acData[] = {
                             // ... Data here ...
                           };
//
static const U8 acSalt[] = "SEGGER - The embedded Experts";
static       U8 acSig[512];
//
int _Sign(void) {
  int r;
  r = SECURE_RSA_Sign(&GLOBAL_PrivateKey,
                      acSalt, 30,
                      acData, sizeof(acData),
                      acSig,  sizeof(acSig));
  return r;
}
```


6.2.3 SECURE_RSA_SignEx()

Description

Signs message, external allocation.

Prototype

```
int SECURE_RSA_SignEx(const SECURE_RSA_PRIVATE_KEY * pPrivate,
                      const U8                     * pSalt,
                            int                      SaltLen,
                      const U8                     * pMessage,
                            int                      MessageLen,
                            U8                     * pSignature,
                            int                      SignatureLen,
                            SEGGER_MEM_CONTEXT     * pMem);
```

Parameters

Parameter	Description
pPrivate	Pointer to private key to sign with.
pSalt	Pointer to salt value to embed.
SaltLen	Octet length of the salt.
pMessage	Pointer to message to sign.
MessageLen	Octet length of message to sign.
pSignature	Pointer to object that receives the generated signature.
SignatureLen	Octet length of the signature.
pMem	Allocator to use for temporary storage.

Return value

≤ 0 Signature failure (signature buffer too small).
> 0 Success, number of bytes written to the the signature buffer that constitutes the signature.

Additional information

This is functionally identical to `SECURE_RSA_Sign()` but rather than using stack-allocated temporary storage, storage is provided by the initialized memory context.

6.2.4 SECURE_RSA_Verify()

Description

Verify message.

Prototype

```
int SECURE_RSA_Verify(const SECURE_RSA_PUBLIC_KEY * pPublic,
                            U8                    * pSalt,
                            int                     SaltLen,
                      const U8                    * pMessage,
                            int                     MessageLen,
                      const U8                    * pSignature,
                            int                     SignatureLen);
```

Parameters

Parameter	Description
pPublic	Public key used to verify the message.
pSalt	Pointer to buffer that receives the recovered salt. If pSalt is null, the salt is not recovered, but a correct SaltLen must still be provided.
SaltLen	Octet length of the original salt.
pMessage	Pointer to message to verify.
MessageLen	Octet length of message.
pSignature	Pointer to signature to verify.
SignatureLen	Octet length of the signature.

Return value

≤ 0	Verification failure (incorrect structure of encoded message, signature does not match).
> 0	Correct verification of the signature.

Example

```
static const U8 acData[] = {
                            // ... Data here ...
                           };
static const U8 acSig[]  = {
                            // ... Signature here ...
                           };
static       U8 acSalt[256];
//
int _Verify(void) {
  int r;
  r = SECURE_RSA_Verify(&GLOBAL_PublicKey,
                        acSalt, 30,
                        acData, sizeof(acData),
                        acSig, sizeof(acSig));
  return r;
}
```


6.2.5 SECURE_RSA_VerifyEx()

Description

Verify message, external allocation.

Prototype

```
int SECURE_RSA_VerifyEx(const SECURE_RSA_PUBLIC_KEY * pPublic,
                              U8                    * pSalt,
                              int                     SaltLen,
                        const U8                    * pMessage,
                              int                     MessageLen,
                        const U8                    * pSignature,
                              int                     SignatureLen,
                              SEGGER_MEM_CONTEXT    * pMem);
```

Parameters

Parameter	Description
pPublic	Pointer to public key used to verify the message.
pSalt	Pointer to object that receives the recovered salt. If pSalt is null, the salt is not recovered, but SaltByteCnt must still be given.
SaltLen	Octet length of the original salt.
pMessage	Pointer to message to verify.
MessageLen	Octet length of the message.
pSignature	Pointer to signature to verify.
SignatureLen	Octet length of the signature.
pMem	Allocator to use for temporary storage.

Return value

≤ 0 Verification failure (incorrect structure of encoded message, signature does not match).
> 0 Correct verification of the signature.

Additional information

This is functionally identical to `SECURE_RSA_Sign()` but rather than using stack-allocated temporary storage, storage is provided by the initialized memory context.

6.2.6 SECURE_RSA_HASH_Add()

Description

Add message data, incremental.

Prototype

```
void SECURE_RSA_HASH_Add(      SECURE_RSA_HASH_CONTEXT * pHash,
                         const void                    * pData,
                               unsigned                  DataLen);
```

Parameters

Parameter	Description
pHash	Pointer to hash context.
pData	Pointer to message fragment to add to hash.
DataLen	Octet length of the message fragment.

See also

Incremental sign and verify on page 31

6.2.7 SECURE_RSA_HASH_Init()

Description

Initialize, incremental sign or verify.

Prototype

```
void SECURE_RSA_HASH_Init(SECURE_RSA_HASH_CONTEXT * pHash);
```

Parameters

Parameter	Description
pHash	Hash context covering message.

See also

Incremental sign and verify on page 31

6.2.8 SECURE_RSA_HASH_Sign()

Description

Sign message, incremental.

Prototype

```
int SECURE_RSA_HASH_Sign(      SECURE_RSA_HASH_CONTEXT * pHash,
                         const SECURE_RSA_PRIVATE_KEY  * pPrivate,
                         const U8                      * pSalt,
                               int                       SaltLen,
                               U8                      * pSignature,
                               int                       SignatureLen);
```

Parameters

Parameter	Description
pHash	Hash context covering message.
pPrivate	Pointer to private key to sign with.
pSalt	Pointer to salt octet string.
SaltLen	Octet length of the salt.
pSignature	Pointer to object that receives the generated signature.
SignatureLen	Octet length of the signature.

Return value

≤ 0 Signature failure (signature buffer too small).
> 0 Success, number of bytes written to the the signature buffer that constitute the signature.

Additional information

The signature object have a capacity of at least the octet length of the modulus. Hence, for a 1024-bit prime, the signature object must be at lease 128 bytes in length.

See also

Incremental sign and verify on page 31

6.2.9 SECURE_RSA_HASH_Verify()

Description

Verify message, incremental.

Prototype

```
int SECURE_RSA_HASH_Verify(      SECURE_RSA_HASH_CONTEXT * pHash,
                           const SECURE_RSA_PUBLIC_KEY   * pPublic,
                                 U8                      * pSalt,
                                 int                       SaltLen,
                           const U8                      * pSignature,
                                 int                       SignatureLen);
```

Parameters

Parameter	Description
pHash	Hash context covering message.
pPublic	Public key used to verify the message.
pSalt	Pointer to buffer for recovered salt. If pSalt is null, the salt is not recovered, but SaltLen must still be given.
SaltLen	Length of the original salt.
pSignature	Pointer to signature to verify.
SignatureLen	Octet length of the signature.

Return value

≤ 0 Verification failure (incorrect structure of encoded message, signature does not match).
> 0 Correct verification of the signature.

See also

Incremental sign and verify on page 31

6.2.10 SECURE_RSA_SignDigest()

Description

Sign message digest.

Prototype

```
int SECURE_RSA_SignDigest(const SECURE_RSA_PRIVATE_KEY * pPrivate,
                          const U8                     * pSalt,
                                int                      SaltLen,
                          const U8                     * pDigest,
                                U8                     * pSignature,
                                int                      SignatureLen);
```

Parameters

Parameter	Description
pPrivate	Pointer to private key to sign with.
pSalt	Pointer to salt value to embed.
SaltLen	Octet length of the salt.
pDigest	Pointer to octet string that contains the hash of the message to sign.
pSignature	Pointer to object that receives the generated signature.
SignatureLen	Octet length of the signature.

Return value

≤ 0	Signature failure (signature buffer too small).
> 0	Success, number of bytes written to the the signature buffer that constitute the signature.

6.2.11 SECURE_RSA_VerifyDigest()

Description

Verify message digest.

Prototype

```
int SECURE_RSA_VerifyDigest(const SECURE_RSA_PUBLIC_KEY * pPublic,
                                  U8                    * pSalt,
                                  int                     SaltLen,
                            const U8                    * pDigest,
                            const U8                    * pSignature,
                                  int                     SignatureLen);
```

Parameters

Parameter	Description
pPublic	Public key used to verify the message.
pSalt	Pointer to buffer for recovered salt. If pSalt is null, the salt is not recovered, but SaltLen must still be given.
SaltLen	Octet length of the original salt.
pDigest	Hash of original message.
pSignature	Pointer to signature to verify.
SignatureLen	Octet length of the signature.

Return value

≤ 0 Verification failure (incorrect structure of encoded message, signature does not match).
> 0 Correct verification of the signature.

6.2.12 SECURE_RSA_InitPrivateKey()

Description

Initialize private key.

Prototype

```
void SECURE_RSA_InitPrivateKey(      SECURE_RSA_PRIVATE_KEY   * pPrivate,
                               const SECURE_RSA_KEY_PARAMETER * pParamDP,
                               const SECURE_RSA_KEY_PARAMETER * pParamDQ,
                               const SECURE_RSA_KEY_PARAMETER * pParamP,
                               const SECURE_RSA_KEY_PARAMETER * pParamQ,
                               const SECURE_RSA_KEY_PARAMETER * pParamQInv);
```

Parameters

Parameter	Description
pPrivate	Pointer to private key to be initialized.
pParamDP	Pointer to public key parameter DP.
pParamDQ	Pointer to public key parameter DQ.
pParamP	Pointer to public key parameter P.
pParamQ	Pointer to public key parameter Q.
pParamQInv	Pointer to public key parameter QInv.

6.2.13 SECURE_RSA_InitPublicKey()

Description

Initialize public key.

Prototype

```
void SECURE_RSA_InitPublicKey(      SECURE_RSA_PUBLIC_KEY    * pPublic,
                              const SECURE_RSA_KEY_PARAMETER * pParamE,
                              const SECURE_RSA_KEY_PARAMETER * pParamN);
```

Parameters

Parameter	Description
pPublic	Pointer to public key to be initialized.
pParamE	Pointer to public key parameter E.
pParamN	Pointer to public key parameter N.

6.2.14 SECURE_RSA_GetCopyrightText()

Description

Get copyright as printable string.

Prototype

```
char *SECURE_RSA_GetCopyrightText(void);
```

Return value

Zero-terminated copyright string.

6.2.15 SECURE_RSA_GetVersionText()

Description

Get version as printable string.

Prototype

```
char *SECURE_RSA_GetVersionText(void);
```

Return value

Zero-terminated version string.

Chapter 7

Configuring emSecure-RSA

emSecure-RSA can be configured using preprocessor symbols. All compile-time configuration symbols are preconfigured with valid values which match the requirements of most applications.

The cryptography modules (prefixed `CRYPTO_`), and SEGGER modules (prefixed `SEGGER_`) are shared with other SEGGER products, for instance emSSL, and should be configured to match all modules which are used in the same application.

7.1 Algorithm parameters

7.1.1 Key length

Default

```
#define SECURE_RSA_MAX_KEY_LENGTH     2048
```

Override

To define a non-default value, define this symbol in `SECURE_RSA_Conf.h`.

Description

Configure the maximum length of keys. This preprocessor symbol is used to reserve enough memory when signing or verifying a message.

7.1.2 Hash function

Default

```
#define SECURE_HASH_FUNCTION     SHA1
```

Override

To define a non-default value, define this symbol in `SECURE_RSA_Conf.h`.

Description

Configure the hash function to use. This preprocessor symbol `SECURE_HASH_FUNCTION`, if defined, must be set to one of the following:

Value	Description
`SHA1`	Use SHA-1 with 20-byte digest as the underlying hash function.
`SHA256`	Use SHA-256 with a 32-byte digest as the underlying hash function.
`SHA512`	Use SHA-512 with a 64-byte digest as the underlying hash function.

7.1.3 Signature scheme

Default

```
#define SECURE_SIGNATURE_SCHEME     PSS
```

Override

To define a non-default value, define this symbol in `SECURE_RSA_Conf.h`.

Description

Configure the signature scheme use. This preprocessor symbol `SECURE_SIGNATURE_SCHEME`, if defined, must be set to one of the following:

Value	Description
`PSS`	Use RSASSA-PSS as the signature scheme.
`PKCS1`	Use RSASSA-PKCS1-v1.5 as the signature scheme.

7.2 Crytographic components

The following definitions must be set to configure the underlying cryptographic algorithm library, emCrypt.

The relevant sections of the emCrypt documentation are included here.

7.2.1 Multiprecision integers

Default

```
#define CRYPTO_MPI_BITS_PER_LIMB     32
```

Override

To define a non-default value, define this symbol in `CRYPTO_Conf.h`.

Description

This preprocessor symbol configures the number of bits per limb for multiprecision integer algorithms. The default of 32 matches 32-bit targets well, such as ARM and PIC32. In general, it is best to set the number of bits per limb to the number of bits in the standard `int` or `unsigned` type used by the target compiler.

Supported configurations are:

- 32 — requires the target compiler to support 64-bit types natively (i.e. `unsigned long long` or `unsigned __int64`),
- 16 — which should run on any ISO compiler whose native integer types are 16 or 32 bit and supports 32-bit `unsigned long`.
- 8 — 8-bit limb sizes are supported and selecting this size may well lead to better multiplication performance on 8-bit architectures.

7.2.2 Hash algorithms

The following definitions can be set for the hash components in order to balance code size and performance.

7.2.2.1 SHA-1

7.2.2.2 SHA-256

7.2.2.3 SHA-512

Chapter 8

Performance and resource use

This chapter describes the memory requirements and performance of emSecure-RSA.

8.1 Performance

RSA sign and verify performance can be benchmarked with the application SECURE_RSA_Bench_Performance.c.

The following is the output for the Cortex-M4 on a SEGGER emPower board:

```
(c) 2014-2018 SEGGER Microcontroller GmbH & Co. KG    www.segger.com
emSecure-RSA Performance Benchmark compiled May 19 2017 09:24:12

Compiler: clang 4.0.0 (tags/RELEASE_400/final)
System:   Processor speed           = 168.000 MHz
Config:   CRYPTO_VERSION            = 20000 [2.00]
Config:   SECURE_RSA_VERSION        = 23000 [2.30]
Config:   CRYPTO_MPI_BITS_PER_LIMB  = 32
Config:   SECURE_RSA_MAX_KEY_LENGTH = 2048 bits

Sign/Verify Performance
=======================

+----------+----------+----------+----------+
|  Modulus |  Message |     Sign |   Verify |
|    /bits |   /bytes |      /ms |      /ms |
+----------+----------+----------+----------+
|      512 |        0 |    32.03 |     1.95 |
|      512 |     1024 |    32.19 |     2.22 |
|      512 |   102400 |    55.16 |    25.22 |
+----------+----------+----------+----------+
|     1024 |        0 |   146.71 |     5.05 |
|     1024 |     1024 |   147.00 |     5.32 |
|     1024 |   102400 |   169.83 |    28.33 |
+----------+----------+----------+----------+
|     2048 |        0 |   847.00 |    17.61 |
|     2048 |     1024 |   846.00 |    17.82 |
|     2048 |   102400 |   869.00 |    40.76 |
+----------+----------+----------+----------+

Benchmark complete
```


8.1.1 SECURE_RSA_Bench_Performance.c listing

```
/*********************************************************************
*                   (c) SEGGER Microcontroller GmbH                  *
*                        The Embedded Experts                        *
*                           www.segger.com                           *
**********************************************************************

-------------------------- END-OF-HEADER -----------------------------

File        : SECURE_RSA_Bench_Performance.c
Purpose     : Benchmark emSecure-RSA performance.

*/

/*********************************************************************
*
*       #include section
*
**********************************************************************
*/

#include "SECURE_RSA.h"
#include "SECURE_RSA_PrivateKey_512b.h"
#include "SECURE_RSA_PrivateKey_1024b.h"
#include "SECURE_RSA_PrivateKey_2048b.h"
#include "SECURE_RSA_PublicKey_512b.h"
#include "SECURE_RSA_PublicKey_1024b.h"
#include "SECURE_RSA_PublicKey_2048b.h"
#include "SEGGER_SYS.h"

/*********************************************************************
*
*       Defines, fixed
*
**********************************************************************
*/

#define STRINGIZE(X)   #X
#define STRINGIZEX(X)  STRINGIZE(X)

/*********************************************************************
*
*       Local data types
*
**********************************************************************
*/

typedef struct {
  const CRYPTO_RSA_PRIVATE_KEY * pPrivateKey;
  const CRYPTO_RSA_PUBLIC_KEY  * pPublicKey;
  const U8                     * pMesssage;
  unsigned                       MessageLen;
} BENCH_PARA;

/*********************************************************************
*
*       Static const data
*
**********************************************************************
*/

static const U8 _aMessage_100k[100*1024] = {
  0x00,
};

static const BENCH_PARA _aBenchKeys[] = {
  { &_SECURE_RSA_PrivateKey_512b,  &_SECURE_RSA_PublicKey_512b,  _aMessage_100k,        0u },
  { &_SECURE_RSA_PrivateKey_512b,  &_SECURE_RSA_PublicKey_512b,  _aMessage_100k,     1024u },
  { &_SECURE_RSA_PrivateKey_512b,  &_SECURE_RSA_PublicKey_512b,  _aMessage_100k, 100*1024u },
  { NULL,                          NULL,                         NULL,                  0u },
  { &_SECURE_RSA_PrivateKey_1024b, &_SECURE_RSA_PublicKey_1024b, _aMessage_100k,        0u },
  { &_SECURE_RSA_PrivateKey_1024b, &_SECURE_RSA_PublicKey_1024b, _aMessage_100k,     1024u },
  { &_SECURE_RSA_PrivateKey_1024b, &_SECURE_RSA_PublicKey_1024b, _aMessage_100k, 100*1024u },
  { NULL,                          NULL,                         NULL,                  0u },
  { &_SECURE_RSA_PrivateKey_2048b, &_SECURE_RSA_PublicKey_2048b, _aMessage_100k,        0u },
  { &_SECURE_RSA_PrivateKey_2048b, &_SECURE_RSA_PublicKey_2048b, _aMessage_100k,     1024u },
  { &_SECURE_RSA_PrivateKey_2048b, &_SECURE_RSA_PublicKey_2048b, _aMessage_100k, 100*1024u }
};

/*********************************************************************
*
*       Static data
*
**********************************************************************
```

```
*/

static union {
  //
  // Temporary memory used for signing
  //
  CRYPTO_MPI_LIMB       aWs[5][CRYPTO_MPI_LIMBS_REQUIRED(SECURE_RSA_MAX_KEY_LENGTH) + 2];
  //
  // Temporary memory used for verifing
  //
  CRYPTO_MPI_LIMB       aWv[4][(CRYPTO_MPI_LIMBS_REQUIRED(SECURE_RSA_MAX_KEY_LENGTH*2) + 2)];
} _Workspace;

/*********************************************************************
*
*       Static code
*
**********************************************************************
*/

/*********************************************************************
*
*       _ConvertTicksToSeconds()
*
*  Function description
*    Convert ticks to seconds.
*
*  Parameters
*    Ticks - Number of ticks reported by SEGGER_SYS_OS_GetTimer().
*
*  Return value
*    Number of seconds corresponding to tick.
*/
static float _ConvertTicksToSeconds(U64 Ticks) {
  return SEGGER_SYS_OS_ConvertTicksToMicros(Ticks) / 1000000.0f;
}

/*********************************************************************
*
*       _BenchmarkSignVerify()
*
*  Function description
*    Count the number of signs and verifies completed in one second.
*
*  Parameters
*    pPara - Pointer to benchmark parameters.
*/
static void _BenchmarkSignVerify(const BENCH_PARA *pPara) {
  U8    aSignature[270];
  U64   OneSecond;
  U64   T0;
  U64   Elapsed;
  int   SignatureLen;
  int   Loops;
  int   Status;
  float Time;
  SEGGER_MEM_CHUNK_HEAP Heap;
  SEGGER_MEM_CONTEXT    Context;
  //
  SEGGER_SYS_IO_Printf("| %8d | %8u | ",
                       CRYPTO_MPI_BitCount(&pPara->pPublicKey->N),
                       pPara->MessageLen);
  //
  Loops = 0;
  OneSecond = SEGGER_SYS_OS_ConvertMicrosToTicks(1000000);
  T0 = SEGGER_SYS_OS_GetTimer();
  do {

  SEGGER_MEM_CHUNK_HEAP_Init(&Context, &Heap, _Workspace.aWs, SEGGER_COUNTOF(_Workspace.aWs), sizeof(_Workspace.aWs[0]));
    SignatureLen = SECURE_RSA_SignEx(pPara->pPrivateKey,
                                     NULL,            0,
                                     pPara->pMesssage, pPara->MessageLen,
                                     aSignature,       sizeof(aSignature),
                                     &Context);
    Elapsed = SEGGER_SYS_OS_GetTimer() - T0;
    ++Loops;
  } while (SignatureLen >= 0 && Elapsed < OneSecond);
  //
  Time = 1000.0f * _ConvertTicksToSeconds(Elapsed) / Loops;
  if (SignatureLen < 0) {
    SEGGER_SYS_IO_Printf("%8s | ", "-Fail-");
  } else {
    SEGGER_SYS_IO_Printf("%8.2f | ", Time);
  }
  //
  if (SignatureLen < 0) {
    SEGGER_SYS_IO_Printf("%8s |\n", "-Skip-");
```

```
  } else {
    Loops = 0;
    OneSecond = SEGGER_SYS_OS_ConvertMicrosToTicks(1000000);
    T0 = SEGGER_SYS_OS_GetTimer();
    do {

  SEGGER_MEM_CHUNK_HEAP_Init(&Context, &Heap, _Workspace.aWv, SEGGER_COUNTOF(_Workspace.aWv), sizeof(_Workspace.aWv[0]));
      Status = SECURE_RSA_VerifyEx(pPara->pPublicKey,
                                   NULL,              0,
                                   pPara->pMesssage, pPara->MessageLen,
                                   aSignature,       SignatureLen,
                                   &Context);
      Elapsed = SEGGER_SYS_OS_GetTimer() - T0;
      ++Loops;
    } while (Status >= 0 && Elapsed < OneSecond);
    //
    Time = 1000.0f * _ConvertTicksToSeconds(Elapsed) / Loops;
    if (Status <= 0) {
      SEGGER_SYS_IO_Printf("%8s |\n", "-Fail-");
    } else {
      SEGGER_SYS_IO_Printf("%8.2f |\n", Time);
    }
  }
}

/*********************************************************************
*
*       Public code
*
**********************************************************************
*/

/*********************************************************************
*
*       MainTask()
*
*  Function description
*    Main entry point for application to run all the tests.
*/
void MainTask(void);
void MainTask(void) {
  unsigned i;
  //
  SECURE_RSA_Init();
  SEGGER_SYS_Init();
  //
  SEGGER_SYS_IO_Printf("\n");
  SEGGER_SYS_IO_Printf("%s    www.segger.com\n", SECURE_RSA_GetCopyrightText());
  SEGGER_SYS_IO_Printf("emSecure-RSA Performance Benchmark compiled " __DATE__ " " __TIME__ "\n\n");
  //
  SEGGER_SYS_IO_Printf("Compiler: %s\n", SEGGER_SYS_GetCompiler());
  if (SEGGER_SYS_GetProcessorSpeed() > 0) {
    SEGGER_SYS_IO_Printf("System:   Processor speed             = %.3f MHz\n",
                         (float)SEGGER_SYS_GetProcessorSpeed() / 1000000.0f);
  }
  SEGGER_SYS_IO_Printf("Config:   CRYPTO_VERSION              = %u
 [%s]\n", CRYPTO_VERSION, CRYPTO_GetVersionText());
  SEGGER_SYS_IO_Printf("Config:   SECURE_RSA_VERSION          = %u
 [%s]\n", SECURE_RSA_VERSION, SECURE_RSA_GetVersionText());
  SEGGER_SYS_IO_Printf("Config:   CRYPTO_MPI_BITS_PER_LIMB    = %u\n",      CRYPTO_MPI_BITS_PER_LIMB);
  SEGGER_SYS_IO_Printf("Config:   SECURE_RSA_MAX_KEY_LENGTH   = %u bits\n", SECURE_RSA_MAX_KEY_LENGTH);
  SEGGER_SYS_IO_Printf("Config:   SECURE_RSA_HASH_FUNCTION    = %s\n",
  STRINGIZEX(SECURE_RSA_HASH_FUNCTION));
  SEGGER_SYS_IO_Printf("Config:   SECURE_RSA_SIGNATURE_SCHEME = %s\n",
  STRINGIZEX(SECURE_RSA_SIGNATURE_SCHEME));
  SEGGER_SYS_IO_Printf("\n");
  //
  SEGGER_SYS_IO_Printf("Sign/Verify Performance\n");
  SEGGER_SYS_IO_Printf("=======================\n\n");
  //
  SEGGER_SYS_IO_Printf("+----------+----------+----------+----------+\n");
  SEGGER_SYS_IO_Printf("|  Modulus |  Message |     Sign |   Verify |\n");
  SEGGER_SYS_IO_Printf("|    /bits |    /bytes |      /ms |      /ms |\n");
  SEGGER_SYS_IO_Printf("+----------+----------+----------+----------+\n");
  for (i = 0; i < SEGGER_COUNTOF(_aBenchKeys); ++i) {
    if (_aBenchKeys[i].pPublicKey == NULL) {
      SEGGER_SYS_IO_Printf("+----------+----------+----------+----------+\n");
    } else {
      _BenchmarkSignVerify(&_aBenchKeys[i]);
    }
  }
  SEGGER_SYS_IO_Printf("+----------+----------+----------+----------+\n");
  SEGGER_SYS_IO_Printf("\n");
  //
  SEGGER_SYS_IO_Printf("Benchmark complete\n");
  SEGGER_SYS_OS_PauseBeforeHalt();
  SEGGER_SYS_OS_Halt(0);
```



```
}
/*************************** End of file ****************************/
```


8.2 Memory footprint

The following table lists the memory requirements of emSecure-RSA configured for operation with a 2048 bit key. There are two components to the ROM use, one for the emSecure-RSA code and one for the emCrypt component that can be shared with other security products such as emSecure-ECDSA, emSSL, and emSSH.

Process	ROM (SECURE)	ROM (CRYPTO)	Total ROM	Static RAM	Stack
Sign only	0.21 KB	5.42 KB	5.63 KB	0.01 KB	2.12 KB
Verify only	0.21 KB	4.35 KB	4.56 KB	0.01 KB	2.93 KB
Sign and verify	0.34 KB	5.70 KB	6.04 KB	0.01 KB	2.93 KB

Test configuration:

- SEGGER Embedded Studio 3.20
- Cortex-M4 microcontroller running at 168 MHz (SEGGER emPower).

Chapter 9

Frequently asked questions

Q: *I want to prevent copying a whole firmware from one product hardware to another cloned one. How can I prevent it to be run from the cloned version with emSecure?*

A: Nearly every modern MCU includes a unique ID, which is different on every device. When the signature covers this UID it is only valid on one single device and cannot be run on a cloned or copied product. The firmware can verify the signature at boot-time.

Q: *I added a digital signature to my product. Where should I verify it?*

A: Signature verification can be done in-product or off-product. With in-product verification the firmware for example verifies the digital signature at boot-time and refuses to run when the signature cannot be verified. With off-product verification an external application, e.g. a PC application communicating with the device, reads the signature and data from the product and verifies it.

Q: *I want my product to only run genuine firmware images. How can I achieve this with emSecure?*

A: To make sure a firmware image is genuine, the complete image can be signed with a digital signature. Like when using a CRC for integrity checks, the signature is sent with the firmware data upon a firmware update. The application or bootloader programming the firmware onto the device validates the firmware data with its signature. The signature can only be generated with the private key and should be provided by the developer with the firmware data.

Q: *I am providing additional licenses for my product which shall be locked to a specific user or computer. Can I generate license keys with emSecure?*

A: Yes. emSecure can generate unique license keys for any data, like a computer ID, a user name, e-mail address or any other data.

Q: *My product is sending data to a computer application. Can I make sure the computer application is getting data only from my product with emSecure?*

A: Yes. In this case the product is used to sign the data and the computer applications verifies it. To prevent the private key from being read from the product it might be stored encrypted on the product or in the application and decrypted prior to signing the data.

Chapter 10

Reference

10.1 Technical background

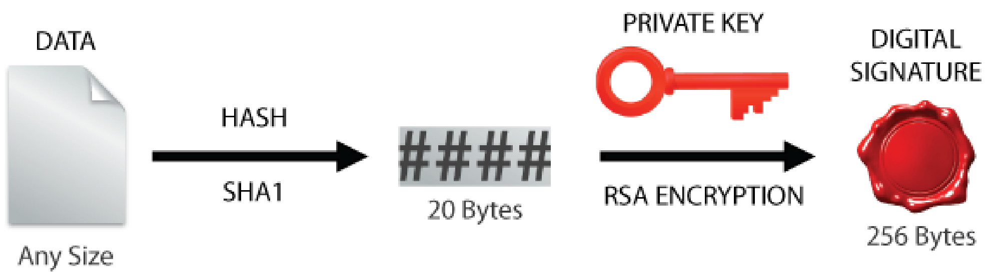
emSecure makes use of the RSA-PSS signature scheme, following the "hash-then-sign" paradigm.

A hash of the data to be signed is generated and transformed to create an encoded message which is as long as the private key modulus length. The encoded message is then RSA encrypted.

On verification the message is decrypted again, the encoded message is checked for consistency and the hash is compared with the hash generated from the data to be verified.

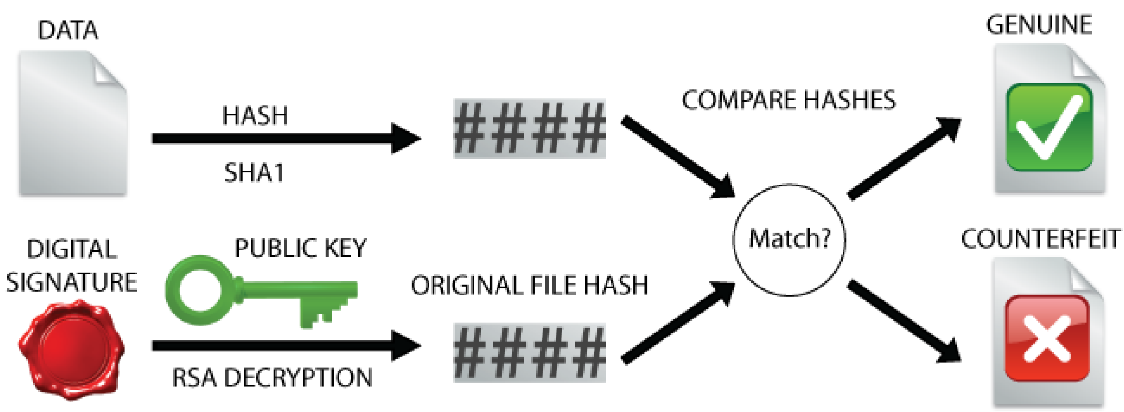
10.1.1 emSecure Signing Technical Details

The emSecure signing operation starts by using a secure hash algorithm (SHA1) to generate a hash from the original data. Then using the 2kbit RSA private key along with the hash a digital signature is generated using RSA encryption.



10.1.2 emSecure Verification Technical Details:

The emSecure verification process starts with the data one wishes to verify and the digital signature which was created from the original file. A hash file is generated for the unverified data. The public key and RSA decryption is used to generate the original hash and then compared to verify whether the data file is genuine.



10.2 RSA

RSA (named after Ron Rivest, Adi Shamir and Leonard Adleman) is the first published asymmetric public-key cryptosystem algorithm.

RSA uses a set of two keys. One key, the public key, for encryption of data and verification of digital signatures, the other key, the private key, for decrypting and digitally signing data.

RSA keys are generated using two large prime numbers (`P` and `Q`). With the knowledge of the public key, consisting of the product of the two prime factors (the modulus, `N`) and an auxiliary value (`E`) it is possible to encrypt a message which can then only be decrypted with the private key, but due to the problem of factoring a number with large enough prime factors, there is no known efficient method to get the private key's value (`D`) or the two primes from the public key, when the prime factors are kept secret.

For emSecure-RSA, the modulus (`N`) can be chosen to be of a width between 512 and 4096 bits. In practice RSA is presumed to be safe if `N` is large enough. In 2010 an RSA modulus of 768 bits has been factorized, there is no known factorization of a higher number.

A modulus of 1024 bits is graded as commercial grade, whereas a 2048 bit modulus is of military grade. Both widths are presumed to be safe for now.

RSA is often used in hybrid cryptosystems because it is known to be 1,000 times slower than symmetric algorithms such as AES. Therefore RSA is used to exchange the encrypted password or initialization vector for symmetric encrypted communication.

Encryption algorithm	Decryption algorithm
`c = m^E mod(N)`	`m = c^D mod(N)`

`m: Message  c: Ciphertext`

`E: Public value  D: Private value  N: Modulus (P*Q)`

10.3 Signature scheme

10.3.1 PKCS #1

PKCS #1, the Public-Key Cryptography Standards, is the first set of standards for cryptography, published by RSA Laboratories. It describes the definitions and recommendations for RSA as well as the properties of public and private keys, primitive operations and secure cryptographic schemes.

RSA-PSS is part of PKCS #1 V2.1.

http://www.ietf.org/rfc/rfc3447.txt

10.3.2 RSA-PSS

RSA-PSS is a signature scheme based on RSA that provides secure digital signatures. PSS refers to the Probabilistic Signature Scheme by Mihir Bellare and Phillip Rogaway. RSA-PSS follows the "hash-then-sign" paradigm. Instead of encrypting a message, the hash value of the message is encrypted.

A signature is generated in three steps:

- Generate a hash `H` from the message `M` to be signed (using an approved hashing algorithm such as SHA-1, SHA-256, SHA-512).
- Transform `H` and the optional salt `salt` to generate the encrypted message `EM` (using the hashing algorithm and a mask generation function).
- Encrypt `EM` with the private key, creating the signature `S` (using RSA).

It is verified in the same way:

- Generate a hash `H`' from the message to be verified.
- Decrypt the signature `S` to recover `EM`.
- Verify `EM` is valid and consistent.
- Verify that `EM` is a valid transformation of `H`'.
- Optionally recover `salt` from `EM`.

10.3.3 SHA-1

SHA, Secure Hash Algorithm, is a set of cryptographic hash functions. The hash functions were designed by the U.S. National Security Agency (NSA) and NIST as a Federal Information Processing Standard (FIPS).

Secure Hash Algorithms are used to create a unique hash value for any message which is used as the base for digital signatures and provides the integrity of a message. This requires the collision resistance of hash values, the should be practically no two different messages with the same hash value.

SHA-1 produces a 160-bit secure hash of a message of up to `2^64 - 1` bits.

10.3.4 MGF

MGF, a Mask Generation Function, creates an output string of a desired length from an input of variable length. The output is completely determined by the input and should be pseudo-random.

MGF1, used in the RSA-PSS signature scheme is a mask generation function based on a hash function.

Chapter 11

Appendix

11.1 Example key pair

The example key pair was generated with emKeyGen, default settings (2048 bit key, proven primes), and the pass phrase “SEGGER - The Embedded Experts”.

```
C:> emKeyGenRSA -pw "SEGGER - The Embedded Experts" -k SECURE_RSA_Expert_Key

(c) 2014-2018 SEGGER Microcontroller GmbH & Co. KG    www.segger.com
emSecure-RSA KeyGen V2.38 compiled May 18 2017 11:51:53

Generating proven prime key pair with public modulus of 2048 bits
Public encryption exponent is set to 65537
Initial seed is 0xADBE961296F573AD2FA65468E1A8837D
Checking keys are consistent: OK
Writing public key file SECURE_RSA_Expert_Key.pub.
Writing private key file SECURE_RSA_Expert_Key.prv.

C:> _
```

The two files that are written contain the public key and the private key, together making a matched key pair. The private key is required when signing some data and must be kept private and secure. The public key is required when verifying some signed data and can be distributed without concern for privacy.

The keys are converted to compilable form using emPrintKey:

```
C:> emPrintKeyRSA SECURE_RSA_Expert_Key.prv -p _SECURE_RSA_PrivateKey_Expert \
      >SECURE_RSA_PrivateKey_Expert.h

(c) 2014-2018 SEGGER Microcontroller GmbH & Co. KG    www.segger.com
emSecure-RSA PrintKey V2.38 compiled May 17 2017 10:43:55

Probing file to determine type of key

C:> emPrintKeyRSA SECURE_RSA_Expert_Key.pub -p _SECURE_RSA_PublicKey_Expert \
      >SECURE_RSA_PublicKey_Expert.h

(c) 2014-2018 SEGGER Microcontroller GmbH & Co. KG    www.segger.com
emSecure-RSA PrintKey V2.38 compiled May 17 2017 10:43:55

Probing file to determine type of key

C:> _
```

The generated source files are shown in the following sections.

11.1.1 SECURE_RSA_PrivateKey_Expert.h listing

```
static const CRYPTO_MPI_LIMB _SECURE_RSA_PrivateKey_Expert_D_aLimbs[] = {
  CRYPTO_MPI_LIMB_DATA4(0xC1, 0x0E, 0x67, 0x6C),
  CRYPTO_MPI_LIMB_DATA4(0xDE, 0xF0, 0x5E, 0x22),
  CRYPTO_MPI_LIMB_DATA4(0x90, 0xBC, 0xCC, 0xA8),
  CRYPTO_MPI_LIMB_DATA4(0x28, 0xA3, 0x0A, 0x05),
  CRYPTO_MPI_LIMB_DATA4(0x5C, 0x2A, 0xB6, 0x7E),
  CRYPTO_MPI_LIMB_DATA4(0xC2, 0xD5, 0xA3, 0xEC),
  CRYPTO_MPI_LIMB_DATA4(0x1B, 0x77, 0x80, 0x55),
  CRYPTO_MPI_LIMB_DATA4(0xAB, 0x78, 0xC6, 0x40),
  CRYPTO_MPI_LIMB_DATA4(0xFA, 0xC3, 0x0B, 0x9A),
  CRYPTO_MPI_LIMB_DATA4(0x66, 0x8B, 0x8A, 0xB2),
  CRYPTO_MPI_LIMB_DATA4(0x0D, 0x88, 0x68, 0x4B),
  CRYPTO_MPI_LIMB_DATA4(0x3A, 0xFE, 0xC0, 0x75),
  CRYPTO_MPI_LIMB_DATA4(0x4B, 0x80, 0x92, 0xF5),
  CRYPTO_MPI_LIMB_DATA4(0x4D, 0x14, 0xA2, 0xC9),
  CRYPTO_MPI_LIMB_DATA4(0x98, 0x50, 0xE0, 0x47),
  CRYPTO_MPI_LIMB_DATA4(0x33, 0x39, 0x79, 0xE3),
  CRYPTO_MPI_LIMB_DATA4(0x7E, 0x67, 0x85, 0x4D),
  CRYPTO_MPI_LIMB_DATA4(0x12, 0xDE, 0xE9, 0xCD),
  CRYPTO_MPI_LIMB_DATA4(0x54, 0xF3, 0x95, 0xB9),
  CRYPTO_MPI_LIMB_DATA4(0xFA, 0xC0, 0x2A, 0x7A),
  CRYPTO_MPI_LIMB_DATA4(0xEC, 0x5B, 0x9F, 0x7E),
  CRYPTO_MPI_LIMB_DATA4(0x35, 0xCA, 0x31, 0xF1),
  CRYPTO_MPI_LIMB_DATA4(0x7F, 0x4A, 0x7E, 0xC3),
  CRYPTO_MPI_LIMB_DATA4(0xD9, 0xFF, 0xC9, 0x1E),
  CRYPTO_MPI_LIMB_DATA4(0xF7, 0x68, 0x5F, 0x06),
  CRYPTO_MPI_LIMB_DATA4(0xCF, 0x2A, 0x1A, 0x25),
  CRYPTO_MPI_LIMB_DATA4(0xF7, 0x22, 0x21, 0x6E),
  CRYPTO_MPI_LIMB_DATA4(0x42, 0xDF, 0xE9, 0xF2),
  CRYPTO_MPI_LIMB_DATA4(0x70, 0x10, 0xC3, 0x33),
  CRYPTO_MPI_LIMB_DATA4(0xB1, 0xAC, 0x2A, 0x0C),
  CRYPTO_MPI_LIMB_DATA4(0x8E, 0x9C, 0x9E, 0x5B),
  CRYPTO_MPI_LIMB_DATA4(0x30, 0xE0, 0x58, 0x17),
  CRYPTO_MPI_LIMB_DATA4(0x6D, 0x51, 0xB9, 0x00),
  CRYPTO_MPI_LIMB_DATA4(0xD9, 0x7E, 0xC9, 0xEE),
  CRYPTO_MPI_LIMB_DATA4(0xAA, 0x4B, 0xCC, 0xD6),
  CRYPTO_MPI_LIMB_DATA4(0x47, 0x6C, 0xDC, 0xBA),
  CRYPTO_MPI_LIMB_DATA4(0x3C, 0x6E, 0x52, 0x88),
  CRYPTO_MPI_LIMB_DATA4(0xF3, 0xC0, 0x76, 0xA8),
  CRYPTO_MPI_LIMB_DATA4(0xE7, 0x23, 0x27, 0x61),
  CRYPTO_MPI_LIMB_DATA4(0xB9, 0x20, 0x33, 0x4B),
  CRYPTO_MPI_LIMB_DATA4(0x08, 0x8D, 0xEF, 0x36),
  CRYPTO_MPI_LIMB_DATA4(0x1C, 0x63, 0x9F, 0x9E),
  CRYPTO_MPI_LIMB_DATA4(0xF5, 0xA2, 0xDF, 0x11),
  CRYPTO_MPI_LIMB_DATA4(0x5D, 0xDC, 0xE1, 0x11),
  CRYPTO_MPI_LIMB_DATA4(0x24, 0xBE, 0x2E, 0xAD),
  CRYPTO_MPI_LIMB_DATA4(0xD3, 0xB3, 0xC1, 0xDA),
  CRYPTO_MPI_LIMB_DATA4(0xF4, 0xC8, 0x22, 0xC0),
  CRYPTO_MPI_LIMB_DATA4(0xE6, 0x17, 0x54, 0xF5),
  CRYPTO_MPI_LIMB_DATA4(0xAD, 0xCF, 0x94, 0xB9),
  CRYPTO_MPI_LIMB_DATA4(0x92, 0xB9, 0xD5, 0x25),
  CRYPTO_MPI_LIMB_DATA4(0x7A, 0xE7, 0xEC, 0x7B),
  CRYPTO_MPI_LIMB_DATA4(0x64, 0xE6, 0xEB, 0x0C),
  CRYPTO_MPI_LIMB_DATA4(0x70, 0xBD, 0x63, 0xA4),
  CRYPTO_MPI_LIMB_DATA4(0x44, 0xA9, 0x49, 0x03),
  CRYPTO_MPI_LIMB_DATA4(0x74, 0x21, 0x0B, 0x79),
  CRYPTO_MPI_LIMB_DATA4(0xED, 0xFA, 0x1A, 0xFC),
  CRYPTO_MPI_LIMB_DATA4(0x38, 0x7F, 0x11, 0x3A),
  CRYPTO_MPI_LIMB_DATA4(0x81, 0xD2, 0x08, 0xFE),
  CRYPTO_MPI_LIMB_DATA4(0x8D, 0xCB, 0x5C, 0x5B),
  CRYPTO_MPI_LIMB_DATA4(0xBF, 0x06, 0x99, 0x0D),
  CRYPTO_MPI_LIMB_DATA4(0xD8, 0xF0, 0xBD, 0xBA),
  CRYPTO_MPI_LIMB_DATA4(0x4D, 0x93, 0x35, 0xF3),
  CRYPTO_MPI_LIMB_DATA4(0xAA, 0x2F, 0xAA, 0x48),
  CRYPTO_MPI_LIMB_DATA4(0xEC, 0x64, 0xDA, 0x10)
```

```
};

static const CRYPTO_MPI_LIMB _SECURE_RSA_PrivateKey_Expert_P_aLimbs[] = {
  CRYPTO_MPI_LIMB_DATA4(0xB7, 0x0E, 0x63, 0x39),
  CRYPTO_MPI_LIMB_DATA4(0x88, 0x90, 0x46, 0xFE),
  CRYPTO_MPI_LIMB_DATA4(0xA7, 0xB4, 0x02, 0x91),
  CRYPTO_MPI_LIMB_DATA4(0xE0, 0x82, 0xD0, 0x45),
  CRYPTO_MPI_LIMB_DATA4(0x6F, 0x66, 0x49, 0x71),
  CRYPTO_MPI_LIMB_DATA4(0x76, 0xE6, 0xC2, 0xEB),
  CRYPTO_MPI_LIMB_DATA4(0x36, 0x00, 0xAF, 0xF8),
  CRYPTO_MPI_LIMB_DATA4(0x91, 0xC7, 0x02, 0x76),
  CRYPTO_MPI_LIMB_DATA4(0xA4, 0x72, 0xBB, 0x3C),
  CRYPTO_MPI_LIMB_DATA4(0xD8, 0xC8, 0x48, 0x9E),
  CRYPTO_MPI_LIMB_DATA4(0x09, 0xF5, 0xD7, 0x8F),
  CRYPTO_MPI_LIMB_DATA4(0xAE, 0x5E, 0x10, 0xE8),
  CRYPTO_MPI_LIMB_DATA4(0xDB, 0x6D, 0xD4, 0x90),
  CRYPTO_MPI_LIMB_DATA4(0xD5, 0x0D, 0x20, 0xAB),
  CRYPTO_MPI_LIMB_DATA4(0x0A, 0xC3, 0x18, 0xB1),
  CRYPTO_MPI_LIMB_DATA4(0xF6, 0x01, 0x67, 0xA6),
  CRYPTO_MPI_LIMB_DATA4(0xE9, 0x90, 0x4E, 0x7F),
  CRYPTO_MPI_LIMB_DATA4(0xFA, 0x2B, 0xA6, 0x87),
  CRYPTO_MPI_LIMB_DATA4(0x26, 0xEF, 0x83, 0xA0),
  CRYPTO_MPI_LIMB_DATA4(0x6B, 0x73, 0xA8, 0xD4),
  CRYPTO_MPI_LIMB_DATA4(0x8F, 0xF8, 0x9F, 0x78),
  CRYPTO_MPI_LIMB_DATA4(0x20, 0xEE, 0x7A, 0x5C),
  CRYPTO_MPI_LIMB_DATA4(0x69, 0x53, 0xA2, 0xDE),
  CRYPTO_MPI_LIMB_DATA4(0x66, 0xAE, 0x8C, 0x07),
  CRYPTO_MPI_LIMB_DATA4(0x1A, 0x15, 0x97, 0x81),
  CRYPTO_MPI_LIMB_DATA4(0xE2, 0x1D, 0x42, 0x41),
  CRYPTO_MPI_LIMB_DATA4(0xE9, 0xD2, 0x37, 0x0B),
  CRYPTO_MPI_LIMB_DATA4(0x8A, 0xD7, 0xC1, 0x5A),
  CRYPTO_MPI_LIMB_DATA4(0x6C, 0x51, 0xC6, 0xBF),
  CRYPTO_MPI_LIMB_DATA4(0x87, 0xAB, 0x25, 0xEB),
  CRYPTO_MPI_LIMB_DATA4(0x62, 0x57, 0x56, 0xC4),
  CRYPTO_MPI_LIMB_DATA4(0x22, 0xA8, 0x49, 0xD2)
};

static const CRYPTO_MPI_LIMB _SECURE_RSA_PrivateKey_Expert_Q_aLimbs[] = {
  CRYPTO_MPI_LIMB_DATA4(0x89, 0x0C, 0x90, 0x27),
  CRYPTO_MPI_LIMB_DATA4(0x48, 0xE4, 0xEA, 0xFD),
  CRYPTO_MPI_LIMB_DATA4(0x1D, 0xFC, 0x5A, 0x33),
  CRYPTO_MPI_LIMB_DATA4(0x08, 0x07, 0x44, 0x5A),
  CRYPTO_MPI_LIMB_DATA4(0xB9, 0xFA, 0x0C, 0x4A),
  CRYPTO_MPI_LIMB_DATA4(0x63, 0x4D, 0x9C, 0x08),
  CRYPTO_MPI_LIMB_DATA4(0xFB, 0x27, 0x44, 0xF2),
  CRYPTO_MPI_LIMB_DATA4(0xA9, 0x72, 0x3D, 0x6F),
  CRYPTO_MPI_LIMB_DATA4(0x37, 0xB0, 0xBF, 0xE6),
  CRYPTO_MPI_LIMB_DATA4(0x18, 0xE9, 0x29, 0xA5),
  CRYPTO_MPI_LIMB_DATA4(0xDF, 0xCA, 0x4E, 0xFB),
  CRYPTO_MPI_LIMB_DATA4(0x9B, 0xC0, 0xFE, 0x84),
  CRYPTO_MPI_LIMB_DATA4(0xB2, 0xDD, 0xDC, 0xDF),
  CRYPTO_MPI_LIMB_DATA4(0x19, 0x6E, 0x50, 0x23),
  CRYPTO_MPI_LIMB_DATA4(0xD3, 0xCD, 0x14, 0x67),
  CRYPTO_MPI_LIMB_DATA4(0x1C, 0xC5, 0x8A, 0x7B),
  CRYPTO_MPI_LIMB_DATA4(0x90, 0xFD, 0x79, 0x60),
  CRYPTO_MPI_LIMB_DATA4(0xB1, 0x75, 0x54, 0x41),
  CRYPTO_MPI_LIMB_DATA4(0x98, 0x44, 0xC5, 0x66),
  CRYPTO_MPI_LIMB_DATA4(0x6C, 0x1F, 0xB7, 0x59),
  CRYPTO_MPI_LIMB_DATA4(0x35, 0x9E, 0xC3, 0xCA),
  CRYPTO_MPI_LIMB_DATA4(0xE4, 0xD4, 0xC5, 0x65),
  CRYPTO_MPI_LIMB_DATA4(0xFB, 0x4C, 0x7E, 0xCF),
  CRYPTO_MPI_LIMB_DATA4(0x15, 0x82, 0x06, 0xC8),
  CRYPTO_MPI_LIMB_DATA4(0xE8, 0xB9, 0x12, 0x65),
  CRYPTO_MPI_LIMB_DATA4(0xB8, 0x38, 0xE7, 0x57),
  CRYPTO_MPI_LIMB_DATA4(0x48, 0x09, 0xF0, 0x9E),
  CRYPTO_MPI_LIMB_DATA4(0x39, 0x6E, 0x81, 0xAD),
  CRYPTO_MPI_LIMB_DATA4(0xF7, 0x80, 0x79, 0xA1),
  CRYPTO_MPI_LIMB_DATA4(0xBC, 0xFD, 0x32, 0x2A),
```



```
  CRYPTO_MPI_LIMB_DATA4(0x19, 0xC2, 0x62, 0x81),
  CRYPTO_MPI_LIMB_DATA4(0x41, 0xA3, 0x73, 0xD8)
};

static const CRYPTO_MPI_LIMB _SECURE_RSA_PrivateKey_Expert_DP_aLimbs[] = {
  CRYPTO_MPI_LIMB_DATA4(0x4B, 0xCF, 0xDB, 0xDC),
  CRYPTO_MPI_LIMB_DATA4(0x52, 0x33, 0x3D, 0x8B),
  CRYPTO_MPI_LIMB_DATA4(0x66, 0xC6, 0x20, 0x55),
  CRYPTO_MPI_LIMB_DATA4(0x4E, 0x17, 0x39, 0xB4),
  CRYPTO_MPI_LIMB_DATA4(0x56, 0xDD, 0x9B, 0x3B),
  CRYPTO_MPI_LIMB_DATA4(0xE3, 0xEB, 0x1C, 0xC6),
  CRYPTO_MPI_LIMB_DATA4(0xAB, 0x80, 0xDB, 0x79),
  CRYPTO_MPI_LIMB_DATA4(0x57, 0x01, 0xE1, 0x47),
  CRYPTO_MPI_LIMB_DATA4(0x1B, 0xCE, 0x75, 0x35),
  CRYPTO_MPI_LIMB_DATA4(0xB7, 0xB5, 0x7D, 0x9A),
  CRYPTO_MPI_LIMB_DATA4(0xE6, 0xF1, 0x6B, 0xEB),
  CRYPTO_MPI_LIMB_DATA4(0x0D, 0x3A, 0x74, 0x0B),
  CRYPTO_MPI_LIMB_DATA4(0x08, 0xDB, 0xB5, 0x84),
  CRYPTO_MPI_LIMB_DATA4(0x96, 0x7D, 0xF5, 0x84),
  CRYPTO_MPI_LIMB_DATA4(0x1E, 0x43, 0x8A, 0x84),
  CRYPTO_MPI_LIMB_DATA4(0x84, 0x46, 0x05, 0xE0),
  CRYPTO_MPI_LIMB_DATA4(0x84, 0x78, 0x63, 0xAD),
  CRYPTO_MPI_LIMB_DATA4(0x72, 0xB6, 0x5D, 0xFB),
  CRYPTO_MPI_LIMB_DATA4(0x8A, 0x29, 0x4A, 0x99),
  CRYPTO_MPI_LIMB_DATA4(0xEC, 0x50, 0x7C, 0x54),
  CRYPTO_MPI_LIMB_DATA4(0x1E, 0xD8, 0xE7, 0xC7),
  CRYPTO_MPI_LIMB_DATA4(0x2D, 0x61, 0xAD, 0xBF),
  CRYPTO_MPI_LIMB_DATA4(0x4E, 0x0D, 0x7E, 0x80),
  CRYPTO_MPI_LIMB_DATA4(0x21, 0x7B, 0xAD, 0x85),
  CRYPTO_MPI_LIMB_DATA4(0xAE, 0x1E, 0x95, 0xAF),
  CRYPTO_MPI_LIMB_DATA4(0x9A, 0x0C, 0xAC, 0x09),
  CRYPTO_MPI_LIMB_DATA4(0xE9, 0xA8, 0x4F, 0x77),
  CRYPTO_MPI_LIMB_DATA4(0x3C, 0x1E, 0x05, 0x19),
  CRYPTO_MPI_LIMB_DATA4(0x40, 0xAB, 0x71, 0x93),
  CRYPTO_MPI_LIMB_DATA4(0x8D, 0xE9, 0xD1, 0x65),
  CRYPTO_MPI_LIMB_DATA4(0x2F, 0x05, 0xD7, 0x8A),
  CRYPTO_MPI_LIMB_DATA4(0x0F, 0x29, 0xD1, 0xCF)
};

static const CRYPTO_MPI_LIMB _SECURE_RSA_PrivateKey_Expert_DQ_aLimbs[] = {
  CRYPTO_MPI_LIMB_DATA4(0x09, 0x75, 0x79, 0x77),
  CRYPTO_MPI_LIMB_DATA4(0x06, 0x77, 0x27, 0x47),
  CRYPTO_MPI_LIMB_DATA4(0xDA, 0x0C, 0x7D, 0x14),
  CRYPTO_MPI_LIMB_DATA4(0x01, 0x34, 0x37, 0xA1),
  CRYPTO_MPI_LIMB_DATA4(0x20, 0xAF, 0xB4, 0x9A),
  CRYPTO_MPI_LIMB_DATA4(0x77, 0x2B, 0xAC, 0x60),
  CRYPTO_MPI_LIMB_DATA4(0x34, 0x8B, 0x6D, 0x81),
  CRYPTO_MPI_LIMB_DATA4(0xC4, 0x16, 0x9A, 0xDD),
  CRYPTO_MPI_LIMB_DATA4(0x1A, 0xAC, 0x94, 0x5A),
  CRYPTO_MPI_LIMB_DATA4(0x01, 0xDC, 0xF2, 0x0D),
  CRYPTO_MPI_LIMB_DATA4(0x53, 0x94, 0x45, 0x35),
  CRYPTO_MPI_LIMB_DATA4(0x9D, 0x40, 0xE2, 0x63),
  CRYPTO_MPI_LIMB_DATA4(0xBB, 0x7B, 0x3D, 0xE5),
  CRYPTO_MPI_LIMB_DATA4(0xD8, 0x13, 0x6F, 0xED),
  CRYPTO_MPI_LIMB_DATA4(0x3E, 0xFB, 0x4F, 0xB1),
  CRYPTO_MPI_LIMB_DATA4(0xD2, 0x77, 0x43, 0x00),
  CRYPTO_MPI_LIMB_DATA4(0x15, 0x8C, 0xB2, 0x21),
  CRYPTO_MPI_LIMB_DATA4(0x09, 0xBF, 0xBF, 0x17),
  CRYPTO_MPI_LIMB_DATA4(0xA1, 0x94, 0xD9, 0x43),
  CRYPTO_MPI_LIMB_DATA4(0x4D, 0x24, 0xCF, 0x5A),
  CRYPTO_MPI_LIMB_DATA4(0x3E, 0x00, 0x8A, 0xE9),
  CRYPTO_MPI_LIMB_DATA4(0xE9, 0x4A, 0x23, 0x06),
  CRYPTO_MPI_LIMB_DATA4(0x8F, 0xB3, 0x33, 0x75),
  CRYPTO_MPI_LIMB_DATA4(0x03, 0xF8, 0x67, 0x84),
  CRYPTO_MPI_LIMB_DATA4(0xB3, 0xB2, 0xBC, 0xDC),
  CRYPTO_MPI_LIMB_DATA4(0xD7, 0xA7, 0x92, 0x90),
  CRYPTO_MPI_LIMB_DATA4(0x03, 0x01, 0xC6, 0x55),
  CRYPTO_MPI_LIMB_DATA4(0x8A, 0xE7, 0x1C, 0xFB),
```

```
  CRYPTO_MPI_LIMB_DATA4(0xB4, 0xA6, 0x48, 0xE1),
  CRYPTO_MPI_LIMB_DATA4(0x25, 0xC2, 0x14, 0xA5),
  CRYPTO_MPI_LIMB_DATA4(0xCF, 0x3A, 0xBF, 0x7D),
  CRYPTO_MPI_LIMB_DATA4(0xEC, 0xC8, 0x5B, 0xCE)
};

static const CRYPTO_MPI_LIMB _SECURE_RSA_PrivateKey_Expert_QInv_aLimbs[] = {
  CRYPTO_MPI_LIMB_DATA4(0x8E, 0xA6, 0xAC, 0x41),
  CRYPTO_MPI_LIMB_DATA4(0x06, 0xDC, 0xEA, 0xBA),
  CRYPTO_MPI_LIMB_DATA4(0x6D, 0xBF, 0xC2, 0x82),
  CRYPTO_MPI_LIMB_DATA4(0x66, 0x2C, 0xBE, 0xBC),
  CRYPTO_MPI_LIMB_DATA4(0x74, 0xA1, 0xE3, 0x3B),
  CRYPTO_MPI_LIMB_DATA4(0x31, 0x10, 0x85, 0xE8),
  CRYPTO_MPI_LIMB_DATA4(0x87, 0x19, 0x4D, 0xA1),
  CRYPTO_MPI_LIMB_DATA4(0x47, 0x95, 0xAB, 0x2A),
  CRYPTO_MPI_LIMB_DATA4(0x8B, 0x2F, 0xBD, 0x0E),
  CRYPTO_MPI_LIMB_DATA4(0xAB, 0xF7, 0xCD, 0x2F),
  CRYPTO_MPI_LIMB_DATA4(0xDB, 0x00, 0x24, 0x1C),
  CRYPTO_MPI_LIMB_DATA4(0x9F, 0x56, 0xA7, 0xFF),
  CRYPTO_MPI_LIMB_DATA4(0xFC, 0xB3, 0x67, 0xD9),
  CRYPTO_MPI_LIMB_DATA4(0x24, 0xA5, 0x3C, 0x22),
  CRYPTO_MPI_LIMB_DATA4(0xC3, 0xC1, 0xE9, 0xEE),
  CRYPTO_MPI_LIMB_DATA4(0xDA, 0x45, 0x7B, 0xB6),
  CRYPTO_MPI_LIMB_DATA4(0xA9, 0x0C, 0x26, 0xEB),
  CRYPTO_MPI_LIMB_DATA4(0x29, 0xD3, 0xEA, 0x41),
  CRYPTO_MPI_LIMB_DATA4(0x7C, 0x5E, 0x90, 0x45),
  CRYPTO_MPI_LIMB_DATA4(0x30, 0xD0, 0xDB, 0x28),
  CRYPTO_MPI_LIMB_DATA4(0x71, 0x69, 0x00, 0xD1),
  CRYPTO_MPI_LIMB_DATA4(0xAB, 0x71, 0x55, 0xCF),
  CRYPTO_MPI_LIMB_DATA4(0x6A, 0xC6, 0xEA, 0x7F),
  CRYPTO_MPI_LIMB_DATA4(0x3B, 0x00, 0xD8, 0xC0),
  CRYPTO_MPI_LIMB_DATA4(0xBE, 0x0D, 0x71, 0x4C),
  CRYPTO_MPI_LIMB_DATA4(0xF4, 0x50, 0x25, 0xFB),
  CRYPTO_MPI_LIMB_DATA4(0xB7, 0x96, 0x66, 0x13),
  CRYPTO_MPI_LIMB_DATA4(0x4F, 0x12, 0x15, 0x34),
  CRYPTO_MPI_LIMB_DATA4(0xD7, 0x18, 0xDA, 0x0F),
  CRYPTO_MPI_LIMB_DATA4(0x4D, 0x2D, 0xDF, 0x47),
  CRYPTO_MPI_LIMB_DATA4(0x8D, 0x6E, 0x1C, 0xCD),
  CRYPTO_MPI_LIMB_DATA4(0x86, 0xE1, 0xA5, 0xAF)
};

static const CRYPTO_MPI_LIMB _SECURE_RSA_PrivateKey_Expert_N_aLimbs[] = {
  CRYPTO_MPI_LIMB_DATA4(0xEF, 0x73, 0xA3, 0x82),
  CRYPTO_MPI_LIMB_DATA4(0x05, 0x3A, 0x25, 0x1B),
  CRYPTO_MPI_LIMB_DATA4(0xC6, 0x77, 0xFE, 0xAE),
  CRYPTO_MPI_LIMB_DATA4(0x77, 0xFA, 0x56, 0x47),
  CRYPTO_MPI_LIMB_DATA4(0xDC, 0x0B, 0x00, 0x91),
  CRYPTO_MPI_LIMB_DATA4(0x08, 0x1D, 0x43, 0xDF),
  CRYPTO_MPI_LIMB_DATA4(0xA1, 0x7A, 0xF6, 0xD9),
  CRYPTO_MPI_LIMB_DATA4(0x0D, 0x15, 0x9E, 0x8A),
  CRYPTO_MPI_LIMB_DATA4(0xD1, 0xE2, 0x20, 0x2E),
  CRYPTO_MPI_LIMB_DATA4(0x4D, 0x1D, 0x54, 0x7C),
  CRYPTO_MPI_LIMB_DATA4(0x66, 0xE7, 0x34, 0xFA),
  CRYPTO_MPI_LIMB_DATA4(0xA1, 0xDF, 0x7F, 0xCE),
  CRYPTO_MPI_LIMB_DATA4(0x82, 0xCF, 0x98, 0x02),
  CRYPTO_MPI_LIMB_DATA4(0xC3, 0x8B, 0xCC, 0xC7),
  CRYPTO_MPI_LIMB_DATA4(0x24, 0xA9, 0x08, 0x7E),
  CRYPTO_MPI_LIMB_DATA4(0x3E, 0x50, 0x2D, 0xE0),
  CRYPTO_MPI_LIMB_DATA4(0xF6, 0xCC, 0x2E, 0x51),
  CRYPTO_MPI_LIMB_DATA4(0x82, 0x38, 0xA7, 0x1D),
  CRYPTO_MPI_LIMB_DATA4(0x2D, 0x17, 0xB1, 0x6C),
  CRYPTO_MPI_LIMB_DATA4(0x45, 0xEA, 0xA6, 0x6C),
  CRYPTO_MPI_LIMB_DATA4(0x11, 0x4E, 0xFF, 0x69),
  CRYPTO_MPI_LIMB_DATA4(0xA0, 0x98, 0x6B, 0x7D),
  CRYPTO_MPI_LIMB_DATA4(0x12, 0x02, 0x60, 0x01),
  CRYPTO_MPI_LIMB_DATA4(0x8A, 0xD0, 0x91, 0x1F),
  CRYPTO_MPI_LIMB_DATA4(0x5E, 0xC7, 0x6D, 0x6E),
  CRYPTO_MPI_LIMB_DATA4(0x6F, 0x6A, 0x8B, 0x8A),
```



```
  CRYPTO_MPI_LIMB_DATA4(0xBE, 0xE7, 0x27, 0x4E),
  CRYPTO_MPI_LIMB_DATA4(0xDA, 0x2F, 0x53, 0x1C),
  CRYPTO_MPI_LIMB_DATA4(0xF1, 0xEB, 0x91, 0x90),
  CRYPTO_MPI_LIMB_DATA4(0x62, 0xDA, 0x95, 0x20),
  CRYPTO_MPI_LIMB_DATA4(0xC6, 0xF2, 0x4E, 0x67),
  CRYPTO_MPI_LIMB_DATA4(0x0C, 0x6F, 0x27, 0x05),
  CRYPTO_MPI_LIMB_DATA4(0x3F, 0xC6, 0x67, 0x2F),
  CRYPTO_MPI_LIMB_DATA4(0x75, 0x49, 0xFB, 0xB7),
  CRYPTO_MPI_LIMB_DATA4(0x31, 0x91, 0x5D, 0xC1),
  CRYPTO_MPI_LIMB_DATA4(0x30, 0x59, 0xEC, 0xB0),
  CRYPTO_MPI_LIMB_DATA4(0xC5, 0x6B, 0xB6, 0x7B),
  CRYPTO_MPI_LIMB_DATA4(0x33, 0xBA, 0xE1, 0x31),
  CRYPTO_MPI_LIMB_DATA4(0x59, 0x36, 0x5A, 0x1E),
  CRYPTO_MPI_LIMB_DATA4(0xC0, 0x63, 0xBE, 0xD8),
  CRYPTO_MPI_LIMB_DATA4(0x2C, 0x86, 0xD0, 0x00),
  CRYPTO_MPI_LIMB_DATA4(0x9E, 0xA8, 0x70, 0x45),
  CRYPTO_MPI_LIMB_DATA4(0xC0, 0xA2, 0xBA, 0xB1),
  CRYPTO_MPI_LIMB_DATA4(0xC9, 0xB4, 0xD7, 0x6F),
  CRYPTO_MPI_LIMB_DATA4(0xAB, 0x64, 0x52, 0x37),
  CRYPTO_MPI_LIMB_DATA4(0xE1, 0xE2, 0xF5, 0xD0),
  CRYPTO_MPI_LIMB_DATA4(0x5A, 0xA2, 0x9D, 0x46),
  CRYPTO_MPI_LIMB_DATA4(0xF6, 0xE0, 0x5A, 0x9C),
  CRYPTO_MPI_LIMB_DATA4(0xCC, 0x58, 0xD6, 0xEE),
  CRYPTO_MPI_LIMB_DATA4(0x11, 0xB0, 0xCE, 0xF2),
  CRYPTO_MPI_LIMB_DATA4(0x63, 0x15, 0xD8, 0x9A),
  CRYPTO_MPI_LIMB_DATA4(0x59, 0x69, 0x05, 0x1C),
  CRYPTO_MPI_LIMB_DATA4(0x1D, 0x22, 0x46, 0x7F),
  CRYPTO_MPI_LIMB_DATA4(0x06, 0xAC, 0x10, 0xA6),
  CRYPTO_MPI_LIMB_DATA4(0x40, 0x31, 0x25, 0xF5),
  CRYPTO_MPI_LIMB_DATA4(0x3F, 0x09, 0xEA, 0x52),
  CRYPTO_MPI_LIMB_DATA4(0xFE, 0x18, 0x0B, 0xBB),
  CRYPTO_MPI_LIMB_DATA4(0x56, 0x25, 0x40, 0x3F),
  CRYPTO_MPI_LIMB_DATA4(0x86, 0x44, 0x6B, 0x8F),
  CRYPTO_MPI_LIMB_DATA4(0x11, 0xCE, 0xF4, 0xF8),
  CRYPTO_MPI_LIMB_DATA4(0x56, 0x70, 0x7C, 0x78),
  CRYPTO_MPI_LIMB_DATA4(0x28, 0xFE, 0xEB, 0xEA),
  CRYPTO_MPI_LIMB_DATA4(0xB7, 0x1D, 0x51, 0x92),
  CRYPTO_MPI_LIMB_DATA4(0x0E, 0x23, 0xCD, 0xB1)
};

static const CRYPTO_RSA_PRIVATE_KEY _SECURE_RSA_PrivateKey_Expert = {
  { CRYPTO_MPI_INIT_RO(_SECURE_RSA_PrivateKey_Expert_D_aLimbs) },
  { CRYPTO_MPI_INIT_RO(_SECURE_RSA_PrivateKey_Expert_P_aLimbs) },
  { CRYPTO_MPI_INIT_RO(_SECURE_RSA_PrivateKey_Expert_Q_aLimbs) },
  { CRYPTO_MPI_INIT_RO(_SECURE_RSA_PrivateKey_Expert_DP_aLimbs) },
  { CRYPTO_MPI_INIT_RO(_SECURE_RSA_PrivateKey_Expert_DQ_aLimbs) },
  { CRYPTO_MPI_INIT_RO(_SECURE_RSA_PrivateKey_Expert_QInv_aLimbs) },
  { CRYPTO_MPI_INIT_RO(_SECURE_RSA_PrivateKey_Expert_N_aLimbs) },
  { CRYPTO_MPI_INIT_RO_ZERO }
};
```

11.1.2 SECURE_RSA_PublicKey_Expert.h listing

```
static const CRYPTO_MPI_LIMB _SECURE_RSA_PublicKey_Expert_N_aLimbs[] = {
  CRYPTO_MPI_LIMB_DATA4(0xEF, 0x73, 0xA3, 0x82),
  CRYPTO_MPI_LIMB_DATA4(0x05, 0x3A, 0x25, 0x1B),
  CRYPTO_MPI_LIMB_DATA4(0xC6, 0x77, 0xFE, 0xAE),
  CRYPTO_MPI_LIMB_DATA4(0x77, 0xFA, 0x56, 0x47),
  CRYPTO_MPI_LIMB_DATA4(0xDC, 0x0B, 0x00, 0x91),
  CRYPTO_MPI_LIMB_DATA4(0x08, 0x1D, 0x43, 0xDF),
  CRYPTO_MPI_LIMB_DATA4(0xA1, 0x7A, 0xF6, 0xD9),
  CRYPTO_MPI_LIMB_DATA4(0x0D, 0x15, 0x9E, 0x8A),
  CRYPTO_MPI_LIMB_DATA4(0xD1, 0xE2, 0x20, 0x2E),
  CRYPTO_MPI_LIMB_DATA4(0x4D, 0x1D, 0x54, 0x7C),
  CRYPTO_MPI_LIMB_DATA4(0x66, 0xE7, 0x34, 0xFA),
  CRYPTO_MPI_LIMB_DATA4(0xA1, 0xDF, 0x7F, 0xCE),
  CRYPTO_MPI_LIMB_DATA4(0x82, 0xCF, 0x98, 0x02),
  CRYPTO_MPI_LIMB_DATA4(0xC3, 0x8B, 0xCC, 0xC7),
  CRYPTO_MPI_LIMB_DATA4(0x24, 0xA9, 0x08, 0x7E),
  CRYPTO_MPI_LIMB_DATA4(0x3E, 0x50, 0x2D, 0xE0),
  CRYPTO_MPI_LIMB_DATA4(0xF6, 0xCC, 0x2E, 0x51),
  CRYPTO_MPI_LIMB_DATA4(0x82, 0x38, 0xA7, 0x1D),
  CRYPTO_MPI_LIMB_DATA4(0x2D, 0x17, 0xB1, 0x6C),
  CRYPTO_MPI_LIMB_DATA4(0x45, 0xEA, 0xA6, 0x6C),
  CRYPTO_MPI_LIMB_DATA4(0x11, 0x4E, 0xFF, 0x69),
  CRYPTO_MPI_LIMB_DATA4(0xA0, 0x98, 0x6B, 0x7D),
  CRYPTO_MPI_LIMB_DATA4(0x12, 0x02, 0x60, 0x01),
  CRYPTO_MPI_LIMB_DATA4(0x8A, 0xD0, 0x91, 0x1F),
  CRYPTO_MPI_LIMB_DATA4(0x5E, 0xC7, 0x6D, 0x6E),
  CRYPTO_MPI_LIMB_DATA4(0x6F, 0x6A, 0x8B, 0x8A),
  CRYPTO_MPI_LIMB_DATA4(0xBE, 0xE7, 0x27, 0x4E),
  CRYPTO_MPI_LIMB_DATA4(0xDA, 0x2F, 0x53, 0x1C),
  CRYPTO_MPI_LIMB_DATA4(0xF1, 0xEB, 0x91, 0x90),
  CRYPTO_MPI_LIMB_DATA4(0x62, 0xDA, 0x95, 0x20),
  CRYPTO_MPI_LIMB_DATA4(0xC6, 0xF2, 0x4E, 0x67),
  CRYPTO_MPI_LIMB_DATA4(0x0C, 0x6F, 0x27, 0x05),
  CRYPTO_MPI_LIMB_DATA4(0x3F, 0xC6, 0x67, 0x2F),
  CRYPTO_MPI_LIMB_DATA4(0x75, 0x49, 0xFB, 0xB7),
  CRYPTO_MPI_LIMB_DATA4(0x31, 0x91, 0x5D, 0xC1),
  CRYPTO_MPI_LIMB_DATA4(0x30, 0x59, 0xEC, 0xB0),
  CRYPTO_MPI_LIMB_DATA4(0xC5, 0x6B, 0xB6, 0x7B),
  CRYPTO_MPI_LIMB_DATA4(0x33, 0xBA, 0xE1, 0x31),
  CRYPTO_MPI_LIMB_DATA4(0x59, 0x36, 0x5A, 0x1E),
  CRYPTO_MPI_LIMB_DATA4(0xC0, 0x63, 0xBE, 0xD8),
  CRYPTO_MPI_LIMB_DATA4(0x2C, 0x86, 0xD0, 0x00),
  CRYPTO_MPI_LIMB_DATA4(0x9E, 0xA8, 0x70, 0x45),
  CRYPTO_MPI_LIMB_DATA4(0xC0, 0xA2, 0xBA, 0xB1),
  CRYPTO_MPI_LIMB_DATA4(0xC9, 0xB4, 0xD7, 0x6F),
  CRYPTO_MPI_LIMB_DATA4(0xAB, 0x64, 0x52, 0x37),
  CRYPTO_MPI_LIMB_DATA4(0xE1, 0xE2, 0xF5, 0xD0),
  CRYPTO_MPI_LIMB_DATA4(0x5A, 0xA2, 0x9D, 0x46),
  CRYPTO_MPI_LIMB_DATA4(0xF6, 0xE0, 0x5A, 0x9C),
  CRYPTO_MPI_LIMB_DATA4(0xCC, 0x58, 0xD6, 0xEE),
  CRYPTO_MPI_LIMB_DATA4(0x11, 0xB0, 0xCE, 0xF2),
  CRYPTO_MPI_LIMB_DATA4(0x63, 0x15, 0xD8, 0x9A),
  CRYPTO_MPI_LIMB_DATA4(0x59, 0x69, 0x05, 0x1C),
  CRYPTO_MPI_LIMB_DATA4(0x1D, 0x22, 0x46, 0x7F),
  CRYPTO_MPI_LIMB_DATA4(0x06, 0xAC, 0x10, 0xA6),
  CRYPTO_MPI_LIMB_DATA4(0x40, 0x31, 0x25, 0xF5),
  CRYPTO_MPI_LIMB_DATA4(0x3F, 0x09, 0xEA, 0x52),
  CRYPTO_MPI_LIMB_DATA4(0xFE, 0x18, 0x0B, 0xBB),
  CRYPTO_MPI_LIMB_DATA4(0x56, 0x25, 0x40, 0x3F),
  CRYPTO_MPI_LIMB_DATA4(0x86, 0x44, 0x6B, 0x8F),
  CRYPTO_MPI_LIMB_DATA4(0x11, 0xCE, 0xF4, 0xF8),
  CRYPTO_MPI_LIMB_DATA4(0x56, 0x70, 0x7C, 0x78),
  CRYPTO_MPI_LIMB_DATA4(0x28, 0xFE, 0xEB, 0xEA),
  CRYPTO_MPI_LIMB_DATA4(0xB7, 0x1D, 0x51, 0x92),
  CRYPTO_MPI_LIMB_DATA4(0x0E, 0x23, 0xCD, 0xB1)
```



```
};

static const CRYPTO_MPI_LIMB _SECURE_RSA_PublicKey_Expert_E_aLimbs[] = {
  CRYPTO_MPI_LIMB_DATA3(0x01, 0x00, 0x01)
};

static const CRYPTO_RSA_PUBLIC_KEY _SECURE_RSA_PublicKey_Expert = {
  { CRYPTO_MPI_INIT_RO(_SECURE_RSA_PublicKey_Expert_N_aLimbs) },
  { CRYPTO_MPI_INIT_RO(_SECURE_RSA_PublicKey_Expert_E_aLimbs) },
};
```

Chapter 12

Indexes

12.1 Index of functions

