emSecure-ECDSA

Digital signature suite

User Guide & Reference Manual

Document: UM12004
Software Version: 2.46
Revision: 0
Date: June 1, 2022

A product of SEGGER Microcontroller GmbH

www.segger.com

Disclaimer

The information written in this document is assumed to be accurate without guarantee. The information in this manual is subject to change for functional or performance improvements without notice. SEGGER Microcontroller GmbH (SEGGER) assumes no responsibility for any errors or omissions in this document. SEGGER disclaims any warranties or conditions, express, implied or statutory for the fitness of the product for a particular purpose. It is your sole responsibility to evaluate the fitness of the product for any specific use.

Copyright notice

Trademarks

Names mentioned in this manual may be trademarks of their respective companies.

Brand and product names are trademarks or registered trademarks of their respective holders.

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

*By sending us an email your (personal) data will automatically be processed. For further information please refer to our privacy policy which is available at https://www.segger.com/legal/privacy-policy/.

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: June 1, 2022

Software	Revision	Date	By	Description
2.46	0	220501	PC	Update to latest software version.
2.44	0	210422	JL	Update to latest software version.
2.42	0	190311	PC	Update to latest software version.
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_ECDSA_VERSION` added. • `SECURE_ECDSA_GetVersionText()` added. • `SECURE_ECDSA_GetCopyrightText()` added.
2.32	0	170728	PC	Update to latest software version.
2.30	0	170518	PC	Chapter "Included applications" • Added tables for all command line options. Chapter "emSecure-ECDSA walkthrough" • Updated for simplified static keys. Chapter "Performance" • Updated for latest Embedded Studio and algorithms. • Added benchmark performance for twin multiply configurations. • Added complete listing of benchmark application. Chapter "Configuring emSecure-ECDSA" • Updated for using shared emCrypt component.
2.20	0	160426	PC	Added index of functions; no API changes.
2.16	0	151204	PC	Added sign and verify using precomputed digest.
2.14	0	150929	PC	Added incremental sign and verify capability.
2.12	0	150923	PC	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-ECDSA

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

1.1 What is emSecure-ECDSA?

emSecure-ECDSA is a SEGGER software package that allows creation and verification of digital signatures. One important feature is that emSecure-ECDSA 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-ECDSA 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-ECDSA 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-ECDSA 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-ECDSA into an existing product. emSecure-ECDSA 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-ECDSA?

emSecure-ECDSA 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-ECDSA creates digital signatures using the modern Elliptic Curve Digital Signature Algorithm.

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

- 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
- Full source code

1.4 Recommended project structure

We recommend keeping emSecure-ECDSA 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-ECDSA by simply replacing the directories. Your application files can be stored anywhere.

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

1.5 Package content

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

Files	Description
`Application`	emSSL sample applications for bare metal and embOS.
`Config`	Configuration header files.
`Doc`	emSecure-ECDSA documentation.
`CRYPTO`	Shared cryptographic library source code.
`SECURE`	emSecure-ECDSA implementation code.
`SEGGER`	SEGGER software component source code used in emSecure-ECDSA.
`Sample/Config`	Example emSecure-ECDSA configuration.
`Sample/Keys`	Example emSecure-ECDSA 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-ECDSA if you have old files included and therefore mix different versions. If you keep emSecure-ECDSA 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-ECDSA

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

2.1 Introduction

emSecure-ECDSA 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-ECDSA 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-ECDSA 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-ECDSA 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 190 ms on a Cortex-M4, running at 200 MHz, which is not significantly more time for a bootloader to start a firmware.

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

emSecure-ECDSA 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.

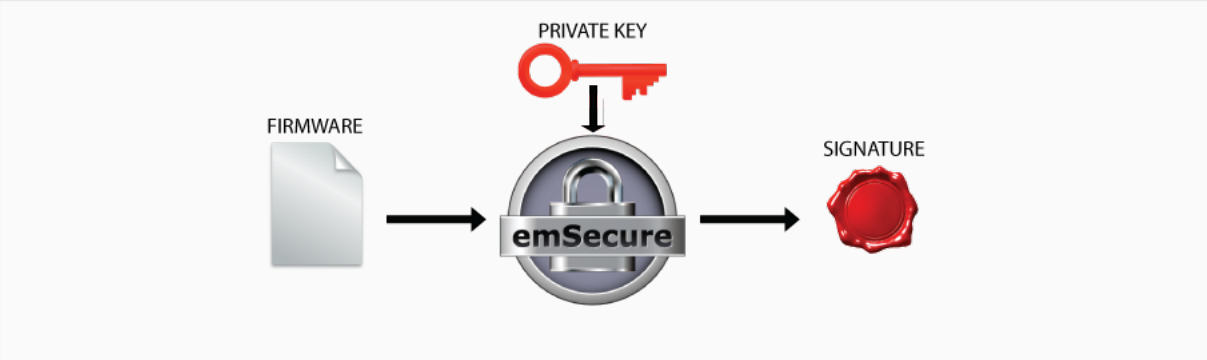
emSecure-ECDSA is licensed in the same way as other SEGGER middleware products and not covered by an open-source or required-attribution license. It can be integrated in any commercial or proprietary product without the obligation to disclose the combined source. It can be used royalty-free in your product.

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-ECDSA. To do this an emSecure-ECDSA key pair is generated one time.

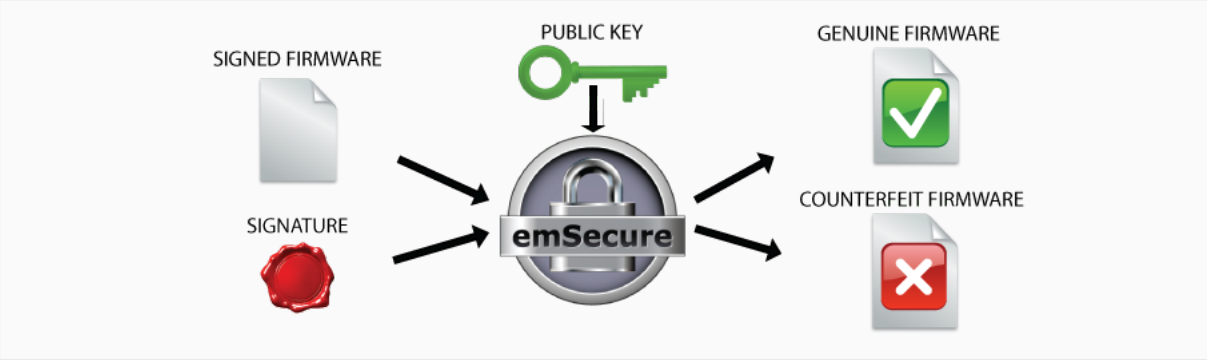
The private key will be included in the production process of the firmware. Once a firmware is created and ready to be shipped or included into a product it will be signed with this private key. The signature will be transferred and stored in the product alongside the firmware.



Signing 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.



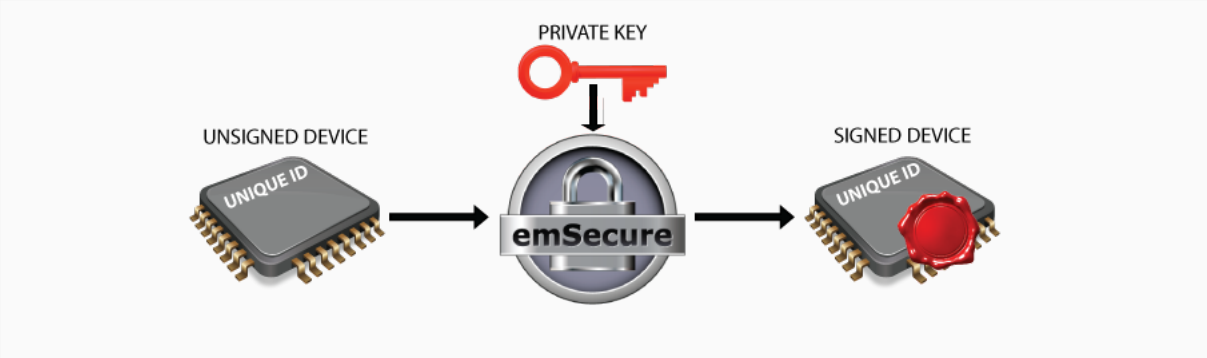
Verifying firmware

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-ECDSA will be used to sign each genuine product unit.

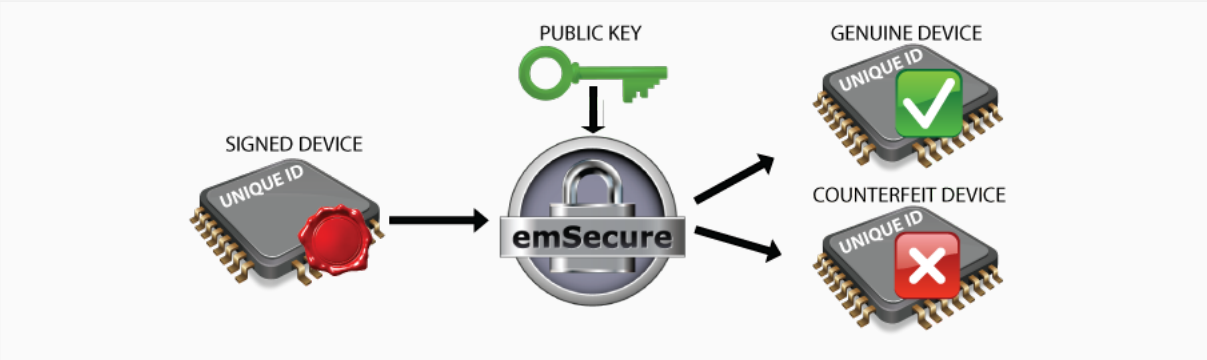
First an emSecure-ECDSA key pair is generated one time. This is likely done at the production site.



Signing firmware for a specific device

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-ECDSA 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.



Verifying firmware for a specific device

2.4 Additional measures to keep the system secure

When it comes to the degree of security emSecure-ECDSA 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.
- 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.

2.5 Signature strength

The strength of an ECDSA signature directly depends upon the elliptic curve chosen for the digital signature: more bits in the curve’s prime make the signature stronger, but also longer.

Chapter 3

Included applications

This chapter describes the applications which are part of the emSecure-ECDSA package. The executables can be found at `\Windows\SECURE\`. The source code of the basic utilities is included. For the source code of the key generator, please contact SEGGER: `mailto:info@segger.com`.

3.1 emSecure-ECDSA Key Generator

emSecure-ECDSA KeyGen generates an ECDSA public and a private key. The generation parameters can be set via command line options. The keys are saved in a common key file format and can be published and exchanged.

Usage

`emKeyGenECDSA.exe` [<Options>]

Command line options

emSecure-ECDSA 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`".
`-p`*n*	Selects the NIST prime curve P-*arg* which has *arg* bits in its base prime. The five prime curves supported by emSecure-ECDSA are P-192, P-224, P-256, P-384, and P-521.

3.2 Utilities

emSecure-ECDSA 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.

3.2.1 emSecure-ECDSA Sign

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

Usage

`emSignECDSA.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`".
`-d`	Use RFC 6979 deterministic ECDSA. Default.
`-nd`	Use random "secure k" signing.

3.2.2 emSecure-ECDSA Verify

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

Usage

`emVerifyECDSA.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`".

3.2.3 emSecure-ECDSA Print Key

emPrintKeyECDSA 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

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

Command line options

emSecure-ECDSA PrintKey accepts the following command line options.

Option	Description
`-v`	Increase verbosity of log output.

Option	Description
-x	Define objects with external linkage.
-p *string*	Prefix object names with *string*. Default is "_".

Chapter 4

emSecure-ECDSA 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 emKeyGenECDSA will construct the keys for you: they are generated using secure random numbers such that they are strong.

To use emKeyGenECDSA you must choose the elliptic curve that the signature scheme will use. Curves that have longer base primes offer stronger signatures but also require more storage and computation in order to both sign and verify.

The curves you can select from are:

- P-192
- P-224
- P-256
- P-384
- P-521

In this example we will choose the curve P-256 as it offers excellent security whilst having modest performance and memory requirements.

Generating the keys is a matter of running emKeyGenECDSA specifying P-256 and the curve:

```
C:> emKeyGenECDSA -p256

(c) 2014-2015 SEGGER Microcontroller GmbH & Co. KG    www.segger.com
emSecure-ECDSA KeyGen V2.38 compiled May 17 2018 10:43:55

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:> emSignECDSA test.txt

(c) 2014-2015 SEGGER Microcontroller GmbH & Co. KG    www.segger.com
emSecure-ECDSA Sign V2.38 compiled May 17 2018 10:43:55

Loading private key from emSecure.prv
Probing file to determine type of key: ECDSA key detected
Elliptic curve is P-256
Loading content from test.txt
Loaded content is 790 bytes
Writing ECDSA signature file test.txt.sig

C:> _
```

Once signed, you can verify the signature:

```
C:> emVerifyECDSA test.txt

(c) 2014-2015 SEGGER Microcontroller GmbH & Co. KG    www.segger.com
emSecure-ECDSA Verify V2.38 compiled May 17 2018 10:43:55

Loading public key from emSecure.pub
Probing file to determine type of key: ECDSA key detected
Elliptic curve is P-256
Loading ECDSA 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:> emVerifyECDSA test.txt

(c) 2014-2015 SEGGER Microcontroller GmbH & Co. KG    www.segger.com
emSecure-ECDSA Verify V2.38 compiled May 17 2018 10:43:55

Loading public key from emSecure.pub
Probing file to determine type of key: ECDSA key detected
Elliptic curve is P-256
Loading ECDSA 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 `emPrintKeyECDSA` 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:> emPrintKeyECDSA -p _SECURE_ECDSA_PrivateKey_P256 emSecure.prv \
      >SECURE_ECDSA_PrivateKey_P256.h
C>> emPrintKeyECDSA -p _SECURE_ECDSA_PrivateKey_P256 emSecure.pub \
      >SECURE_ECDSA_PublicKey_P256.h
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_ECDSA_Example1.c
Purpose     : Sign and verify a message.

Additional information:
  Preparations:
    None.

  Expected behavior:
    Signs and verifies an encrypted message using the Elliptic Curve
    Digital Signature Algorithm (ECSDA).
    For more detailed information see the emSecure ECSDA manual.

  Sample output:
    Signed message...SUCCESS!
    Verified message is correctly signed...SUCCESS!

*/

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

#include "SECURE_ECDSA.h"
#include "SECURE_ECDSA_PrivateKey_P256.h"
#include "SECURE_ECDSA_PublicKey_P256.h"
#include <stdio.h>

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

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



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

static U8 _aSignature[64];

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

static int _Sign(void) {
  return SECURE_ECDSA_Sign(&_SECURE_ECDSA_PrivateKey_P256,
                           &_aMessage[0],   sizeof(_aMessage),
                           &_aSignature[0], sizeof(_aSignature));
}

static int _Verify(void) {
  return SECURE_ECDSA_Verify(&_SECURE_ECDSA_PublicKey_P256,
                             &_aMessage[0],   sizeof(_aMessage),
                             &_aSignature[0], sizeof(_aSignature));
}

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

int main(void) {
  if (_Sign() > 0) {
    printf("Signed message...SUCCESS!\n");
    if (_Verify() > 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_ECDSA_PrivateKey_P256` in `SECURE_ECDSA_PrivateKey_P256.h` The public key is defined as a constant `SECURE_ECDSA_PublicKey_P256` in `SECURE_ECDSA_PublicKey_P256.h`

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

```
C:> SECURE_ECDSA_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…


```
if (_Sign() > 0) {
  printf("Signed message...SUCCESS!\n");
  _aMessage[0]++;
  if (_Verify() > 0) {
```

...causes verification to fail:

```
C:> SECURE_ECDSA_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-ECDSA 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_ECDSA_HASH_Init`.
- Repeatedly add the message content to sign or verify to the hash context using `SECURE_ECDSA_HASH_Add`.
- Finally, sign or verify the message using `SECURE_ECDSA_HASH_Sign` or `SECURE_ECDSA_HASH_Verify`.

The following example shows show this is achieved:

```
/*********************************************************************
*                   (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.

Additional information:
  Preparations:
    None.

  Expected behavior:
    Incrementally signs and verifies an encrypted message split over
    several parts using the Elliptic Curve Digital Signature
    Algorithm (ECSDA).
    For more detailed information see the emSecure ECSDA manual.

  Sample output:
    Signed message...SUCCESS!
    Verified message is correctly signed...SUCCESS!

*/

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

#include "SECURE_ECDSA.h"
#include "SECURE_ECDSA_PrivateKey_P256.h"
#include "SECURE_ECDSA_PublicKey_P256.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[64];

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

static int _Sign(void) {
  SECURE_ECDSA_HASH_CONTEXT Context;
  //
  // Sign message incrementally.
  //
  SECURE_ECDSA_HASH_Init(&Context);
  SECURE_ECDSA_HASH_Add (&Context, &_aMessagePart1[0], sizeof(_aMessagePart1));
  SECURE_ECDSA_HASH_Add (&Context, &_aMessagePart2[0], sizeof(_aMessagePart2));
  //
  return SECURE_ECDSA_HASH_Sign(&Context,
                                &_SECURE_ECDSA_PrivateKey_P256,
                                &_aSignature[0], sizeof(_aSignature));
}

static int _Verify(void) {
  SECURE_ECDSA_HASH_CONTEXT Context;
  //
  // Verify message incrementally.
  //
  SECURE_ECDSA_HASH_Init(&Context);
  SECURE_ECDSA_HASH_Add (&Context, &_aMessagePart1[0], sizeof(_aMessagePart1));
  SECURE_ECDSA_HASH_Add (&Context, &_aMessagePart2[0], sizeof(_aMessagePart2));
  //
  return SECURE_ECDSA_HASH_Verify(&Context,
                                  &_SECURE_ECDSA_PublicKey_P256,
                                  &_aSignature[0], sizeof(_aSignature));
}

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

int main(void) {
  if (_Sign() > 0) {
    printf("Signed message...SUCCESS!\n");
    if (_Verify() > 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

API reference

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

5.1 Preprocessor symbols

5.1.1 Version number

Description

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

Definition

```
#define SECURE_ECDSA_VERSION    24400
```

Symbols

Definition	Description
SECURE_ECDSA_VERSION	Format is “Mmmrr” so, for example, 23800 corresponds to version 2.38.

5.2 Elliptic curves

Declaration

```
extern const SECURE_ECDSA_CURVE SECURE_ECDSA_CURVE_P192;
extern const SECURE_ECDSA_CURVE SECURE_ECDSA_CURVE_P224;
extern const SECURE_ECDSA_CURVE SECURE_ECDSA_CURVE_P256;
extern const SECURE_ECDSA_CURVE SECURE_ECDSA_CURVE_P384;
extern const SECURE_ECDSA_CURVE SECURE_ECDSA_CURVE_P521;
```

Description

emSecure-ECDSA supports the following NIST prime elliptic curves:

C name	NIST prime curve
SECURE_ECDSA_CURVE_P192	P-192
SECURE_ECDSA_CURVE_P224	P-224
SECURE_ECDSA_CURVE_P256	P-256
SECURE_ECDSA_CURVE_P384	P-384
SECURE_ECDSA_CURVE_P521	P-521

See also

SECURE_ECDSA_InitPublicKey on page 43, *SECURE_ECDSA_InitPrivateKey* on page 42

5.3 API functions

Function	Description
Initialization	
SECURE_ECDSA_Init()	Initialize emSecure-ECDSA.
Complete message signature functions	
SECURE_ECDSA_Sign()	Sign message.
SECURE_ECDSA_Verify()	Verify message.
Precomputed digest message signature functions	
SECURE_ECDSA_SignDigest()	Sign message digest.
SECURE_ECDSA_VerifyDigest()	Verify message digest.
Incremental message signature functions	
SECURE_ECDSA_HASH_Init()	Initialize, incremental sign or verify.
SECURE_ECDSA_HASH_Add()	Add message data to hash.
SECURE_ECDSA_HASH_Sign()	Sign a message hash with a private key.
SECURE_ECDSA_HASH_Verify()	Verify message, incremental.
Initialization functions	
SECURE_ECDSA_InitPrivateKey()	Initialize private key.
SECURE_ECDSA_InitPublicKey()	Initialize public key.
Version information	
SECURE_ECDSA_GetCopyrightText()	Get copyright as printable string.
SECURE_ECDSA_GetVersionText()	Get version as printable string.

5.3.1 SECURE_ECDSA_Init()

Description

Initialize emSecure-ECDSA.

Prototype

```
void SECURE_ECDSA_Init(void);
```

Additional information

If not already installed as part of CRYPTO initialization, install the software-implemented SHA-256 hash function.

5.3.2 SECURE_ECDSA_Sign()

Description

Sign message.

Prototype

```
int SECURE_ECDSA_Sign(const SECURE_ECDSA_PRIVATE_KEY * pPrivate,
                      const U8                       * pMessage,
                            int                        MessageLen,
                            U8                       * pSignature,
                            int                        SignatureLen);
```

Parameters

Parameter	Description
pPrivate	Pointer to the private key to sign the message with.
pMessage	Pointer to message to sign.
MessageLen	Octet length of the message to sign.
pSignature	Pointer to buffer that received the generated signature.
SignatureLen	Octet length of the signature buffer.

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 buffer must be at least twice the number of bytes required for curve's underlying prime field. For instance, the P-256 prime is 256 bits (32 bytes) hence the signature will require 64 bytes. The P-521 prime requires 66 bytes and the signature 132 bytes.

5.3.3 SECURE_ECDSA_Verify()

Description

Verify message.

Prototype

```
int SECURE_ECDSA_Verify(const SECURE_ECDSA_PUBLIC_KEY * pPublic,
                        const U8                      * pMessage,
                              int                       MessageLen,
                        const U8                      * pSignature,
                              int                       SignatureLen);
```

Parameters

Parameter	Description
pPublic	Pointer to public key used to verify the message.
pMessage	Pointer to message to verify.
MessageLen	Octet length of the message.
pSignature	Pointer to signature to verify.
SignatureLen	Octet length of the signature.

Return value

< 0 Processing error verifying signature, signature is not verified.
= 0 Processing successful, signature is not verified.
> 0 Processing successful, signature is verified.

Additional information

The signature buffer must be exactly twice the number of bytes required for the curve's underlying prime field. For instance, the P-256 prime is 256 bits (32 bytes) hence the signature will require 64 bytes. The P-521 prime requires 66 bytes and the signature 132 bytes.

5.3.4 SECURE_ECDSA_SignDigest()

Description

Sign message digest.

Prototype

```
int SECURE_ECDSA_SignDigest(const SECURE_ECDSA_PRIVATE_KEY * pPrivate,
                            const U8                       * pDigest,
                                  U8                       * pSignature,
                                  int                        SignatureLen);
```

Parameters

Parameter	Description
pPrivate	Pointer to the private key to sign the digest with.
pDigest	Pointer to octet string that is the digest of the message.
pSignature	Pointer to buffer that receives the generated signature.
SignatureLen	Octet length of the signature buffer.

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 buffer must be at least twice the number of bytes required for curve's underlying prime field. For instance, the P-256 prime is 256 bits (32 bytes) hence the signature will require 64 bytes. The P-521 prime requires 66 bytes and the signature 132 bytes.

5.3.5 SECURE_ECDSA_VerifyDigest()

Description

Verify message digest.

Prototype

```
int SECURE_ECDSA_VerifyDigest(const SECURE_ECDSA_PUBLIC_KEY * pPublic,
                              const U8                      * pDigest,
                              const U8                      * pSignature,
                                    int                       SignatureLen);
```

Parameters

Parameter	Description
pPublic	Pointer to public key used to verify the message.
pDigest	Pointer to digest of the original message to be verified.
pSignature	Pointer to signature to verify.
SignatureLen	Octet length of the signature.

Return value

< 0 Processing error verifying signature, signature is not verified.
= 0 Processing successful, signature is not verified.
> 0 Processing successful, signature is verified.

Additional information

The signature buffer must be exactly twice the number of bytes required for the curve's underlying prime field. For instance, the P-256 prime is 256 bits (32 bytes) hence the signature will require 64 bytes. The P-521 prime requires 66 bytes and the signature 132 bytes.

5.3.6 SECURE_ECDSA_InitPrivateKey()

Description

Initialize private key.

Prototype

```
void SECURE_ECDSA_InitPrivateKey(      SECURE_ECDSA_PRIVATE_KEY   * pPrivate,
                                 const SECURE_ECDSA_KEY_PARAMETER * pParamX,
                                 const SECURE_ECDSA_CURVE         * pCurve);
```

Parameters

Parameter	Description
pPrivate	Pointer to private key to be initialized.
pParamX	Public key parameter X.
pCurve	Elliptic curve that points are embedded on.

5.3.7 SECURE_ECDSA_InitPublicKey()

Description

Initialize public key.

Prototype

```
void SECURE_ECDSA_InitPublicKey(      SECURE_ECDSA_PUBLIC_KEY    * pPublic,
                                const SECURE_ECDSA_KEY_PARAMETER * pParamYX,
                                const SECURE_ECDSA_KEY_PARAMETER * pParamYY,
                                const SECURE_ECDSA_CURVE         * pCurve);
```

Parameters

Parameter	Description
pPublic	Pointer to public key to be initialized.
pParamYX	Public key parameter Y (x coordinate).
pParamYY	Public key parameter Y (y coordinate).
pCurve	Elliptic curve that points are embedded on.

5.3.8 SECURE_ECDSA_HASH_Init()

Description

Initialize, incremental sign or verify.

Prototype

```
void SECURE_ECDSA_HASH_Init(SECURE_ECDSA_HASH_CONTEXT * pHash);
```

Parameters

Parameter	Description
pHash	Pointer to hash context.

5.3.9 SECURE_ECDSA_HASH_Add()

Description

Add message data to hash.

Prototype

```
void SECURE_ECDSA_HASH_Add(      SECURE_ECDSA_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.

5.3.10 SECURE_ECDSA_HASH_Sign()

Description

Sign a message hash with a private key.

Prototype

```
int SECURE_ECDSA_HASH_Sign(      SECURE_ECDSA_HASH_CONTEXT * pHash,
                           const SECURE_ECDSA_PRIVATE_KEY  * pPrivate,
                                 U8                        * pSignature,
                                 int                         SignatureLen);
```

Parameters

Parameter	Description
pHash	Pointer to hash context.
pPrivate	Pointer to the private key to sign the message with.
pSignature	Pointer to object that receives the generated signature.
SignatureLen	Octet length of the signature buffer.

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 buffer must be at least twice the number of bytes required for curve's underlying prime field. For instance, the P-256 prime is 256 bits (32 bytes) hence the signature will require 64 bytes. The P-521 prime requires 66 bytes and the signature 132 bytes.

5.3.11 SECURE_ECDSA_HASH_Verify()

Description

Verify message, incremental.

Prototype

```
int SECURE_ECDSA_HASH_Verify(      SECURE_ECDSA_HASH_CONTEXT * pHash,
                             const SECURE_ECDSA_PUBLIC_KEY   * pPublic,
                             const U8                        * pSignature,
                                   int                         SignatureLen);
```

Parameters

Parameter	Description
pHash	Pointer to hash context.
pPublic	Pointer to public key used to verify the message.
pSignature	Pointer to signature to verify.
SignatureLen	Octet length of the signature.

Return value

< 0 Processing error verifying signature, signature is not verified.
= 0 Processing successful, signature is not verified.
> 0 Processing successful, signature is verified.

5.3.12 SECURE_ECDSA_GetCopyrightText()

Description

Get copyright as printable string.

Prototype

```
char *SECURE_ECDSA_GetCopyrightText(void);
```

Return value

Zero-terminated copyright string.

5.3.13 SECURE_ECDSA_GetVersionText()

Description

Get version as printable string.

Prototype

```
char *SECURE_ECDSA_GetVersionText(void);
```

Return value

Zero-terminated version string.

Chapter 6

Configuring emSecure-ECDSA

emSecure-ECDSA can be configured via preprocessor flags at compile-time. All compile-time configuration flags are preconfigured with valid values, which match the requirements of most applications.

In order to configure the shared memory component and to select the way that SHA-256 is compiled to balance code size and execution performance you can change the compile-time flags in the main configuration files `CRYPTO_Conf.h` and `SEGGER_MEM_Conf.h`.

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.

6.1 Algorithm parameters

6.1.1 Key length

Default

```
#define SECURE_ECDSA_MAX_KEY_LENGTH      256
```

This preprocessor symbol, if overridden, must be set in the configuration file `SECURE_ECDSA_Conf.h`.

Description

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

The key length is in bits. If you are using P-256, for instance, set this to 256. If you intend to sign and verify signatures using P-521, set this to 521.

The default is 256 which supports any curve up to P-256.

6.2 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.

6.2.1 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.

6.2.2 Twin multiply

Default

```
#define CRYPTO_CONFIG_ECDSA_TWIN_MULTIPLY     1
```

This preprocessor symbol, if overridden, must be set in the configuration file `CRYPTO_Conf.h`.

Description

Configure whether ECDSA signature verification uses twin point multiplication or discrete point multiplication. ECDSA signature verification speed approximately doubles when using twin point multiplication at the expense of additional memory.

6.2.3 SHA-256 hash algorithm

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

Default

```
#define CRYPTO_CONFIG_SHA256_OPTIMIZE     0
```

Override

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

Description

Set this preprocessor symbol to zero to optimize the SHA-256 hash functions for size rather than for speed. When optimized for speed, the SHA-256 function is open coded and faster, but is significantly larger.

Profile

The following table shows required context size, lookup table (LUT) size, and code size in kilobytes for each configuration value. All values are approximate and for a Cortex-M3 processor.

Setting	Context size	LUT	LUT size	Code size	Total size
0	0.17 KB	Flash	0.3 KB	0.5 KB	0.8 KB
1	0.17 KB	-	-	7.7 KB	7.7 KB

Chapter 7

Performance and resource use

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

7.1 Performance

ECDSA sign and verify performance can be benchmarked with the application SECURE_ECDSA_Bench_Performance.c.

7.1.1 Twin multiply enabled

The following is the output for the Cortex-M4 on a SEGGER emPower board when configured for twin mulitply:

```
(c) 2014-2018 SEGGER Microcontroller GmbH & Co. KG    www.segger.com
emSecure-ECDSA Performance Benchmark compiled May 18 2018 18:44:15

Compiler: clang 4.0.0 (tags/RELEASE_400/final)
System:   Processor speed                   = 168.000 MHz
Config:   CRYPTO_VERSION                    = 20000 [2.00]
Config:   SECURE_ECDSA_VERSION              = 22000 [2.30]
Config:   CRYPTO_MPI_BITS_PER_LIMB          = 32
Config:   CRYPTO_CONFIG_ECDSA_TWIN_MULTIPLY = 1
Config:   CRYPTO_CONFIG_SHA1_OPTIMIZE       = 1
Config:   SECURE_ECDSA_MAX_KEY_LENGTH       = 521 bits

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

+-----------+----------+----------+----------+
|     Curve |  Message |     Sign |   Verify |
|           |   /bytes |      /ms |      /ms |
+-----------+----------+----------+----------+
| secp192r1 |        0 |   156.86 |    73.93 |
| secp192r1 |     1024 |   161.29 |    72.36 |
| secp192r1 |   102400 |   200.60 |   116.78 |
+-----------+----------+----------+----------+
| secp224r1 |        0 |   200.20 |    93.73 |
| secp224r1 |     1024 |   193.83 |    91.91 |
| secp224r1 |   102400 |   238.60 |   135.00 |
+-----------+----------+----------+----------+
| secp256r1 |        0 |   298.00 |   142.25 |
| secp256r1 |     1024 |   308.25 |   140.12 |
| secp256r1 |   102400 |   348.33 |   181.67 |
+-----------+----------+----------+----------+
| secp384r1 |        0 |   504.50 |   234.40 |
| secp384r1 |     1024 |   515.50 |   235.40 |
| secp384r1 |   102400 |   539.00 |   279.00 |
+-----------+----------+----------+----------+
| secp521r1 |        0 |   899.50 |   421.33 |
| secp521r1 |     1024 |   906.00 |   419.67 |
| secp521r1 |   102400 |   940.50 |   464.67 |
+-----------+----------+----------+----------+

Benchmark complete
```

7.1.2 Twin multiply disabled

The following is the output for the Cortex-M4 on a SEGGER emPower board when configured with twin mulitply disabled:

```
(c) 2014-2018 SEGGER Microcontroller GmbH & Co. KG    www.segger.com
emSecure-ECDSA Performance Benchmark compiled May 18 2018 18:48:33

Compiler: clang 4.0.0 (tags/RELEASE_400/final)
System:   Processor speed                   = 168.000 MHz
Config:   CRYPTO_VERSION                    = 20000 [2.00]
Config:   SECURE_ECDSA_VERSION              = 22000 [2.30]
```



```
Config:   CRYPTO_MPI_BITS_PER_LIMB          = 32
Config:   CRYPTO_CONFIG_ECDSA_TWIN_MULTIPLY = 0
Config:   CRYPTO_CONFIG_SHA1_OPTIMIZE       = 1
Config:   SECURE_ECDSA_MAX_KEY_LENGTH       = 521 bits

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

+-----------+----------+----------+----------+
|     Curve |  Message |     Sign |   Verify |
|           |   /bytes |      /ms |      /ms |
+-----------+----------+----------+----------+
| secp192r1 |        0 |   158.14 |   153.71 |
| secp192r1 |     1024 |   162.57 |   149.43 |
| secp192r1 |   102400 |   201.60 |   194.83 |
+-----------+----------+----------+----------+
| secp224r1 |        0 |   200.40 |   183.83 |
| secp224r1 |     1024 |   194.17 |   189.67 |
| secp224r1 |   102400 |   238.60 |   231.00 |
+-----------+----------+----------+----------+
| secp256r1 |        0 |   291.25 |   284.50 |
| secp256r1 |     1024 |   301.50 |   296.50 |
| secp256r1 |   102400 |   341.67 |   323.50 |
+-----------+----------+----------+----------+
| secp384r1 |        0 |   495.00 |   480.00 |
| secp384r1 |     1024 |   505.50 |   490.33 |
| secp384r1 |   102400 |   529.50 |   528.00 |
+-----------+----------+----------+----------+
| secp521r1 |        0 |   888.00 |   892.00 |
| secp521r1 |     1024 |   894.00 |   873.50 |
| secp521r1 |   102400 |   928.50 |   931.50 |
+-----------+----------+----------+----------+

Benchmark complete
```


7.1.3 SECURE_ECDSA_Bench_Performance.c listing

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

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

File        : SECURE_ECDSA_Bench_Performance.c
Purpose     : Benchmark emSecure-ECDSA performance.

*/

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

#include "SECURE_ECDSA.h"
#include "SECURE_ECDSA_PrivateKey_P192.h"
#include "SECURE_ECDSA_PrivateKey_P224.h"
#include "SECURE_ECDSA_PrivateKey_P256.h"
#include "SECURE_ECDSA_PrivateKey_P384.h"
#include "SECURE_ECDSA_PrivateKey_P521.h"
#include "SECURE_ECDSA_PublicKey_P192.h"
#include "SECURE_ECDSA_PublicKey_P224.h"
#include "SECURE_ECDSA_PublicKey_P256.h"
#include "SECURE_ECDSA_PublicKey_P384.h"
#include "SECURE_ECDSA_PublicKey_P521.h"
#include "SEGGER_SYS.h"

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

typedef struct {
  const CRYPTO_ECDSA_PRIVATE_KEY * pPrivateKey;
  const CRYPTO_ECDSA_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_ECDSA_PrivateKey_P192, &_SECURE_ECDSA_PublicKey_P192, _aMessage_100k,        0u },
  { &_SECURE_ECDSA_PrivateKey_P192, &_SECURE_ECDSA_PublicKey_P192, _aMessage_100k,     1024u },
  { &_SECURE_ECDSA_PrivateKey_P192, &_SECURE_ECDSA_PublicKey_P192, _aMessage_100k, 100*1024u },
  { NULL,                           NULL,                          NULL,                0u },
  { &_SECURE_ECDSA_PrivateKey_P224, &_SECURE_ECDSA_PublicKey_P224, _aMessage_100k,        0u },
  { &_SECURE_ECDSA_PrivateKey_P224, &_SECURE_ECDSA_PublicKey_P224, _aMessage_100k,     1024u },
  { &_SECURE_ECDSA_PrivateKey_P224, &_SECURE_ECDSA_PublicKey_P224, _aMessage_100k, 100*1024u },
  { NULL,                           NULL,                          NULL,                0u },
  { &_SECURE_ECDSA_PrivateKey_P256, &_SECURE_ECDSA_PublicKey_P256, _aMessage_100k,        0u },
  { &_SECURE_ECDSA_PrivateKey_P256, &_SECURE_ECDSA_PublicKey_P256, _aMessage_100k,     1024u },
  { &_SECURE_ECDSA_PrivateKey_P256, &_SECURE_ECDSA_PublicKey_P256, _aMessage_100k, 100*1024u },
  { NULL,                           NULL,                          NULL,                0u },
  { &_SECURE_ECDSA_PrivateKey_P384, &_SECURE_ECDSA_PublicKey_P384, _aMessage_100k,        0u },
  { &_SECURE_ECDSA_PrivateKey_P384, &_SECURE_ECDSA_PublicKey_P384, _aMessage_100k,     1024u },
  { &_SECURE_ECDSA_PrivateKey_P384, &_SECURE_ECDSA_PublicKey_P384, _aMessage_100k, 100*1024u },
  { NULL,                           NULL,                          NULL,                0u },
  { &_SECURE_ECDSA_PrivateKey_P521, &_SECURE_ECDSA_PublicKey_P521, _aMessage_100k,        0u },
  { &_SECURE_ECDSA_PrivateKey_P521, &_SECURE_ECDSA_PublicKey_P521, _aMessage_100k,     1024u },
  { &_SECURE_ECDSA_PrivateKey_P521, &_SECURE_ECDSA_PublicKey_P521, _aMessage_100k, 100*1024u },
};

/*********************************************************************
*
*       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_SYS_IO_Printf("| %9s | %8u | ", pPara->pPublicKey->pCurve->aCurveName, pPara->MessageLen);
  //
  Loops = 0;
  OneSecond = SEGGER_SYS_OS_ConvertMicrosToTicks(1000000);
  T0 = SEGGER_SYS_OS_GetTimer();
  do {
    SignatureLen = SECURE_ECDSA_Sign(pPara->pPrivateKey,
                                     pPara->pMesssage, pPara->MessageLen,
                                     aSignature,       sizeof(aSignature));
    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 {
      Status = SECURE_ECDSA_Verify(pPara->pPublicKey,
                                   pPara->pMesssage, pPara->MessageLen,
                                   aSignature,       SignatureLen);
      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_ECDSA_Init();
  SEGGER_SYS_Init();
  //
  SEGGER_SYS_IO_Printf("\n");
  SEGGER_SYS_IO_Printf("%s    www.segger.com\n", SECURE_ECDSA_GetCopyrightText());
  SEGGER_SYS_IO_Printf("emSecure-ECDSA 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_ECDSA_VERSION             = %u [%s]\n",
                       SECURE_ECDSA_VERSION, SECURE_ECDSA_GetVersionText());
  SEGGER_SYS_IO_Printf("Config:   CRYPTO_MPI_BITS_PER_LIMB         = %u\n",
                       CRYPTO_MPI_BITS_PER_LIMB);
  SEGGER_SYS_IO_Printf("Config:   CRYPTO_CONFIG_ECDSA_TWIN_MULTIPLY = %u\n",
                       CRYPTO_CONFIG_ECDSA_TWIN_MULTIPLY);
  SEGGER_SYS_IO_Printf("Config:   CRYPTO_CONFIG_SHA256_OPTIMIZE    = %u\n",
                       CRYPTO_CONFIG_SHA256_OPTIMIZE);
  SEGGER_SYS_IO_Printf("Config:   SECURE_ECDSA_MAX_KEY_LENGTH      = %u bits\n",
                       SECURE_ECDSA_MAX_KEY_LENGTH);
  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("|     Curve |  Message |     Sign |   Verify |\n");
  SEGGER_SYS_IO_Printf("|           |   /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 ***************************/
```


7.2 Memory footprint

The following table lists the memory requirements of emSecure-ECDSA configured for operation using the P-256 curve. There are two components to the ROM use, one for the emSecure-ECDSA code and one for the emCrypt component that can be shared with other security products such as emSecure-RSA, emSSL, and emSSH.

Twin multiply disabled

Process	ROM (SECURE)	ROM (CRYPTO)	Total ROM	Static RAM
Sign only	0.26 KB	10.68 KB	10.94 KB	0.01 KB
Verify only	0.36 KB	9.34 KB	9.70 KB	0.01 KB
Sign and verify	0.70 KB	11.04 KB	11.74 KB	0.01 KB

Twin multiply enabled

Process	ROM (SECURE)	ROM (CRYPTO)	Total ROM	Static RAM
Sign only	0.26 KB	10.68 KB	10.94 KB	0.01 KB
Verify only	0.36 KB	10.00 KB	10.36 KB	0.01 KB
Sign and verify	0.70 KB	11.68 KB	12.38 KB	0.01 KB

Chapter 8

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 9

Appendix

9.1 Example key pair

The example key pair was generated with `emKeyGenECDSA` and used in the example walkthrough.

9.1.1 SECURE_ECDSA_PrivateKey_P256.h listing

```
static const CRYPTO_MPI_LIMB _SECURE_ECDSA_PrivateKey_P256_X_aLimbs[] = {
  CRYPTO_MPI_LIMB_DATA4(0x91, 0xC9, 0x0B, 0x85),
  CRYPTO_MPI_LIMB_DATA4(0x0F, 0x58, 0xD4, 0x46),
  CRYPTO_MPI_LIMB_DATA4(0x15, 0xA1, 0x3B, 0x4C),
  CRYPTO_MPI_LIMB_DATA4(0x17, 0x00, 0xEC, 0xE9),
  CRYPTO_MPI_LIMB_DATA4(0xC8, 0xB1, 0x62, 0x5A),
  CRYPTO_MPI_LIMB_DATA4(0x9F, 0x8D, 0x88, 0x8A),
  CRYPTO_MPI_LIMB_DATA4(0x52, 0x1F, 0x6C, 0xB5),
  CRYPTO_MPI_LIMB_DATA4(0x1E, 0x8D, 0x1B, 0xE0)
};

static const CRYPTO_ECDSA_PRIVATE_KEY _SECURE_ECDSA_PrivateKey_P256 = {
  { CRYPTO_MPI_INIT_RO(_SECURE_ECDSA_PrivateKey_P256_X_aLimbs) },
  &CRYPTO_EC_CURVE_secp256r1
};
```

9.1.2 SECURE_ECDSA_PublicKey_P256.h listing

```
static const CRYPTO_MPI_LIMB _SECURE_ECDSA_PublicKey_P256_YX_aLimbs[] = {
  CRYPTO_MPI_LIMB_DATA4(0x28, 0x59, 0x5E, 0x86),
  CRYPTO_MPI_LIMB_DATA4(0xAD, 0xED, 0x67, 0xC9),
  CRYPTO_MPI_LIMB_DATA4(0x82, 0x0A, 0x6B, 0x47),
  CRYPTO_MPI_LIMB_DATA4(0xA9, 0x44, 0x41, 0xC1),
  CRYPTO_MPI_LIMB_DATA4(0xD6, 0x46, 0xEE, 0x03),
  CRYPTO_MPI_LIMB_DATA4(0xB4, 0x69, 0x77, 0xDD),
  CRYPTO_MPI_LIMB_DATA4(0xF4, 0x82, 0x22, 0xB5),
  CRYPTO_MPI_LIMB_DATA4(0xA7, 0xE3, 0x75, 0xB9)
};

static const CRYPTO_MPI_LIMB _SECURE_ECDSA_PublicKey_P256_YY_aLimbs[] = {
  CRYPTO_MPI_LIMB_DATA4(0x28, 0xC5, 0xF6, 0xF9),
  CRYPTO_MPI_LIMB_DATA4(0x1D, 0x5F, 0xAB, 0x1C),
  CRYPTO_MPI_LIMB_DATA4(0x3C, 0x93, 0x5E, 0x34),
  CRYPTO_MPI_LIMB_DATA4(0x29, 0xB1, 0xB5, 0x94),
  CRYPTO_MPI_LIMB_DATA4(0xE1, 0xB7, 0x45, 0x2F),
  CRYPTO_MPI_LIMB_DATA4(0x2B, 0x4E, 0xD9, 0xAD),
  CRYPTO_MPI_LIMB_DATA4(0x38, 0xA1, 0xAE, 0x27),
  CRYPTO_MPI_LIMB_DATA4(0x57, 0x97, 0x7E, 0x5C)
};

static const CRYPTO_ECDSA_PUBLIC_KEY _SECURE_ECDSA_PublicKey_P256 = { {
  { CRYPTO_MPI_INIT_RO(_SECURE_ECDSA_PublicKey_P256_YX_aLimbs) },
  { CRYPTO_MPI_INIT_RO(_SECURE_ECDSA_PublicKey_P256_YY_aLimbs) },
  { CRYPTO_MPI_INIT_RO_ZERO },
  { CRYPTO_MPI_INIT_RO_ZERO },
  },
  &CRYPTO_EC_CURVE_secp256r1
};
```


Chapter 10

Indexes

10.1 Index of functions

