

# VS1005 DEVCRYPT DRIVER

VS1005g

**All information in this document is provided as-is without warranty. Features are subject to change without notice.**

Revision History			
Rev.	Date	Author	Description
1.01	2023-11-17	HH	Bug and documentation fix.
1.00	2023-11-15	HH	Initial release of documentation.

## Contents

<b>VS1005 devCrypt Driver Front Page</b>	<b>1</b>
<b>Table of Contents</b>	<b>2</b>
<b>1 Introduction</b>	<b>3</b>
<b>2 Disclaimer</b>	<b>3</b>
<b>3 Definitions</b>	<b>3</b>
<b>4 Overview</b>	<b>4</b>
4.1 Directories in this Package . . . . .	4
4.2 Short Introduction to ChaCha20 . . . . .	5
4.2.1 Secret Key . . . . .	5
4.2.2 Nonce . . . . .	6
4.3 VLSI Solution's ChaCha20 Encryptor and Decryptor . . . . .	7
4.3.1 The PC/Win/Linux/C ChaCha20 Encryptor . . . . .	7
4.3.2 The VS1005 devCrypt ChaCha20 Decryption Driver . . . . .	7
4.3.3 The VS1005 LoadCrypt Launcher . . . . .	7
<b>5 Requirements</b>	<b>8</b>
<b>6 Testing devCrypt</b>	<b>9</b>
6.1 Copying Test Files . . . . .	9
6.2 Booting the Board . . . . .	10
6.3 Starting the Driver . . . . .	11
6.4 Running Decryption Tests . . . . .	12
6.5 Testing Decryption Speed . . . . .	14
<b>7 Encrypting Files</b>	<b>15</b>
7.1 Choosing or Generating a Secret Key . . . . .	15
7.2 Encrypting Files . . . . .	17
<b>8 Latest Document Version Changes</b>	<b>18</b>
<b>9 Contact Information</b>	<b>19</b>

## List of Figures

## 1 Introduction

The VS1005 VSOS decryption library devCrypt.dll is a powerful and fast decryptor device driver that allows for VS1005g to decrypt secure files using the modern and fast ChaCha20 encryption algorithm.

A disclaimer and definitions are presented in Chapters 2 and 3, respectively.

An overview of the decryption device driver is given in Chapter 4, *Overview*, including some theoretical background and practical considerations that affect security.

The overview is followed by prerequisites needed to run and test the driver in Chapter 5, *Requirements*.

How to run the decryption driver is shown in Chapter 6, *Testing devCrypt*.

How to generate your own encryption key and then encrypt your files using the generated key is shown in Chapter 7, *Encrypting Files*.

The document ends with Chapter 8, *Latest Document Version Changes*, and Chapter 9, *Contact Information*.

## 2 Disclaimer

VLSI Solution makes everything it can to make this documentation as accurate as possible. However, no warranties or guarantees are given for the correctness of this documentation.

## 3 Definitions

**ChaCha20** Secure symmetrical encryption method, variation of Salsa20. See <https://en.wikipedia.org/wiki/Salsa20> for more details.

**DSP** Digital Signal Processor.

**Ki** “Kibi” =  $2^{10} = 1024$  (IEC 60027-2).

**Mi** “Mebi” =  $2^{20} = 1048576$  (IEC 60027-2).

**VS\_DSP<sup>4</sup>** VLSI Solution’s DSP core.

**VSIDE** VLSI Solution’s Integrated Development Environment.

**VSOS** VLSI Solution’s Operating System.

## 4 Overview

This package offers a VS1005g decryption library for the modern ChaCha20 encryption algorithm. This makes it possible to create encrypted files that cannot be decrypted without knowledge of the Secret Key of the application.

VLSI Solution has chosen ChaCha20 as its encryption algorithm for the following reasons:

- Security. As of writing this (2023-11-14), there are no known vulnerabilities against ChaCha20.
- Speed. Decoding ChaCha20 files is fast compared to many other ciphers, and close to 1 MiB/second on a VS1005 running at 86 MHz. See Chapter 6.5 for details.
- Random access. Files do not need to be decoded from the beginning to the end. This way it is possible to e.g. fast forward and reverse audio files as if there was no encryption.
- Public domain. There doesn't appear to exist patent mines for ChaCha20.

A similar and compatible package is also available for VS1010.

### 4.1 Directories in this Package

This package contains the following directories:

**devCryptSolution** VSIDE Solution for VS1005g VSOS. Contains the decrypting library. See Chapters 4.3.2 and 6.3 for details.

**Docs/** Documentation, including this document.

**LoadCryptSolution/** VSIDE Solution for VS1005g VSOS. Contains a simple example on how to start the decrypting library. See Chapters 4.3.3 and 6.3 for details.

**MainFolder/** Contains files that should be copied to the main folder of the VS1005g System Disk S: for demonstration of the decryption library. See Chapter 6.1 for details.

**SYS/** Contains files that should be copied to the S:SYS/ folder of the VS1005g System Disk for demonstration of the decryption library. See Chapter 6.1 for details.

**vsiencryptor/** Windows binary and full C source code for a program that can generate Secret Keys and encrypt files. See Chapters 4.3.1 and 7 for details.

## 4.2 Short Introduction to ChaCha20

From Wikipedia (2023-11-09), <https://en.wikipedia.org/wiki/Salsa20> :

Salsa20 and the closely related ChaCha are stream ciphers developed by Daniel J. Bernstein. Salsa20, the original cipher, was designed in 2005, then later submitted to the eSTREAM European Union cryptographic validation process by Bernstein. ChaCha is a modification of Salsa20 published in 2008. It uses a new round function that increases diffusion and increases performance on some architectures.

Both ciphers are built on a pseudorandom function based on add-rotate-XOR (ARX) operations — 32-bit addition, bitwise addition (XOR) and rotation operations. The core function maps a 256-bit key, a 64-bit nonce, and a 64-bit counter to a 512-bit block of the key stream (a Salsa version with a 128-bit key also exists). This gives Salsa20 and ChaCha the unusual advantage that the user can efficiently seek to any position in the key stream in constant time. Salsa20 offers speeds of around 4–14 cycles per byte in software on modern x86 processors, and reasonable hardware performance. It is not patented, and Bernstein has written several public domain implementations optimized for common architectures.

For best security, VLSI Solution's implementation of ChaCha20 uses the most modern parameters: 256 bits for the Secret Key, and 96 bits for Nonce. For best speed, VLSI Solution's implementation's inner loops are written in hand-optimized assembler.

From Wikipedia (2023-11-09), <https://en.wikipedia.org/wiki/Salsa20> :

ChaCha20 usually offers better performance than the more prevalent Advanced Encryption Standard (AES) algorithm on systems where the CPU does not feature AES acceleration (such as the AES instruction set for x86 processors). As a result, ChaCha20 is sometimes preferred over AES in certain use cases involving mobile devices, which mostly use ARM-based CPUs.

### 4.2.1 Secret Key

VLSI Solution's implementation of ChaCha20 has a 256-bit Secret Key. This key is required to open the files encrypted with the key.

The user may generate their own Secret Key with any key generation algorithm, or use VLSI Solution's encryptor `vlsiencryptor.exe` to generate one. After generating a Secret Key, all files intended to be used in the same file system must be encoded with the same key.

The devCrypt library has to be provided with the Secret Key when starting it up. It is the responsibility of the implementation to keep the key secret. The safest way to keep the

key secret is to run a customized version of VSOS from Internal Flash memory. This is possible with VS1005G-F-Q and VS1205G-F-Q).

Devices that don't have an Internal Flash memory, like VS1010, VS1005G-F, or VS1205G-F, make the keys recoverable if the attacker has physical access to the device.

For information on how to give best protection to your key in your commercial application, contact VLSI Solution, e.g. at [sales@vlsi.fi](mailto:sales@vlsi.fi).

#### **4.2.2 Nonce**

VLSI Solution's implementation of ChaCha20 has a 96-bit Nonce. Nonce is not a secret, but it should be different for each file. If it is not, the security of multiple files encoded with the same Secret Key may be compromised.

VLSI Solution's encryptor adds a randomly generated Nonce (12 bytes) to the end of each encrypted file.

Nonce is automatically read by the devCrypt library when opening a file.

### 4.3 VLSI Solution's ChaCha20 Encryptor and Decryptor

In this program package, VLSI Solution provides both a ChaCha20 encryptor and a decryptor for VS1005g.

A similar and compatible package is also available for VS1010.

#### 4.3.1 The PC/Win/Linux/C ChaCha20 Encryptor

A ChaCha20 encoder is provided as standard C89 source code (with the addition of a Posix function call `clock_gettime()`), as well as a precompiled Windows executable.

The encryptor, including its full source code, is located in directory `vlsiencryptor/`.

#### 4.3.2 The VS1005 devCrypt ChaCha20 Decryption Driver

A ChaCha20 decoder, `devCrypt`, is provided as a VSIDE Solution for VS1005g VSOS.

Currently the `devCrypt` driver is only capable of decrypting files. If there is enough commercial interest, an encrypting version of the driver may also be implemented.

The full source code is located in directory `devCryptSolution/`.

#### 4.3.3 The VS1005 LoadCrypt Launcher

An example launch application for `devCrypt`, called `LoadCrypt`, is provided. Note, however, that the launch application does not do any attempts to hide the Secret Key. So it is only usable for demonstrating decryption. It is absolutely not suitable for commercial production.

For information on how to keep the key secret in your commercial application, contact VLSI Solution, e.g. at [sales@vlsi.fi](mailto:sales@vlsi.fi).

The full source code is located in directory `LoadCryptSolution/`.

## 5 Requirements

Before using the VSOS Shell Environment, you need to have the following building blocks:

- VS1005g Developer Board. The VS1005g BreakOut Board will work, too, but these instructions have been tested with the DevBoard.
- Latest version of VSOS installed (VSOS 3.66 or higher recommended).
- USB cable between DevBoard and PC for uploading new software.
- UART or USB->UART cable connected between DevBoard and PC for using the UART interface. Data speed is 115200 bps, format is 8N1.
- Your favorite UART Terminal Emulation program installed on the PC. For Microsoft Windows computers, PuTTY and TeraTerm have been tested and found working. Note that the VSOS Shell uses only the line feed character (0x0a) for line feed, and no carriage return (0x0d). For details on how to configure your terminal program see Chapter *Terminal Program Settings* of the document *VS1005 VSOS Shell*.
- This document assumes you know how to use the VSOS Shell. If you are not familiar with it, first read the document *VS1005 VSOS Shell*, available at <https://www.vlsi.fi/>

When all of this is in order, you are ready to test the devCrypt driver.



## 6 Testing devCrypt

### 6.1 Copying Test Files

Boot the Board to USB Mass Storage mode.

Copy the following files to the VS1005g system disk's main folder 'S':

- MainFolder/rypted.crp
- MainFolder/hello.mp3
- MainFolder/hellomp3.crp
- MainFolder/my.key
- MainFolder/plain.txt

Copy the following files to the VS1005g system disk's system folder 'S:SYS/':

- SYS/devCrypt.dl3
- SYS/LoadCrypt.dl3

## 6.2 Booting the Board

The devCrypt can be started either from the command line or with the help of LoadCrypt. LoadCrypt takes as its parameters a file name where the Secret Key is located. Note that having the Secret Key in a file is inherently unsafe. For more details on how to store the Secret Key, see Chapter 4.2.1, *Secret Key*.

First, we will boot up the board. Below are example printouts from the boot process:

```
Hello.  
VSOS 3.67 build Feb 15 2023 13:34:55  
VLSI Solution Oy 2012-2023 - www.vlsi.fi
```

```
Starting the kernel..  
Starting Devices...  
External SPI Flash  
2048 KiB
```

```
Installed system devices:  
S: 1920K SPI Flash c814, handled by FAT.  
Load drivers, config 0...  
Driver: RUN... SETCLOCK -193 80  
Driver: SDSD... D: SD/SD Card
```

```
Driver: UARTIN...  
Driver: AUODAC...  
Driver: INTTRACE...  
Driver: S:SHELL.AP3...  
VSOS SHELL  
S:>
```

After the boot-up, we will check currently existing drives, then we start devCrypt, and finally we check that we have gotten a new T: drive that is a decrypted mirror of the S: drive.

### 6.3 Starting the Driver

First, let's check what system drives we have available:

```
S:>::  
Currently installed system devices are:  
- D: 241G SD/SD Card, handled by FAT.  
- S: 1920K SPI Flash c814, handled by FAT.
```

In a real system, we would typically want to map the SD card driver D: to e.g. driver letter E:. However, in this case we will map the system driver S: to a decrypted drive T: as follows:

```
S:>loadcrypt my.key s t
```

#### NOTE!

LoadCrypt IS MEANT FOR DEMONSTRATION PURPOSES ONLY!  
DO NOT USE LoadCrypt IN YOUR COMMERCIAL APPLICATION!  
It is a completely unsafe program that cannot in any way  
hide the secret hex key that it provides to devCrypt!  
Actually, you will see the secret key a few lines below.  
For information on how to hide your secret key for your  
commercial application, please contact VLSI Solution directly,  
e.g. at sales@vlsci.fi!

Going to run:

```
driver +devCrypt HEXKEY:3fbedf805f4ea604b9c83d427a0756b0062af212d3dfb2a  
9832f002ee9bc2239 S T
```

We will verify that we got driver S: copied to T: with the following command:

```
S:>::  
Currently installed system devices are:  
- D: 241G SD/SD Card, handled by FAT.  
- S: 1920K SPI Flash c814, handled by FAT.  
- T: 1920K devCrypt S: to T:, handled by FAT.  
S:>
```

#### NOTE:

In this example we are mapping the system disk S: to T: . However, in many cases it is more useful to encrypt the larger data disks to e.g. protect audio data. In that case it would be more useful to map for instance the SD card disk D: to some other drive, e.g. E: . That would be done with the following command:

```
S:>loadcrypt my.key s t
```

With this example, you could encrypt for instance audio data in such a way that it would only play on a certain device, or on a certain group of devices.

## 6.4 Running Decryption Tests

First, we will verify that we see the exactly same files in the system drive S: and decrypted drive T:

```
S:>dir s:
- 1. CONFIG.TXT          4093 2023-11-10 09:45:12 config.txt
- 12. CRYPTED.CRP         68 2023-11-10 11:04:58 crypted.crp
- 4. HELLO.MP3           1695 2010-07-16 12:28:38 hello.mp3
- 15. HELLOMP3.CRP       1707 2023-11-10 11:04:58 hellomp3.crp
- 13. MY.KEY              72 2023-11-06 13:11:54 my.key
- 14. PLAIN.TXT           28 2023-11-10 11:00:54 plain.txt
- 8. SHELL.AP3           1646 2022-10-25 09:42:26 shell.ap3
D 9. SYS                  0 2023-11-10 11:12:52 SYS

S:>dir t:
- 1. CONFIG.TXT          4093 2023-11-10 09:45:12 config.txt
- 12. CRYPTED.CRP         68 2023-11-10 11:04:58 crypted.crp
- 4. HELLO.MP3           1695 2010-07-16 12:28:38 hello.mp3
- 15. HELLOMP3.CRP       1707 2023-11-10 11:04:58 hellomp3.crp
- 13. MY.KEY              72 2023-11-06 13:11:54 my.key
- 14. PLAIN.TXT           28 2023-11-10 11:00:54 plain.txt
- 8. SHELL.AP3           1646 2022-10-25 09:42:26 shell.ap3
D 9. SYS                  0 2023-11-10 11:12:52 SYS

S:>
```

Let's type out a plaintext file from drive S:

```
S:>type s:plain.txt
This is a plain text file.
S:>
```

Now, let's type out a decrypted file from drive T:

```
S:>type t:crypted.crp
This text file is encrypted using ChaCha20 encryption.
S:>
```

We will now verify that the encrypted file will appear as gibberish if you try to type it out from the system disk. Then we will verify that it appears correct if decoded through the devCrypt T: drive:

```
S:>more -x s:crypted.crp
00000000 23 9b 34 9c b3 43 32 75 9b 16 4d 82 52 8f 14 8c |#.4.³C2u..M.R...|
00000010 93 6f 03 fa 3d fd 5f 59 b9 b7 27 de 1d 2c 16 34 |.o.û=y_Y¹.'p.,.4|
00000020 ee 7b 16 2f c1 b0 93 42 d7 5c 69 be 34 90 d7 fb |î{./Á°.B×\i¼.×û|
00000030 51 e5 47 a4 53 85 cb e7 f8 53 72 86 17 80 03 c0 |Q&GOS.ËçøSr....Ä|
00000040 9b 7d 47 a2 |.}Gç|
00000044
S:>more -x t:crypted.crp
00000000 54 68 69 73 20 74 65 78 74 20 66 69 6c 65 20 69 |This text file i|
```

HH

```
00000010 73 20 65 6e 63 72 79 70 74 65 64 20 75 73 69 6e |s encrypted usin|
00000020 67 20 43 68 61 43 68 61 32 30 20 65 6e 63 72 79 |g ChaCha20 encry|
00000030 70 74 69 6f 6e 2e 0d 0a |ption...|
00000038
S:>
```

The same is reversibly true for plaintext files. The encrypted file system cannot show them correctly:

```
S:>more -x s:plain.txt
00000000 54 68 69 73 20 69 73 20 61 20 70 6c 61 69 6e 20 |This is a plain |
00000010 74 65 78 74 20 66 69 6c 65 2e 0d 0a |text file...|
0000001c
S:>more -x t:plain.txt
00000000 88 80 34 bb b7 39 c5 2c 55 2b e8 3d 49 2b e0 83 |..4»·9Å,U+è=I+à.|
00000010
S:>
```

Now, let's try to play an audio file. Below we check what audio files are available on the plaintext S: disk, then play them:

```
S:>s:
S: SPI Flash c814
S:>dir -a
- 4. HELLO.MP3 0:00.4 22050 1 2010-07-16 12:28:38 hello.mp3
S:>playfile hello.mp3
Playing 'hello.mp3'
~0503'Hello
~0505'Panu-Kristian Poiksalo
~0504'VSDSP Testing
~0506'
[00:00]Decode finished.
S:>
```

Then we check what audio files are available on the encrypted T: disk and play them:

```
S:>t:
T: devCrypt S: to T:
T:>dir -a
- 15. HELLOMP3.CRP 0:00.4 22050 1 2023-11-10 11:04:58 hellomp3.crp
T:>playfile hellomp3.crp
Playing 'hellomp3.crp'
~0503'Hello
~0505'Panu-Kristian Poiksalo
~0504'VSDSP Testing
~0506'
[00:00]Decode finished.
T:>
```

## 6.5 Testing Decryption Speed

First, let's see how long it takes to read a very short and a long file from an SD card with and without decryption. The speeds are measured in a working system with the AUODAC.DL3 audio driver running.

```
S:>dir d:
- 2. 01_SLE~1.OGG      4830256 2023-02-13 03:08:14 01_SleepingBag.ogg
- 20. 01_SLE~1.CRP     4830268 2023-11-06 13:14:50 01_SleepingBag.crp
- 21. HELLO.CRP        28 2023-11-08 13:19:58 Hello.crp
- 22. HELLO.TXT        16 2023-02-14 11:57:26 Hello.txt

S:>setclock -l 90 86 -v
SetClock running on VS1205g, clocks:
XTALI 12.288MHz, CLKI 86.016MHz, limit 93.000MHz, src PLL
RAM Delay 1, POR on
CVDD 1.800V(22), IOVDD 3.30V(25), AVDD 3.60V(28)
UART nominal 115200 bps, real 114841 bps (0.3% error), reg 0x006b
SPI0(E) 8.6 Mbit/s, SPI1 43.0 Mbit/s, NAND 43.0 MByte/s, SD 10.8 MByte/s
[... multiple irrelevant lines removed ...]

S:>time more -q d:nothing
E: Cannot open "d:nothing"!
errCode -1, time: 0.053s

S:>time more -q d:01_SleepingBag.ogg
errCode 0, time: 0.567s

S:>time more -q e:01_SleepingBag.crp
errCode 0, time: 4.819s

S:>setclock -v 60
SetClock running on VS1205g, clocks:
XTALI 12.288MHz, CLKI 61.440MHz, limit 93.000MHz, src PLL
RAM Delay 0, POR on
CVDD 1.700V(18), IOVDD 3.30V(25), AVDD 3.60V(28)
UART nominal 115200 bps, real 114841 bps (0.3% error), reg 0x006b
SPI0(E) 4.4 Mbit/s, SPI1 30.7 Mbit/s, NAND 30.7 MByte/s, SD 7.7 MByte/s
[... multiple irrelevant lines removed ...]

S:>time more -q d:nothing
E: Cannot open "d:nothing"!
errCode -1, time: 0.062s

S:>time more -q d:01_SleepingBag.ogg
errCode 0, time: 0.773s

S:>time more -q e:01_SleepingBag.crp
errCode 0, time: 6.879s
```

So, now we can calculate the following table:

devCrypt decoding speed		
Processor speed	Encrypted	Read speed
86.016 MHz	X	989 KiB/s
86.016 MHz		9177 KiB/s
61.440 MHz	X	691 KiB/s
61.440 MHz		6634 KiB/s

From this we can estimate that read+decoding speed is approximately 85 clock cycles per byte, including all VSOS overheads.

## 7 Encrypting Files

The encryption process consists of the following parts:

- Choosing or generating a Secret Key.
- Encrypting files using the Secret Key.

Both of these operations can be done using VLSI Solution's encryptor and key generator program `vlsiencryptor.exe`. This application is provided along with this package both as C source code and as a Windows executable file. As with other VLSI Software, it is entirely free to use and modify as long as you use it for the benefit of any VLSI Solution's Integrated Circuits, e.g. VS1005 or VS1010.

The user interface for `vlsiencryptor` is a command line tool that shows some help with the "-h" parameter:

```
C:\devCrypt\vlsiencryptor\>vlsiencryptor -h
VLSIEncryptor 2023-11-14
Usage:
vlsiencryptor.exe [-h] [-s salt -g keyfile | -k keyfile -i inlistfile]
-s salt Provide random salt data
      To create good salt data, push random keys to make salt entropy
      good enough. Example:
      hrifyuntknwtnnyt5t4n9t83tm3o2mtcy3m4tm34tcmttmcu8t49y8mv893t3JIFEOW
      DO NOT COPY THIS AS IS BUT MASH YOUR OWN ON YOUR KEYBOARD!
-g keyf Generate secret key file keyf (e.g. my.key)
-k keyf Use secret key file keyf for encryption (e.g. my.key)
-i ilf  Encrypt files from inlistfile
      Each line in inlistfile should look like the following:
      hello.mp3 hellomp3.crp
      The previous line tells to encrypt hello.mp3 to hellomp3.crp
      Be careful not to overwrite source files: the file names on
      the line must be different!
-h      Show this help
```

### 7.1 Choosing or Generating a Secret Key

Before you can encrypt files, you need to choose a Secret Key. Depending on your application, this Secret Key may be universal, or it can be selected on a device-to-device basis. In any case, keeping the Secret Key secret is a requirement for encryption to work. If the secret key is lost, no files can be opened. On the other hand, if the secret key is leaked, all files encoded with that key will be decodable by anyone holding the key.

To make for a good, secure 256-bit key, some random input is expected by the user. The random input is provided with the so-called "salt" parameter. For the salt parameter, the user should mash random keys enough so that it appears random enough.

An example of how to generate a key properly is shown below:

```
C:\devCrypt\vlsiencryptor\>vlsiencryptor -g my.key -s 832gr42jgfvJ0IFEhugnj
bdjlkdtmghvgehmtglstdmgfsdsnuiogrewefw5
Estimated potential key entropy 290 bits: Good.
HEXKEY:4dba32e7eda33ede9935ecd59e6d4b1225c7a7e1b80c807c40e81051c80b2a87
Key written to file "my.key".
```

**NOTE!**

This demonstration already has an example key. Try the demonstration out before generating your own key!

**NOTE!**

Do not use the same salt parameter as shown for the “-s” option above. Generate your own salt by mashing random keys on your keyboard!

**NOTE!**

The key generator refuses to overwrite an existing key file. If you want to replace a key with the same name, delete it first, e.g. with command:

```
C:\DEVCRYPT\VLSIENCRYPTOR\>del my.key
```

However, do note that if you delete a key, you can never recover it! It is usually better to create new keys with new names and keep the old keys in store just in case you would need them later.

If your salt parameter is too short, or if it is too repetitive, or if it is missing, the key generator will refuse to create an unsecure key, as shown below:

```
C:\devCrypt\vlsiencryptor\>vlsiencryptor -g my.key -s 832gr42jgfvJ0IFEhugn
Estimated potential key entropy 98 bits: BAD! REFUSING TO WRITE!
vlsiencryptor.exe: Failed to create key file "my.key"
```

If you have a 256-bit random key generator that you feel is more secure than vlsiencryptor, you may use it to create the Secret Key file. The Secret Key file consists of a single line that looks as below:

```
HEXKEY:43d20bf65fea5d8429f3c4f537e147ef822e948426081f0a65f8b3c709e4f77e
```

By replacing the hexadecimal number on the line, you can modify the Secret Key to be anything.



## 7.2 Encrypting Files

To encrypt files using your Secret Key, you may run the following instruction:

```
C:\devCrypt\vlsiencryptor\>vlsiencryptor -k my.key -i infile.txt
Reading key OK
Encrypt 'encrypted_plaintext.txt' to 'encrypted.crp'
Encrypt 'hello.mp3' to 'hellomp3.crp'
Encrypt 'hello.txt' to 'hellotxt.crp'
Encrypt 'text.txt' to 'text.crp'
```

Here, “my.key” is either the example key file provided with this package, or a key generated in Chapter 7.1, *Choosing or Generating a Secret Key*.

“infile.txt” is a text file that contains the names of the files to be encrypted. In this example, “infile.txt” looks as follows:

```
encrypted_plaintext.txt encrypted.crp
hello.mp3 hellomp3.crp
hello.txt hellotxt.crp
text.txt text.crp
```

### NOTE!

Encryption will increase the size of each file by 12 bytes because an auto-generated 96-bit Nonce is added to the end of each file.

### NOTE!

Because of the (partially) randomly generated Nonce, encrypted files will be different each time you regenerate them. This increases security. Even if different in their encrypted forms, the files will still always decrypt to the original files.

## 8 Latest Document Version Changes

This chapter describes the latest and most important changes to this document.

### **Version 1.01, 2023-11-17**

This is a bug and documentation fix.

- Fixed a bug where incorrect data would be read if file handles to both a plaintext file and decrypted file were open at the same time on an SD card.
- Added speed measurement Chapter 6.5, *Testing Decryption Speed*.

### **Version 1.00, 2023-11-15**

First release.

## 9 Contact Information

VLSI Solution Oy  
Entrance G, 2nd floor  
Hermiankatu 8  
FI-33720 Tampere  
FINLAND

URL: <http://www.vlsi.fi/>  
Phone: +358-50-462-3200  
Commercial e-mail: [sales@vlsi.fi](mailto:sales@vlsi.fi)

For technical support or suggestions regarding this document, please participate at  
<http://www.vsdsp-forum.com/>

For confidential technical discussions, contact  
[support@vlsi.fi](mailto:support@vlsi.fi)