Hochschule
Augsburg University of
Applied Sciences

Faculty of
Computer Science

Seminar Paper – Software Systems

Area of Study
Computer Science (Master)

Adam Lang (953321)
Bernd Fecht (954020)
Hannah Lisa Walkiw (952681)
Julian Hillesheimer (954018)

Comparison of Five TLS Libraries

Examiner: Prof. Dr.-Ing. Dominik Merli
02.07.2017

Hochschule für angewandte
Wissenschaften Augsburg
University of Applied Sciences

An der Hochschule 1
D-86161 Augsburg

Telefon +49 821 55 86-0
Fax +49 821 55 86-3222
www.hs-augsburg.de
info@hs-augsburg.de

Faculty of Computer Science
Telephone: +49 821 5586-3450
Fax: +49 821 5586-3499

Writers of the seminar paper:
Adam Lang
Bernd Fecht
Hannah Lisa Walkiw
Julian Hillesheimer

Contents

1. BoringSSL

1.1. Motivation

OpenSSL is a widely used software library to secure communications over computer networks. In 2014 Google started a fork of OpenSSL named BoringSSL. [Borb] Google is a company which is well known for developing good software and libraries like Google Guava. So, therefore it is interesting to see how Google changed and improved the SSL library.

1.2. What is BoringSSL?

Google started BoringSSL because of the huge effort to patch OpenSSL and because of the OpenSSL development restrictions. [Borb] So you can say, "BoringSSL is a fork of OpenSSL that is designed to meet Google's needs." [Bora] This is also the point why Google doesn't recommend to use their library in own projects because there will be no guarantee of a stable API.

1.2.1. Comparison between OpenSSL and BoringSSL

"BoringSSL is an OpenSSL derivative and is mostly source-compatible [...]." [Bord] So because of that in the most cases there is no need to change for BoringSSL support. But there are some removed APIs and some major API changes that will need little changes if used. The changes are listed in the following enumeration.

- size_t instead of int for indices and lengths
- increment reference count must know use the corresponding *_up_ref function
- reworked error codes
- EVP_PKEY_HMAC was removed
- EVP_PKEY_DSA is now deprecated

- Changed return values (1 on success and 0 on failure)
- BoringSSL calls *pthreads* and the corresponding Windows APIs internally and is always thread-safe where the API guarantees it
- *Ioctl-style* functions have been replaced with proper functions

A list of all changes is mentioned in [Bord].

In contrast to OpenSSL, BoringSSL offers public headers with API documentation in comments. The documentation is also available online.

1.2.2. How to Use BoringSSL?

OpenSSL is widely used in linux distributions. For the most linux distributions packages for OpenSSL are offered. For BoringSSL exist no packages. Therefore it is necessary to build BoringSSL from source. The process of building the library is well documented by Google. On linux only three steps are necessary. First of all, get the source code, for example by git clone the repository. After that it is only necessary to run cmake followed by make. To speed up the build process ninja could also be used. For building the library you need Google's language Go. So on some linux systems you must install it first. For Debian the package golang is offered.

Building BoringSSL on Windows is not that easy like on linux. First of all there must be installed a number of different prerequisites. After that there are some pitfalls which must be avoided. Building BoringSSL as shared library on Windows wasn't successfully at all.

Building BoringSSL for iOS, Android or embedded ARM was not tested but is also documented by Google.

After a successful build there are three more steps necessary to compile and run source code which uses BoringSSL. Like using OpenSSL or another library the path for the header files and the library files must be specified. In contrast to OpenSSL it is necessary for BoringSSL to set the linker option –lpthread. This is necessary because BoringSSL uses pthreads.

1.3. AES Implementation

First of all, it was necessary to create a key. A key was created by the function EVP_BytesToKey. For this function, it was necessary to specify the type of AES and the hashing algorithm, also a salt, initialization vector, and some random data are needed. After that with the function EVP_CIPHER_CTX_init and the created key to contexts for the encryption and decryption are created.

For the encryption process the context, the key and the plaintext are being needed. To encrypt the text the three functions EVP_EncryptInit_ex, EVP_EncryptUpdate and EVP_EncryptFinal_ex are used. For decrypting the ciphertext the functions EVP_DecryptInit_ex, EVP_DecryptUpdate and EVP_DecryptFinal_ex are offered. "/dev/urandom" was used to generate random data. The functions from *time.h* are being used for measuring the execution time of the encryption and decryption. The source code is depicted in the appendix A.

1.4. RSA Implementation

To use RSA a public and private key are needed. Therefore the command line interface was used to generate two public and two private keys with the key length 2048 and 4096 bits. For the generation the commands openssl genrsa -des3 -out private.pem 2048 and openssl rsa -in private.pem -outform PEM -pubout -out public.pem are used. [Borc] Like depicted in the appendix B the keys are included as char array in the source code. For encrypting and decrypting with RSA a so called RSA object must be created. With the functions PEM_read_bio_RSA_PUBKEY and PEM_read_bio_RSAPrivateKey and the corresponding char arrays the RSA object for the encryption and decryption can be created. For encrypting a text the RSA object with the public key, the char array with the text and a padding are used by the function RSA_public_encrypt. For decrypting the ciphertext the function RSA_public_decrypt can be used with the RSA object which was created with the private key, the ciphertext as char array and the padding which was also used during the encryption.

For benchmarking the encryption and decryption the same mechanism was used like in the AES implementation. Also, the same function for generating random data was used.

For successful encryption and decryption, the size of the plaintext and ciphertext must be smaller than the size of the key length minus the size of the padding. Is the text larger it becomes necessary to encrypt and decrypt the text block by block.

1.5. Time Measurement

For time measurement the functions from *time.h* were used. To get more accurate results the average time of 10.000 iterations was used for small data. For big data only 10 iterations was used. For small data the time was measured

in microsecond and for big data in milliseconds.

The time measurement was executed in a linux debian virtual machine on an Intel Xeon E3-1231 v3 with 4x 3.40 GHz. This CPU supports Intel AES-NI but this doesn't work in the virtual machine. So, the time measurement was executed without the hardware acceleration AES-NI.

The first three graphs show the results of the libraries BoringSSL, LibreSSL and OpenSSL for the AES with SHA256 implementation.

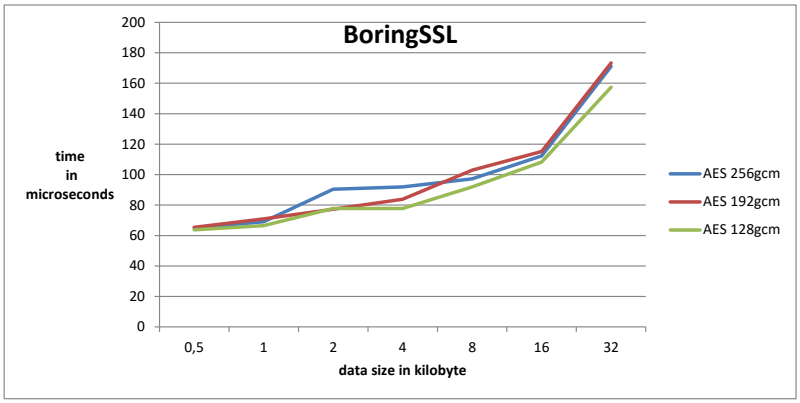

Figure 1.1.: Time measurement of BoringSSL with AES 128gcm, 192gcm and 256gcm

As you can see in the graph 1.1 AES 256gcm is slightly slower than AES 192gcm which therefore is also slightly slower than AES 128gcm.

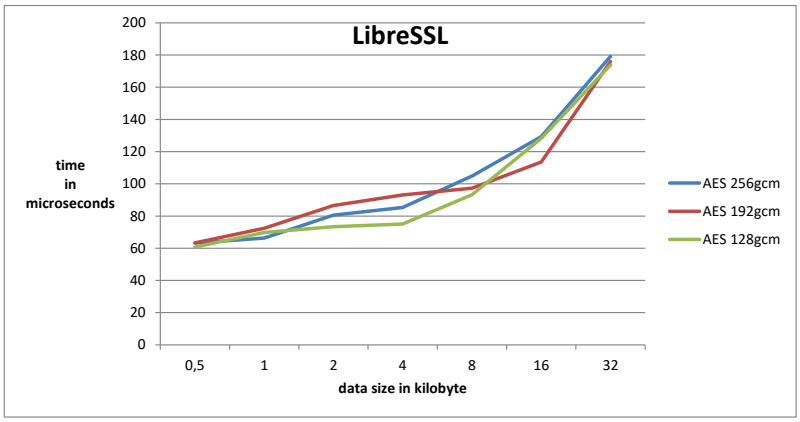

Figure 1.2.: Time measurement of LibreSSL with AES 128gcm, 192gcm and 256gcm

In LibreSSL AES 192gcm is for data size less than 8 kilobyte slower than

AES 128gcm and AES 256gcm, like the graph 1.2 depicts. For data greater than 8 kilobyte AES 192gcm is the fastest.

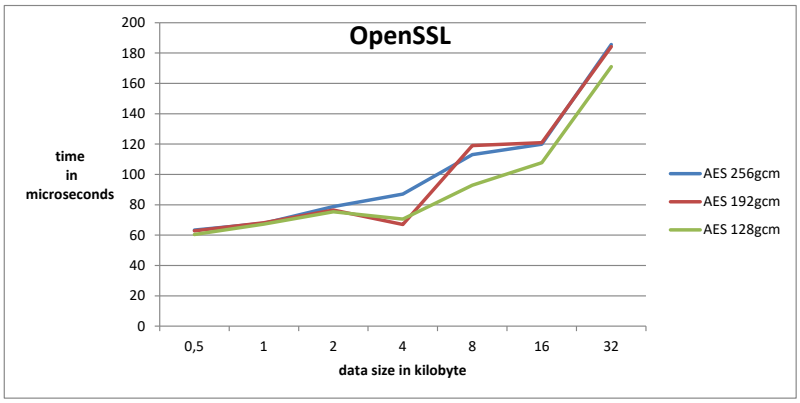

Figure 1.3.: Time measurement of OpenSSL with AES 128gcm, 192gcm and 256gcm

The graph 1.3 shows that in OpenSSL AES 128gcm is the fastest. AES 192gcm is for the most data size as fast as AES 256gcm.

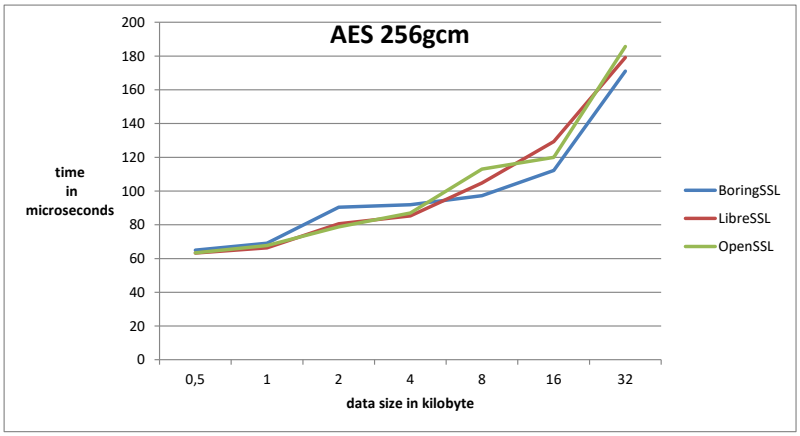

Figure 1.4.: Comparison between the libraries BoringSSL, LibreSSL and OpenSSL for AES 256gcm

As you can see in the graph 1.4 LibreSSL and OpenSSL are nearly on the same level of speed. For data sizes up to 4 kilobyte BoringSSL is slightly slower. But for data size above 8 kilobyte it is faster than LibreSSL and OpenSSL.

For encrypting and decrypting a huge amount of data OpenSSL and LibreSSL are on the same level. For this task BoringSSL is slower, like the graph 1.5 depicts.

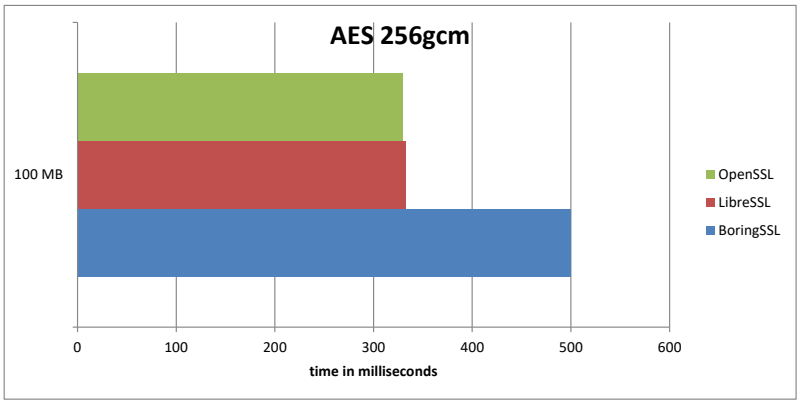

Figure 1.5.: Comparison between the libraries BoringSSL, LibreSSL and OpenSSL for AES 256gcm while encrypting and decrypting 100 MB of random data

The next two diagrams depict the difference between the three libraries for the RSA implementation with a key length of 2048 and 4096 bit.

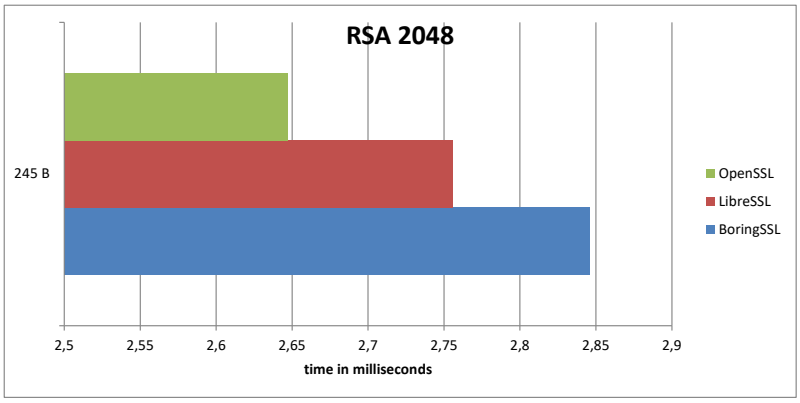

Figure 1.6.: Time measurement of RSA with 2048 Bit key length

As you can see in the graph 1.6 and the graph 1.7 the three libraries are nearly on the same level. In both cases OpenSSL is slightly faster than BoringSSL and LibreSSL.

Use the results of the time measurement carefully. Using a virtual machine and *time.h* for benchmarking may be not the best conditions for getting high accurate results. While micro benchmarking there are countless variables that come into play like locality of reference, effects of caches, memory bandwith, compiler inlining, operating system schedulers, operating system background processes and much more. Maybe a well-defined profiling outside of a virtual

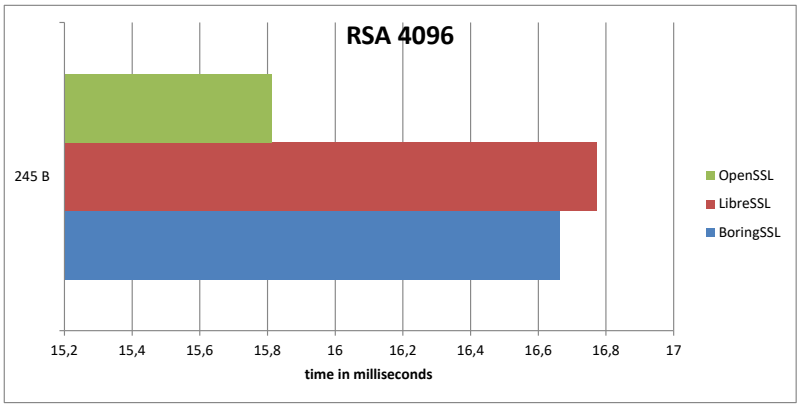

Figure 1.7.: Time measurement of RSA with 4096 Bit key length

machine will provide more appropriate and accurate results.

1.6. Conclusion

BoringSSL is as good as OpenSSL or LibreSSL. You do not need much effort for porting source code to support BoringSSL but you well get a better documented API than OpenSSL or LibreSSL. Google also cleaned up the API and improved the library, e.g. error handling. BoringSSL is used in Android and Chrome so until know it is still an ongoing and active project. The speed of BoringSSL is slightly on the same level as OpenSSL and LibreSSL. So when to use BoringSSL? If the missing guarantee of a stable API is no pitfall BoringSSL would be a good choice. Is there no experience with OpenSSL or LibreSSL it might be a better choice to choose another library because there is less or no documentation or examples besides the documentation of the BoringSSL API.

2. LibreSSL

2.1. Motivation

Nowadays there are many TLS libraries available that provide security mechanisms to safely communicate across networks. One of the most common libraries is OpenSSL. In 2014 the OpenBSD Project forked OpenSSL. [Liba] The main goals of this action were, to create a more modern and secure version of the codebase. Of course, this will raise up the questions, what's the difference between OpenSSL and the forked implementation of LibreSSL and how can you use it? These questions will be answered and discussed in the following pages.

2.2. What is LibreSSL?

As mentioned, LibreSSL was developed by the OpenBSD-Team, that is known for its Operating-System OpenBSD. The ambition of refactoring the OpenSSL code started as a response of the Heartbleed security vulnerability in April 2014. [Libg]

2.2.1. Comparison between OpenSSL and LibreSSL

Especially the OpenBSD-Team complained about the "[...] overcomplex and hard to use [...]" implementation of OpenSSL and that "[...] the best documentation is the **openssl** application code [...]" itself. [Libh] So they developed LibreSSL, to get rid of these problems. LibreSSL wanted to make the OpenSSL codebase "easier to audit, understand and repair". [Libb] Also they tried to "apply best-practice development processes [...]" such as Code Reviews, frequent releases and an open development process. [Libb] Furthermore LibreSSL should fix legacies of OpenSSL, such as removing obsolete or broken features and fixing OS-specific problems.

2.2.2. How to use LibreSSL?

If you want to use LibreSSL on a Linux distribution there are two ways you can take. The first one is to use the OpenBSD-OS itself because LibreSSL is already preinstalled and replaces OpenSSL in this OS. So you don't have to take any further steps to use the library. On all other Linux distributions, you can use the portable version of LibreSSL, available on Github. [Libc] When building from git, you have to ensure that the packages automake, autoconf, git, libtool, perl and pod2man are installed. After cloning the project, you have to run ./autogen.sh in the repository folder, to prepare the source tree. After that, you can use make, cmake or ninja to build and install the package. [Libc]

If you want to run LibreSSL on Windows, LibreSSL supports the mingw-64 toolchain. With that you can use GCC or Clang as a compiler on Windows. It supports 32-bit and 64-bit systems. To use LibreSSL along with Windows Visual Studio, there exists a script that uses mingw-64 to build LibreSSL and generates ready-to-use .DLL and static .LIB files, you can add to your projects. There exists pre-build Windows binaries too if you don't want to use mingw-64. [Libf]

To compile programs with LibreSSL, you simply add the include files and libraries to the compiler options. If you run the program and you have system-wide installation of LibreSSL, there are no further steps required, otherwise, you must inform the dynamic linker to add the OpenSSL libraries. Assuming, that LibreSSL is installed under /opt/libressl the command will be: LD_LIBRARY_PATH="/opt/libressl/lib" ./foo. [Libd]

2.3. AES Implementation

The implementation of an AES encryption and decryption with LibreSSL is similar to OpenSSL and uses the evp.h file, where many functions are declared for SSL. First of all, you have to declare which cipher type, for example AES-192 or AES-265 and which digest like SHA256 should be used. Additionally, with the salt, and some random data (password) these informations are used to create a key and an initialization vector. The function that creates this key is called EVP_BytesToKey().

To encrypt and decrypt some data, of course there is some plaintext needed, as well.

After creating the key and the initialization vector, a new EVP_CIPHER_CTX context must be created and has to be passed together with the key, the initialization vector and the cipher type to the EVP_EncryptInit_ex() function, that initializes the encryption context, mapping the key, initialization

vector and cipher type to it. With the two functions EVP_EncryptUpdate() and EVP_EncryptFinal_ex() the encryption of the plaintext is complete. As result the encrypted ciphertext and the ciphertext-length is provided by the function.

To decrypt the message again, you have to follow similar steps. First of all, the context must be created again, because it gets freed after encryption. The next function calls are EVP_DecryptInit_ex(), EVP_DecryptUpdate() and EVP_DecryptFinal_ex(). These function calls are very similar to the encryption functions but instead of the plaintext and the length of it of course they get the encrypted text and its length as input variables.

2.4. RSA Implementation

Equally to the AES implementation, the first step is to create a key – or in this case a key-pair because of the asymmetric encryption. For that reason there is an command line call provided by OpenSSL to create that keypair: [Borc]

- Creating a 2048-bit private key: openssl genrsa -des3 -out private_2048.pem 2048
- Generating the 2048-bit public key out of the private key: openssl rsa -in private_2048.pem -outform PEM -pubout -out public_2048.pem

There are two key pairs used in this implementation: the 2048-bit key and the 4096-bit key. For encryption, the public key is used and for decryption the private key is used. After creating the keys, they must be opened and written out to an EVP_PKEY object. The function for the public key is PEM_read_PUBKEY() and for the private key PEM_read_PrivateKey(). When using the PEM_read_PUBKEY() function, you have to be careful, that your public key starts with —BEGIN PUBLIC KEY— and ends with —END PUBLIC KEY— and not with —BEGIN RSA PUBLIC KEY— and —END RSA PUBLIC KEY—, because for that annotation, another (misleading, nearly same named) function must be used instead, that is called PEM_read_RSAPublicKey(). [Libe] Not knowing this, leads programmers (including me) to strange error messages and aborts the encryption.

Furthermore, we need a plaintext to be encrypted and decrypted again of course and a type of padding, in this case RSA_PKCS1 padding. The padding is used, to fill up the message with padded bytes to fit the block size of 256 or 512 bytes (256 bytes for 2048 bit key and 512 bytes for 4096 bit key).

After creating the EVP_KEY object, that holds the public key, it can be used to encrypt the plaintext. Therefore, a EVP_PKEY_CTX context has to be created by passing the EVP_KEY object to it. Now the padding is passed to the context with EVP_PKEY_CTX_set_rsa_padding(). After that, the encryption can be initialized, using the function EVP_PKEY_encrypt_init() and determine the size of the final encrypted message, calling EVP_PKEY_encrypt(), passing the context and the plaintext and its length to the function. After that, this function is called again, but this time, the length of the ciphertext and the ciphertext pointer have to be passed to the function. As a result, we get the encrypted text.

To decrypt that message again, we have to follow nearly the same steps again, but in the contrary to the encryption, the EVP_KEY object, that holds the private key of course must be passed to the EVP_PKEY_CTX context, the EVP_PKEY_decrypt_init() function must be called and the encrypted text and its length must be passed to the function EVP_PKEY_decrypt(). As result we get de decrypted message again.

As a supplement it is to say, that only messages with a size, equally or smaller than the key-size in bit divided by eight minus the size of padding (in this case minus eleven) can be used. For all bigger messages, the message needs to be encrypted and decrypted block by block.

2.5. Time Measurement

For the time measurement the same functions are used, that got declared by Julian Hillesheimer in the BoringSSL part, so that all time-measurements base on the same function calls. So the functions from to time.h were used too. To get a more accurate result, the average time of 10.000 iterations for small data and the average of 10 iterations for big data is being calculated. The time is in microseconds.

The time measurements were executed in a Linux Debian virtual machine on an Intel i5-3317U CPU with 2x 1,7 GHz. In the virtual machine, the hardware acceleration with Intel AES-NI is used.

The first three graphs show the results of the LibreSSL for the AES with SHA256 implementation. To get a more accurate result, the average time of 10.000 iterations is being calculated.

As you can see in the graph 2.1, the time used to encode and decode the message isn't really differing from the cipher type AES256, AES192, AES128. This may be because the data used, is very small and the hardware acceleration with Intel AES-NI is very fast. The graph however tells us one thing and that's the fact, that with increasing size of the plaintext, the time needed rises

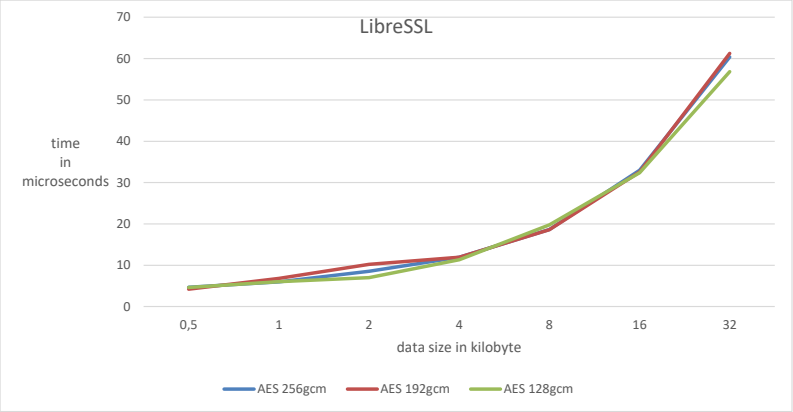

Figure 2.1.: Time measurement of LibreSSL with AES 128gcm, 192gcm and 256gcm

exponentially. To verify, if the different cipher types don't differ in their used time, it is necessary to measure the time with bigger data.

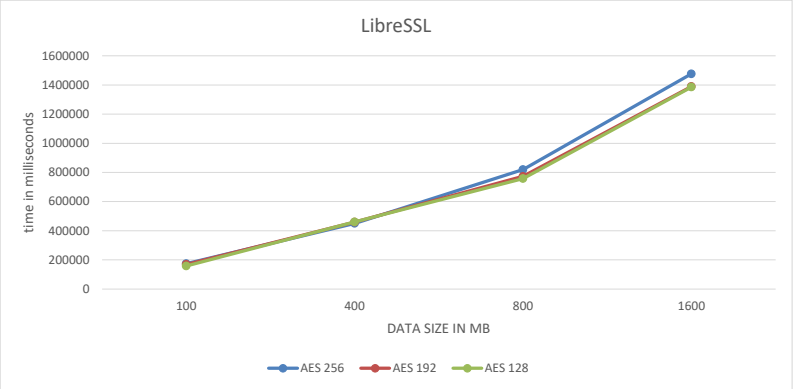

Figure 2.2.: Time measurement of LibreSSL with AES 128gcm, 192gcm and 256gcm and huge data

So in this graph 2.2, we used data from 100 MB up to 1600 MB. To get a more accurate result, the average time of 10 iterations is being calculated. The graph could be interpreted, that the AES256 is slightly slower, than the AES192 and AES128 cipher type, but this is just a nuance and is not really proofing that state. The time needed by AES192 and AES128 are very similar, so that the two lines aren't clearly differentiating from each other. So in conclusion it's to say, that for the runtime, it doesn't matter if you use AES256, AES192 or AES128 (assuming Intel AES-NI is enabled).

The next interesting thing to measure is, how fast other SSL-libraries related

to LibreSSL are. Since the source-code uses just normal OpenSSL functions, it is no problem to run the exactly same code with OpenSSL.

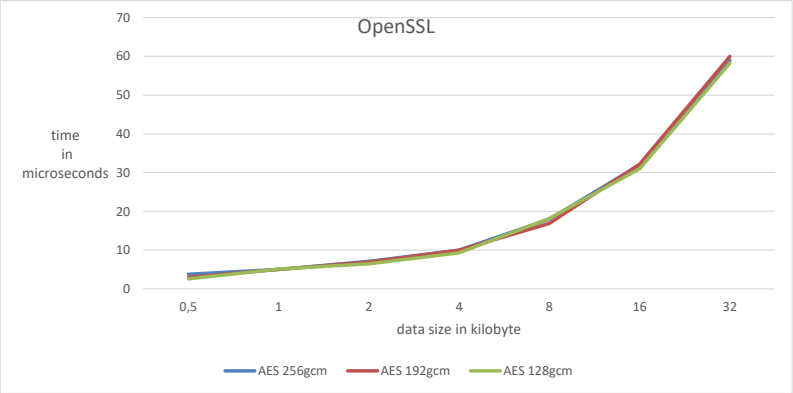

Figure 2.3.: Time measurement of OpenSSL with AES 128gcm, 192gcm and 256gcm

The same data sizes are used as in the first graph 2.3. To get a more accurate result, the average time of 10.000 iterations is being calculated. As we can see, the time for the different cipher types don't differ too much. Comparing the first graph and this one, we can assume, that the time used doesn't differ that much. To verify this assumption, we need a bigger data-size again.

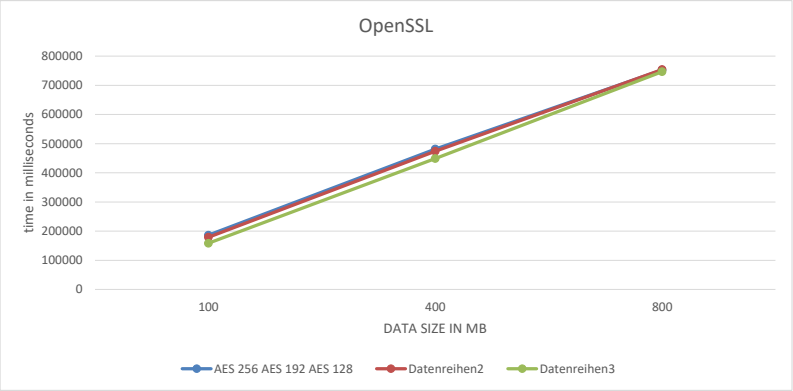

Figure 2.4.: Time measurement of OpenSSL with AES 128gcm, 192gcm and 256gcm and huge data

As we can see in the graph 2.4 for AES, the two SSL-libraries need nearly the same time for their encryption and decryption. However, OpenSSL is slightly faster than LibreSSL.

The next two graphs show the time measurements with RSA and a key-length of 2048 and 4096 bit.

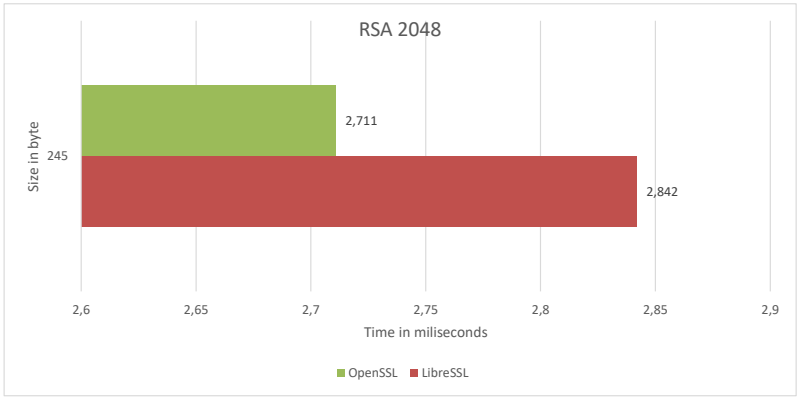

Figure 2.5.: Time measurement of RSA with 2048 bit key length

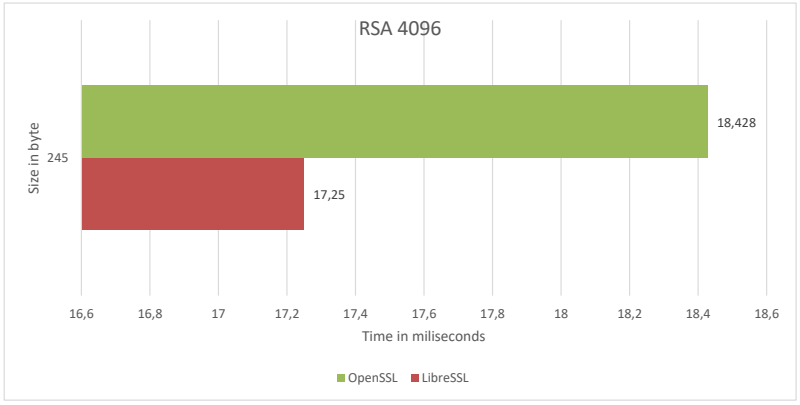

Figure 2.6.: Time measurement of RSA with 4096 bit key length

As you can see in these two graphs 2.5 and 2.6, the time doesn't really differ between OpenSSL and LibreSSL. The fact, that OpenSSL is faster with the 2048 bit key and slower with the 4096 bit key comparing to LibreSSL could be just because the hardware was busier at that time. So the two libraries are nearly on the same level, comparing their time for RSA.

As a result, it is to say, that all measurements were taken in a virtual machine. Therefore, the data could be different using the libraries outside of a VM. There are many variables that could falsify the measured time, like cache effects, schedulers, background processes, to name just a few of them. In summary, the two different libraries didn't differ that much comparing their speed.

2.6. Conclusion

As a conclusion it is to say, that I couldn't find a proper API documentation about LibreSSL at all and had to use the documentation of OpenSSL and because of that, the function calls are the same as in OpenSSL. LibreSSL provides a more stable usage compared to OpenSSL, because they refactored and reduced the codebase and cleaned up old legacies. Accordingly to the developers of LibreSSL, their library is more secure than OpenSSL by preventing more accurately that security risks are contained in their library. [Libh] So LibreSSL is a real alternative to OpenSSL and because it stays API-compatible with OpenSSL it is no problem to adapt OpenSSL projects to run with LibreSSL.

3. mbed TLS and MatrixSSL

3.1. Motivation

The purpose of the following paper is a comparison of different TLS libraries on various hardware platforms. The work has been divided among four group members and the results are collected in this paper.

3.2. Hardware Platform

All the tests of the following chapter were executed on a laptop with following properties:

- Manufacturer: Acer
- Model: TravelMate 8372
- CPU: Intel i3-370M @ 2.4 Ghz Dualcore, 4 Threads [Matc]
- RAM: 8GB
- Operating System: Ubuntu 14.04 LTS, Kernel: 3.19.0-61-generic

3.3. Selecting the Library

For making the decisions of the libraries for the implementation, the libraries mbed TLS, GNUtls, libsodium and MatrixSSL were tested.

3.3.1. Gnu TLS

Already during the installation of gnutls-3.5.0 it turns out that this library is an exentsive one. [Matg] During the installation process it was obligatory to download and install following libraries in order to conclude all dependencies:

- gmp-6.1.2

- libffi-3.2.1
- libtasn1-4.3
- nettle-3.3
- p11-kit.0.23.2

The task of implementing testcases resulted in a vast search of finding appropriate API-functions and the API-functions seemed to have many interdependencies. For the simple task of implementing testcases the GNUtls Library therefore is considered as oversized.

3.3.2. Libsodium

For this reason, another library with the name libsodium was tested. In the documentation the library is introduced as "Sodium is a modern, easy-to-use software library for encryption, decryption, signatures, password hashing and more." [Matd]

Setting up a testcase with libsodium is straightforward, but on the test-hardware the function-call leads to the abort of the program:

```
if (crypto_aead_aes256gcm_is_available() == 0) {
        abort(); /* Not available on this CPU */
}
```

Listing 3.1: Test

If the above-mentioned check is disabled, the program gets stopped from the OS with the message "illegal instruction" if the aes256gcm algorithm is used. The encryption can be done with the crypto_aead_chacha20poly1305_encrypt function without any problems, but it turns out, that libsodium only provides an aes256gcm algorithm with hardware assistance. As the implementation of a testcase with use of the aes-256-gcm algorithm is obligatory, the library cannot be used at this point.

3.3.3. mbed TLS

Mbed TLS is a library developed by ARM. [Mata] It is a configurable library which can be stripped down to make it appropriate for even less powerful embedded devices. It is not as extensive as the GNUtls library but it provides all the algorithms to implement the testcases. The API-calls are documented in the header files and do not have too many dependencies among each other. So mbed tls results as appropriate to implement the aes256gcm testcases. For

implementing the RSA algorithm it turns out, that mbedtls is not enough documented. In the documentation it contains an error which refers to the public key and private key creation. The built-in application gen_key [Matf] which is referenced in the instructions [Matb] to generate keys with which the workflow of an encryption/decryption should be achieved, generates keys in a wrong format. These can not be read and the program terminates with a parser error. The program expects keys which are sourrounded by a text as —BEGIN PUBLIC KEY— and —END PUBLIC KEY— but if editing the keys in an editor the encryption is possible but the decryption process fails. As there is no documentation, except the source files, about how exactly the keyfiles should be, the part of implementing a RSA performance test has been done with another library.

3.3.4. MatrixSSL

The library is called MatrixSSL 3.9.3 Open [Mate] and brings already a performance test program which can be adapted for the given testcase. This program is located in the matrixssl-3-9-3-open/crypto/test/rsaperf directory and has been customized to carry out the tests with the same key length, data size and number of iterations as in the tests of the other group members.

3.4. Performance Tests

For the implementation of the performance tests a declaration regarding data size, key length, algorithm and number of iterations was made among the group members. Therefore, each test has been executed with the use of different libraries on different hardware. These tests are described in the sections of the other contributors of the document.

To restrict side effects from memory access time to a hard disk drive, the test data is loaded into memory. In order to eliminate other side effects like compiler optimization, CPU-caching and other nonpredictable interferences, the buffer with the test data was filled with randomly generated data. To achieve this, in a Linux operating system the random device /dev/urandom can be used.

3.4.1. AES-GCM Performance Test with mbed TLS

The following outputs are the performance tests for mbed TSL aes-gcm encryption. As is visible in the console output, different data sizes are encrypted with the use of the aes_gcm algorithm with different key lengths. The tests

are executed with 10.000 iterations to get a more accurate result, the last test which has a bigger amount of input data is iterated 10 times. The source code of the application is attached in the appendix C. For executing the tests, the line #define AES_GCM must be uncommented.

AES_GCM_128

- 512 bytes: 0.009534 ms
- 1024 bytes: 0.018621 ms
- 2048 bytes: 0.034807 ms
- 4096 bytes: 0.068340 ms
- 1024000 bytes: 17.285999 ms

AES_GCM_192

- 512 bytes: 0.010419 ms
- 1024 bytes: 0.019596 ms
- 2048 bytes: 0.037094 ms
- 4096 bytes: 0.072751 ms
- 1024000 bytes: 1782.343994 ms

AES_GCM_256

- 512 bytes: 0.011098 ms
- 1024 bytes: 0.020489 ms
- 2048 bytes: 0.038626 ms
- 4096 bytes: 0.077457 ms
- 1024000 bytes: 1897.947510 ms

3.4.2. SHA-256 Performance Test with mbed TLS

The tests for creating a SHA-256 hash are performed with the same data sizes as in the preceding tests. The source code of the application is attached in the appendix C. For executing the tests, the line#define SHA_256 must be uncommented.

- 512 bytes: 0.006736 ms
- 1024 bytes: 0.010649 ms
- 2048 bytes: 0.019623 ms
- 1024000 bytes: 912.295715 ms

3.4.3. RSA Performance Test with MatrixSSL

The RSA encryption and decryption is performed with MatrixSSL. The data size is 256 Byte and the algorithms use keys with the length of 2048 and 4096 bit.

RSA with 2048 bit key length

- 11538 microseconds per sig (86 per second)
- 452 microseconds per verify (2212 per second)
- 433 microseconds per encrypt (2309 per second)
- 12402 microseconds per decrypt (80 per second)

RSA with 4096 bit key length

- 86752 microseconds per sig (11 per second)
- 7417 microseconds per verify (134 per second)
- 7471 microseconds per encrypt (133 per second)
- 78971 microseconds per decrypt (12 per second)

The source code of the application is attached in the appendix D. It is an adapted code for the test cases which had to be performed by all group members. The unchanged source code can be downloaded from the MatrixSSL repository [Mate].

3.5. Conclusion

During the work with different libraries it appears that the libraries differ fairly in the way how they are documented and about the scope they cover. The work with the libraries gives a good overview, but to understand how they work much more time would be needed.

The performance tests give an indicator how the length of the key and the amount of the processed date affect the execution time. Altogether longer keys and bigger data sizes lead to an increase in processing time. The user of the algorithms has to take the decision about how much security is needed in relation to the cost of processing time. This decision can be influenced by the hardware platform too, especially when using embedded devices.

4. OpenSSL

"The best that can be expected is that the degree of security be great enough to delay solutions by the enemy for such a length of time that when the solution is finally reached, the information thus obtained has lost all its value." [Sch13]

The following subtask relating to OpenSSL was performed on a 4 GHz Skylake Intel Core i7-6700K processor using the Oracle VM Virtual Box with the Debian 8.7 64-bit system software. Some Linux distributions already come with OpenSSL installed, which seems to be the case for Debian. Here, the OpenSSL version 1.0.1t 3 May 2016, was pre-installed and is used below. The version can be checked with the *openssl version* command.

OpenSSL is an open source cryptography toolkit which supports the Secure Sockets Layer (SSL) as well as the Transport Layer Security (TLS) protocols. It descends from SSLeay and hence remains under a dual license. Both conditions of the OpenSSL and the original SSLeay license apply to the toolkit. It includes two C programming language libraries, libssl and libcrypto, as well as a command line tool, OpenSSL. The libssl API provides implementations for SSL and TLS, while the self-contained libcrypto API provides the general cryptographic routines, which libssl is dependent on. The OpenSSL command line tool consists of different utilities that enable the utilisation of the various cryptography functions of the OpenSSL library directly from the command line. [Opef] [Opeg]

Unfortunately, there is, at the time of this writing, no documentation available for OpenSSL and not many tutorials to be found on the web, either. According to www.openssl.org, however, documentation for developers is currently being written. The documentation section of the official OpenSSL website simply offers frequently asked questions, which might be helpful, and an API reference, which is basically just a listing. Also, a free downloadable OpenSSL cookbook covering the most frequently used features from Ivan Ristić is recommended there. Other places to look are: OpenSSLWiki [Oped] and GitHub [Opeg]. The GitHub project offers some demo programs. However, these are labeled with a note stating that none of them should be expected to work. For more information, the command line tool's manual pages can be referred to. No specific help command exists for OpenSSL, but since it outputs a help text for commands the tool does not recognise, *openssl help* will do just fine. To learn more about a utility, the man pages can be called, as is

customary, for instance, *man ciphers*. [Opeh] [Ris13]

As the plain installation of OpenSSL does not include required files for development, such as the C headers, first of all, the package libssl-dev has to be installed, for example with apt-get libssl-dev. Then the C files need to be compiled and linked with the libraries, for instance, as follows: gcc example.c -o example.o -lcrypto. -lcrypto links the libcrypto API and -lssl the libssl API. [Oped]

4.1. Symmetric Cryptography

Due that the BoringSSL, LibreSSL and OpenSSL libraries all include the high-level cryptographic functions EVP interface, the same source code, which is based on EVP, was used for the OpenSSL symmetric as well as the asymmetric cryptography analysis. Namely, the BoringSSL AES implementation and the LibreSSL RSA implementation respectively.

Some processors offer hardware support for the AES (Advanced Encryption Standard) algorithm, so does the Skylake CPU. Intel's encryption instruction set, designated AES-NI (New Instructions), improves execution of the AES algorithm as well as speeds up data encryption. The EVP interface ensures an implicit use of this cryptographic acceleration, which is not guaranteed for the low-level interfaces. AES-NI also gets passed down to the virtual box. If a given CPU supports AES-NI and if it is enabled can be checked directly, for example, as listed below. [Opef] [Opec] [Opeb] [Opea]

```
# cat /proc/cpuinfo | grep aes | wc -l
8

# grep module /proc/crypto | sort -u
module       : aesni_intel
module       : aes_x86_64
```

The graphs 4.1 and 4.2 list average AES-GCM performance on the Skylake for different amounts of data and three different key lengths. The tables below contain the results of the time measurement. The key lengths are indicated next to the corresponding graphs and are also indicated in the corresponding table heads. The average duration results from 10.000 iterations and ten iterations for the last measurement. To account for fluctuation, ten values are recorded in each case.

AES 128					
byte	512	1024	2048	4096	100 MB
average	3.425100	4.236000	6.017000	9.094200	117380.3
duration	3.460400	4.726600	6.311700	8.898800	118212.7
in	3.377400	4.952800	6.017200	9.240200	115935.5
microsec-	3.553700	4.833600	5.833400	8.756400	116584.9
onds	3.520000	4.589200	5.758100	9.350300	116854.9
	3.553400	6.351200	5.824100	8.840300	117582.3
	3.389900	4.565500	7.126100	8.659700	118288.7
	3.538500	5.065300	6.063600	9.275900	116953.4
	3.458200	4.508200	5.872800	9.081100	117141.5
	3.746200	4.614900	6.226200	8.939300	116684.6

AES 192					
byte	512	1024	2048	4096	100 MB
average	4.880100	5.784000	7.171200	10.316200	123472.8
duration	4.83100	5.936200	7.708400	10.521600	124324.4
in	5.018900	5.847300	7.060500	10.965500	123138.3
microsec-	5.040600	5.771300	7.408000	10.508000	122884.2
onds	5.141100	5.800600	7.633700	10.472200	123327.7
	5.012000	5.638000	7.819800	9.949100	123682.6
	4.996800	5.758300	7.806600	10.649500	123828.3
	5.218600	5.298000	7.506600	16.926300	123433.2
	5.021700	6.114400	7.441800	10.243600	123211.3
	5.055900	5.896900	7.110000	10.125600	123418.3

AES 256					
byte	512	1024	2048	4096	100 MB
average	5.122600	6.855500	9.171000	15.358600	129868.2
duration	4.786900	5.642300	7.810000	11.050700	129892.8
in	5.223700	5.972300	7.780400	14.976900	130118.4
microsec-	5.099500	5.966400	7.361100	11.001200	129597.5
onds	5.002900	5.952200	7.625000	11.088000	129984.6
	5.036200	5.664300	7.497600	11.151600	129984.6
	4.929500	5.992600	7.067800	10.869200	129796.5
	5.044100	6.018900	7.594100	10.577100	129804.2
	5.269000	6.073500	7.257800	11.317300	130080.1
	5.227100	6.312400	7.491200	10.920300	129800.6

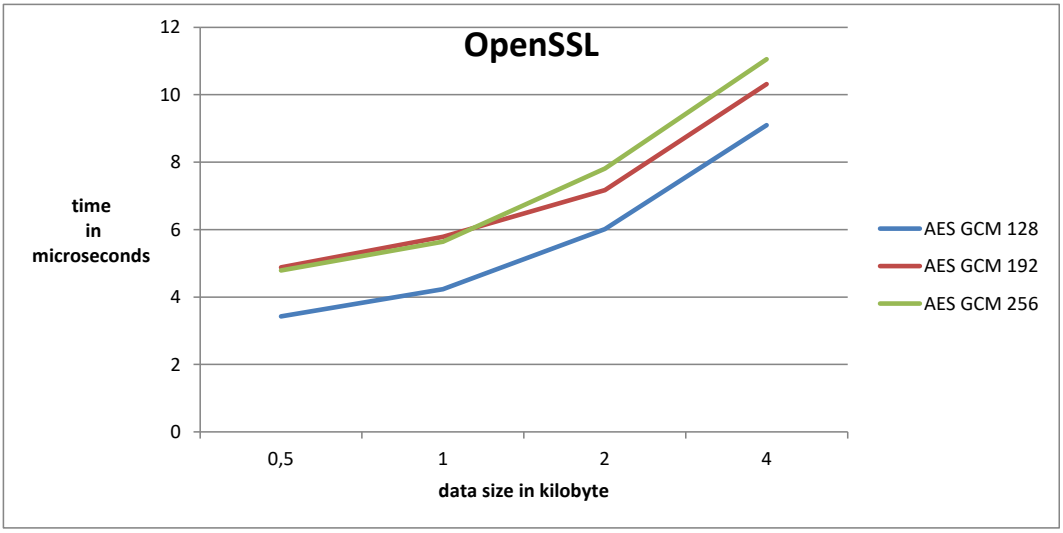

Figure 4.1.: Time measurement of OpenSSL with AES 128gcm, 192gcm and 256gcm

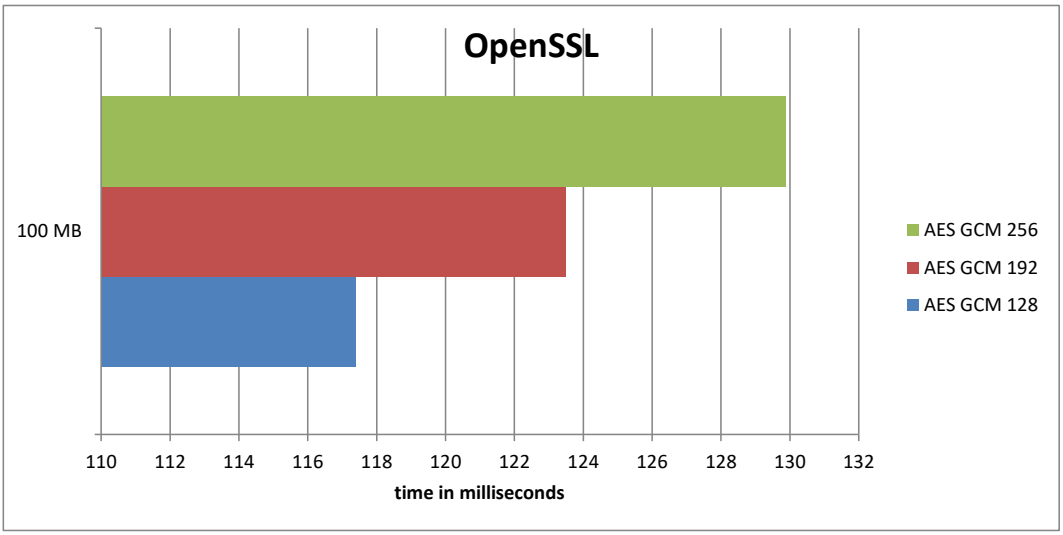

Figure 4.2.: Time measurement of OpenSSL with AES 128gcm, 192gcm and 256gcm and huge data

The results show that AES is fast on high data volume and key lengths. For instance, the average duration for 512 and 1024 bytes is almost constant, while processed data doubled in size. What is more, for 100 megabytes of data different key lengths resulted in similar average durations.

4.2. Asymmetric Cryptography

The graph 4.3 lists average RSA (Rivest Shamir Adleman) performance on the Skylake processor for two different key lengths. The average duration results, at any one time, from 10.000 iterations and to account for fluctuation, ten values are recorded.

RSA		
key length	2048	4096
average duration in microseconds	2075.589000	8809.291000
	2032.222000	8803.788000
	2019.255000	8789.402000
	2079.901000	8860.857000
	2033.272000	8828.783000
	2012.403000	8762.462000
	2104.708000	8810.382000
	2046.829000	8778.340000
	2121.904000	8897.716000
	2047.799000	9067.815000

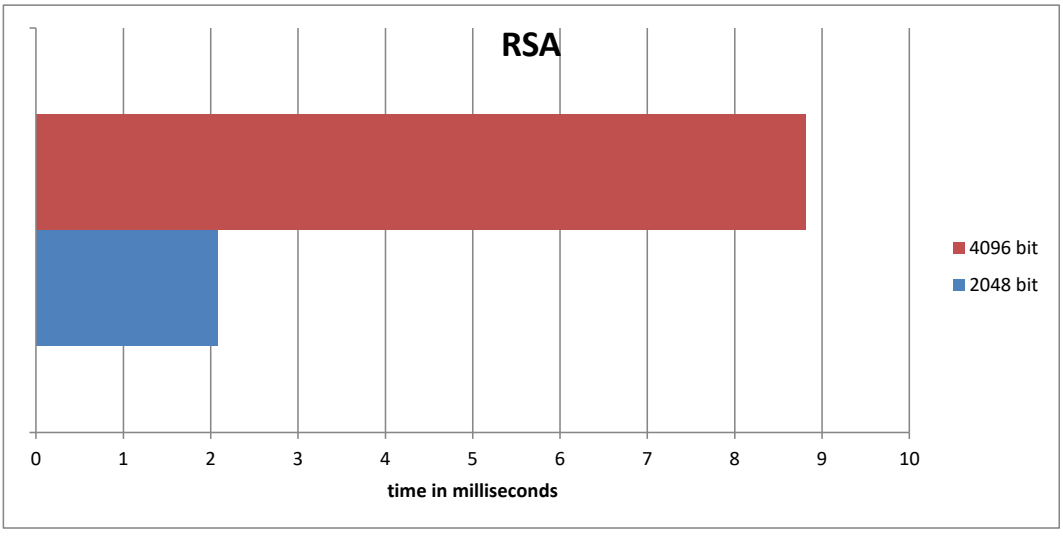

Figure 4.3.: Time measurement of RSA with 2048 bit and 4096 bit key length

For RSA, doubling the key length from 2048 to 4096 bit results in a fourfold average duration here. In contrast to the AES encryption and decryption, the RSA encryption and decryption require more computing time.

4.3. TLS Performance

In order to analyse TLS performance, the openssl command line tool and three of its utilities, req, s_server and s_time, are used here. req is a certificate request and certificate generating utility.s_server is a SSL/TLS server program and s_time a performance timing program. s_server implements a generic SSL/TLS server which listens for connections on a given port using SSL/TLS. Short descriptions of the particular command options are given in the table below. [Opee]

utility	option	description
req	-x509	outputs a self signed certificate
	-newkey	creates a new certificate request and a new private key
	-nodes	private key will not be encrypted
	-days	number of days to certify the certificate for
	-batch	non-interactive mode
s_server	-key	private key to use
	-cert	certificate to use
	-accept	TCP port to listen on for connections
	-www	sends a status message back to the client when it connects
	-quiet	inhibit printing of session and certificate information
s_time	-cipher	allows the cipher list sent by the client to be modified - server determines which cipher suite is used - should take the first supported cipher in the list sent by the client

s_time	-new	performs the timing test using a new session ID for each connection
	-time	Specifies in seconds how long connections should be established
	-reuse	performs the timing test using the same session ID - if neither -new nor -reuse are specified, they are both on by default and executed in sequence

Because the team members worked remotely and three of them used virtual machines, the idea was to establish a connection via the OpenVPN network of the university. This setup turned out to be difficult. Setting up a VPN connection within the Virtual Box and then accessing it via an assigned OpenVPN IP caused problems. Eventually, due to time constraints, the TLS performance test was conducted between only two devices in the same local network as follows:

- Client first, then server:
  4 GHz Skylake Intel Core i7-6700K CPU using the Oracle VM Virtual Box with the Debian 8.7 64-bit
- Server first, then client:
  2.2 GHz Intel Core i5-5200U CPU with the Kali GNU/Linux 2017.1 64-bit

```
// Creating a certificate before starting a test server
# openssl req -x509 -newkey rsa:2048 -sha256 -nodes -days 365 -
    batch -keyout key.pem -out cert.pem

// Starting the server using the previously created certificate
    and private key files. -quiet was // omitted here
# openssl s_server -key key.pem -cert cert.pem -accept 12345 -www

// Starting the client and collecting connection statistics for 20
     seconds
# openssl s_time -connect <IP-address>:12345 -cipher ECDHE-RSA-
    AES256-GCM-SHA384 -new -time 20 (-reuse)
```

```
// Output
# 1823 connections in 2.58s; 706.59 connections/user sec, bytes
    read 0
# 1823 connections in 21 real seconds, 0 bytes read per connection

// Output including -reuse
# 3927 connections in 0.24s; 16362.50 connections/user sec, bytes
    read 0
# 3927 connections in 21 real seconds, 0 bytes read per connection
```

Listing 4.1: Creating a certificate, starting the server and client

This time server and client switched devices.

```
// Output
# 1502 connections in 0.93s; 1615.05 connections/user sec, bytes
    read 0
# 1502 connections in 21 real seconds, 0 bytes read per connection

// Output including -reuse
# 2924 connections in 0.52s; 5623.08 connections/user sec, bytes
    read 0
# 2924 connections in 21 real seconds, 0 bytes read per connection
```

Listing 4.2: Output

Using the same session ID on a follow-up connection via -reuse clearly improves TLS performance. The first client-server setting results in better performance. Now, for the second client server constellation connections with different ciphers were tested, although not all cipher suits work at this point. [Opee]

```
// Output for ECDHE-RSA-AES128-GCM-SHA256
# 1629 connections in 1.04s; 1566.35 connections/user sec, bytes
    read 0
# 1629 connections in 21 real seconds, 0 bytes read per connection

// Output for ECDHE-RSA-AES256-GCM-SHA384
# 1610 connections in 1.04s; 1548.08 connections/user sec, bytes
    read 0
# 1610 connections in 21 real seconds, 0 bytes read per connection
```

Listing 4.3: Output for ECDHE

Choosing these two cipher suits leads to a slight performance increase in comparison to ECDHE-RSA-AES256-GCM-SHA384.

Appendices

A. AES Implementation with BoringSSL

```
#include <openssl/evp.h>
#include <openssl/aes.h>
#include <openssl/err.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <errno.h>
#include <time.h>

#define LENGTH 1024 * 32
#define SIZE (LENGTH * sizeof(char))
#define DEBUG 0

struct timespec start = { 0, 0 };
struct timespec end = { 0, 0 };

void readRandomData(unsigned char* buffer, size_t size, int length
    ) {
    int fileDescriptor = open("/dev/urandom", O_RDONLY);
    read(fileDescriptor, buffer, size);
    close(fileDescriptor);
    buffer[length - 1] = '\0';
}

void init(unsigned char* keyData, int keyDataLength, unsigned char
    * salt, EVP_CIPHER_CTX* encryptContext, EVP_CIPHER_CTX*
    decryptContext) {
    unsigned char key[32], iv[32];
    EVP_BytesToKey(EVP_aes_192_gcm(), EVP_sha256(), salt, keyData,
        keyDataLength, 5, key, iv);
    EVP_CIPHER_CTX_init(encryptContext);
    EVP_EncryptInit_ex(encryptContext, EVP_aes_192_gcm(), NULL,
        key, iv);
    EVP_CIPHER_CTX_init(decryptContext);
    EVP_DecryptInit_ex(decryptContext, EVP_aes_192_gcm(), NULL,
        key, iv);
}
```

```
unsigned char* encrypt(EVP_CIPHER_CTX* context, unsigned char*
    plaintext, int* length) {
    int ciphertextLength = *length + AES_BLOCK_SIZE;
    int finalLength = 0;
    unsigned char* ciphertext = malloc(ciphertextLength);

    EVP_EncryptInit_ex(context, NULL, NULL, NULL, NULL);
    EVP_EncryptUpdate(context, ciphertext, &ciphertextLength,
        plaintext, *length);
    EVP_EncryptFinal_ex(context, ciphertext + ciphertextLength, &
        finalLength);
    *length = ciphertextLength + finalLength;

    return ciphertext;
}

unsigned char* decrypt(EVP_CIPHER_CTX* context, unsigned char*
    ciphertext, int* length) {
    int decryptedtextLength = *length;
    int finalLength = 0;
    unsigned char* decryptedtext = malloc(decryptedtextLength);

    EVP_DecryptInit_ex(context, NULL, NULL, NULL, NULL);
    EVP_DecryptUpdate(context, decryptedtext, &decryptedtextLength
        , ciphertext, *length);
    EVP_DecryptFinal_ex(context, decryptedtext +
        decryptedtextLength, &finalLength);
    *length = decryptedtextLength + finalLength;

    return decryptedtext;
}

int main() {
    EVP_CIPHER_CTX encryptContext, decryptContext;
    unsigned int salt[] = {12345, 54321};
    int length = LENGTH;
    double duration = 0;
    unsigned char* keyData = "Random text!";
    unsigned char* plaintext = malloc(SIZE);
    unsigned char* ciphertext;
    unsigned char* decryptedtext;

    int i;
    for (i = 0; i < 10000; i++) {
        readRandomData(plaintext, SIZE, LENGTH);
        clock_gettime(CLOCK_MONOTONIC, &start);
        init(keyData, strlen(keyData), (unsigned char *)&salt, &
            encryptContext, &decryptContext);
```

```
        ciphertext = encrypt(&encryptContext, plaintext, &length);
        decryptedtext = decrypt(&decryptContext, ciphertext, &
            length);
        clock_gettime(CLOCK_MONOTONIC, &end);

        duration += (double)((end.tv_sec - start.tv_sec) * 1000000
            + (end.tv_nsec - start.tv_nsec) / 1000);
    }


    printf("Average duration: %f us\n", duration / 10000);

    if (DEBUG) {
        printf("Plaintext:\n");
        BIO_dump_fp (stdout, plaintext, LENGTH);
    }

    if (DEBUG) {
        printf("Ciphertext:\n");
        BIO_dump_fp (stdout, ciphertext, LENGTH);
    }

    if (DEBUG) {
        printf("Decrypted text:\n");
        BIO_dump_fp (stdout, decryptedtext, LENGTH);
    }

}
```

Listing A.1: AES implementation with BoringSSL

B. RSA Implementation with BoringSSL

Listing B.1: RSA implementation with BoringSSL

```
#include <openssl/bio.h>
#include <openssl/err.h>
#include <openssl/evp.h>
#include <openssl/pem.h>
#include <openssl/ssl.h>

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <errno.h>
#include <time.h>

#define RSA_SIZE 4096
#define BYTES (2048 / 8 - 11)
#define LENGTH ((BYTES >= 4096) ? BYTES : 4096)
#define SIZE (BYTES * sizeof(char))
#define DEBUG 1

int padding = RSA_PKCS1_PADDING;
struct timespec start = { 0, 0 };
struct timespec end = { 0, 0 };

void readRandomData(unsigned char* buffer, size_t size, int bytes)
    {
    int fileDescriptor = open("/dev/urandom", O_RDONLY);
    read(fileDescriptor, buffer, size);
    close(fileDescriptor);
    int i;
    for (i = 0; i < bytes; i++) {
        buffer[i] = (buffer[i] % 93) + 33;
    }
    buffer[bytes - 1] = '\0';
}

RSA* createRSA(unsigned char* key, int isPublicKey) {
```

```
    RSA* rsa = NULL;
    BIO* keyBIO;
    keyBIO = BIO_new_mem_buf(key, -1);

    if (isPublicKey) {
        rsa = PEM_read_bio_RSA_PUBKEY(keyBIO, &rsa, NULL, NULL);
    } else {
        rsa = PEM_read_bio_RSAPrivateKey(keyBIO, &rsa, NULL, NULL)
            ;
    }

    return rsa;
}

int encryptRSA(unsigned char* plaintext, int plaintextLength,
    unsigned char* key, unsigned char* ciphertext) {
    RSA* rsa = createRSA(key, 1);
    int ciphertextLength = RSA_public_encrypt(plaintextLength,
        plaintext, ciphertext, rsa, padding);
    return ciphertextLength;
}

int decryptRSA(unsigned char* ciphertext, int ciphertextLength,
    unsigned char* key, unsigned char* decryptedText) {
    RSA* rsa = createRSA(key, 0);
    int decryptedTextLength = RSA_private_decrypt(ciphertextLength
        , ciphertext, decryptedText, rsa, padding);
    return decryptedTextLength;
}

int main() {
    char publicKey[] =

    "-----BEGIN PUBLIC KEY-----\n"\
    "MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAy8Dbv8prpJ/0
        kKhlGeJY\n"\
    "ozo2t60EG8L0561g13R29LvMR5hyvGZlGJpmn65+
        A4xHXInJYiPuKzrKUnApeLZ+\n"\
    "vw1HocOAZtWK0z3r26uA8kQYOKX9Qt/
        DbCdvsF9wF8gRK0ptx9M6R13NvBxvVQAp\n"\
    "fc9jB9nTzphOgM4JiEYvlV8FLhg9yZovMYd6Wwf3aoXK891VQxTr/kQYoq1Yp
        +68\n"\
    "i6T4nNq7NWC+UNVjQHxNQMQMzU6lWCX8zyg3yH88OAQkUXIXKfQ+
        NkvYQ1cxaMoV\n"\
    "PpY72+eVthKzpMeyHkBn7ciumk5qgLTEJAfWZpe4f4eFZj/
        Rc8Y8Jj2IS5kVPjUy\n"\
    "wQIDAQAB\n"\
    "-----END PUBLIC KEY-----\n";
```

```
char privateKey[] =

"-----BEGIN RSA PRIVATE KEY-----\n"\
"MIIEowIBAAKCAQEAy8Dbv8prpJ/0
    kKhlGeJYozo2t60EG8L0561g13R29LvMR5hy\n"\
"vGZlGJpmn65+A4xHXInJYiPuKzrKUnApeLZ+
    vw1HocOAZtWK0z3r26uA8kQYOKX9\n"\
"Qt/
    DbCdvsF9wF8gRK0ptx9M6R13NvBxvVQApfc9jB9nTzphOgM4JiEYvlV8FLhg9
    \n"\
"yZovMYd6Wwf3aoXK891VQxTr/kQYoq1Yp+68i6T4nNq7NWC+
    UNVjQHxNQMQMzU6l\n"\
"WCX8zyg3yH88OAQkUXIXKfQ+NkvYQ1cxaMoVPpY72+
    eVthKzpMeyHkBn7ciumk5q\n"\
"gLTEJAfWZpe4f4eFZj/
    Rc8Y8Jj2IS5kVPjUywQIDAQABAoIBADhg1u1Mv1hAAlX8\n"\
"omz1Gn2f4AAW2aos2cM5UDCNw1SYmj+9SRIkaxjRsE/C4o9sw1oxrg1/
    z6kajV0e\n"\
"N/t008FdlVKHXAIYWF93JMoVvIpMmT8jft6AN/
    y3NMpivgt2inmmEJZYNioFJKZG\n"\
"X+/vKYvsVISZm2fw8NfnKvAQK55yu+GRWBZGOeS9K+
    LbYvOwcrjKhHz66m4bedKd\n"\
"gVAix6NE5iwmjNXktSQlJMCjbtdNXg/xo1/G4kG2p/
    MO1HLcKfe1N5FgBiXj3Qjl\n"\
"vgvjJZkh1as2KTgaPOBqZaP03738VnYg23ISyvfT/
    teArVGtxrmFP7939EvJFKpF\n"\
"1wTxuDkCgYEA7t0DR37zt+dEJy+5
    vm7zSmN97VenwQJFWMiulkHGa0yU3lLasxxu\n"\
"m0oUtndIjenIvSx6t3Y+
    agK2F3EPbb0AZ5wZ1p1IXs4vktgeQwSSBdqcM8LZFDvZ\n"\
"
    uPboQnJoRdIkd62XnP5ekIEIBAfOp8v2wFpSfE7nNH2u4CpAXNSF9HsCgYEA2l8D
    \n"\
"JrDE5m9Kkn+J4l+
    AdGfeBL1igPF3DnuPoV67BpgiaAgI4h25UJzXiDKKoa706S0D\n"\
"4XB74zOLX11MaGPMIdhlG+
    SgeQfNoC5lE4ZWXNyESJH1SVgRGT9nBC2vtL6bxCVV\n"\
"WBkTeC5D6c/
    QXcai6yw6OYyNNdp0uznKURe1xvMCgYBVYYcEjWqMuAvyferFGV+5\n"\
"nWqr5gM+yJMFM2bEqupD/
    HHSLoeiMm2O8KIKvwSeRYzNohKTdZ7FwgZYxr8fGMoG\n"\
"PxQ1VK9DxCvZL4tRpVaU5Rmknud9hg9DQG6xIbgIDR+
    f79sb8QjYWmcFGc1SyWOA\n"\
"
    SkjlykZ2yt4xnqi3BfiD9QKBgGqLgRYXmXp1QoVIBRaWUi55nzHg1XbkWZqPXvz1
    \n"\
"I3uMLv1jLjJlHk3euKqTPmC05HoApKwSHeA0/
    gOBmg404xyAYJTDcCidTg6hlF96\n"\
"
```

```
        ZBja3xApZuxqM62F6dV4FQqzFX0WWhWp5n301N33r0qR6FumMKJzmVJ1TA8tmzEF
        \n"\
    "yINRAoGBAJqioYs8rK6eXzA8ywYLjqTLu/yQSLBn/4ta36K8DyCoLNlNxSuox
        +A5\n"\
    "w6z2vEfRVQDq4Hm4vBzjdi3QfYLNkTiTqLcvgWZ+eX44ogXtdTDO7c+
        GeMKWz4XX\n"\
    "uJSUVL5+CVjKLjZEJ6Qc2WZLl94xSwL71E41H4YciVnSCQxVc4Jw\n"\
    "-----END RSA PRIVATE KEY-----\n";

    char publicKey4096[] =

        "-----BEGIN PUBLIC KEY-----\n"\
        [..]
        "-----END PUBLIC KEY-----\n";

    char privateKey4096[] =

        "-----BEGIN RSA PRIVATE KEY-----\n"\
        [..]
        "-----END RSA PRIVATE KEY-----\n";

    unsigned char plaintext[BYTES] = {};
    unsigned char ciphertext[RSA_SIZE * 2] = {};
    unsigned char decryptedText[RSA_SIZE * 2] = {};
    double duration = 0;

    int i;
    for (i = 0; i < 1000; i++) {
        readRandomData(&plaintext, SIZE, BYTES);
        clock_gettime(CLOCK_MONOTONIC, &start);
        int ciphertextLength = encryptRSA(plaintext, strlen(
            plaintext), publicKey4096, ciphertext);
        int decryptedTextLength = decryptRSA(ciphertext,
            ciphertextLength, privateKey4096, decryptedText);
        clock_gettime(CLOCK_MONOTONIC, &end);
        duration += (double)((end.tv_sec - start.tv_sec) * 1000000
            + (end.tv_nsec - start.tv_nsec) / 1000);
    }

    printf("Average duration: %f us\n", duration / 1000);

    return 0;
}
```

C. AES Implementation with mbed TLS

```
/*
------------------------------------------------
Program to create random data in memory or in file
and measure encryption duration
------------------------------------------------
Download mbedtls:   https://tls.mbed.org/download/start/mbedtls
   -2.4.2-gpl.tgz
Compile :       gcc mbedtls_aufgabe1.c -o mbedtls_aufgabe1.o -
   lmbedtls -lmbedcrypto -lmbedx509
Run:            ./mbedtls_aufgabe1.o
View output file:   ls -lh
*/

#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <errno.h>
#include <stdint.h>
#include <time.h>
#include <string.h>
#include <locale.h>

//Mbed TLS crypto Libs
#include "mbedtls/aes.h"
#include "mbedtls/gcm.h"
#include "mbedtls/sha256.h"


#define MEGABYTE (1024* 1024 * sizeof(uint8_t))
#define KILOBYTE (1024 * sizeof(uint8_t))
/*_____________configuration_____________________________________*/
#define BUFFERING_TIMEOUT_MILISEC 15000.f
#define BUF_SIZE (4096 * sizeof(uint8_t))
//#define SAVE_RANDOM_FILE
//#define SHA_256
#define AES_GCM
/* lenght of AES key is e.g. 16*8 = 128 bit */
#define AES_KEY_LEN_BYTES 32
```

```
#define AES_IV_LEN_BYTES (AES_KEY_LEN_BYTES/2)
#define NUM_ITERATIONS 10000
/*_____________________________________________________________*/

//variables and functions for key generation
#include "mbedtls/entropy.h"
#include "mbedtls/ctr_drbg.h"
//source: https://tls.mbed.org/kb/how-to/generate-an-aes-key

unsigned char key[AES_KEY_LEN_BYTES];
unsigned char iv[AES_IV_LEN_BYTES];
unsigned char hash[32];
mbedtls_gcm_context gcm;
mbedtls_sha256_context sha;
uint8_t rand_buffer [BUF_SIZE];

void generate_key(char * buf, int len){
    //printf("\ngenerate_key called with len: %i", len);
    char *pers = "aes generate key";
    int ret;

    mbedtls_ctr_drbg_context ctr_drbg;
    mbedtls_entropy_context entropy;
    mbedtls_entropy_init( &entropy );
    mbedtls_ctr_drbg_init( &ctr_drbg );
    if( ( ret = mbedtls_ctr_drbg_seed( &ctr_drbg,
       mbedtls_entropy_func, &entropy,
        (unsigned char *) pers, strlen( pers ) ) ) != 0 )
    {
        printf( " failed\n ! mbedtls_ctr_drbg_init returned -0x%04
           x\n", -ret );
        exit (1);
    }
    if( ( ret = mbedtls_ctr_drbg_random( &ctr_drbg, buf, len ) )
       != 0 )
    {
        printf( " failed\n ! mbedtls_ctr_drbg_random returned -0x
           %04x\n", -ret );
        exit(1);
    }
}


int main() {
    //open random device
    int fd_random = NULL;
    fd_random = open("/dev/urandom", O_RDONLY);
    if (!fd_random){
        printf("\nerror openening /dev/urandom");
```

```
        exit(1);
    }
    //read random data into buffer
    int n;
    float diff;
    clock_t t1, t2;
    t1 = clock();
    do{
        n = n + read(fd_random, &rand_buffer[n], BUF_SIZE-n);
        t2 = clock();
        diff = ((float) (t2-t1) / CLOCKS_PER_SEC) * 1000;
    }while (n != BUF_SIZE && diff < BUFFERING_TIMEOUT_MILISEC);
    if (diff >= BUFFERING_TIMEOUT_MILISEC){
        printf("\ntimeout reached, read into rand_buffer aborted\n
            ");
        exit (1);
    }
    if (n != BUF_SIZE){
        printf("\nerror read into rand_buffer. error:%i\n", errno)
            ;
        exit (1);
    }else{
        printf("\nReading random data into buffer terminated after
            %f miliseconds", diff);
    }
    close(fd_random);
    printf("\nUsing buffer with %'lu bytes", BUF_SIZE);

#ifdef SAVE_RANDOM_FILE
    //write buffer content into output file
    int fd_out = NULL;
    fd_out = open("./random_data", O_CREAT | O_TRUNC | O_RDWR,
        S_IRUSR | S_IWUSR);
    if (!fd_out){
        printf("\nerror creating output file, error:%i\n", errno);
        exit(1);
    }
    t1 = clock();
    n = write (fd_out, rand_buffer, BUF_SIZE);
    t2 = clock();
    if (n != BUF_SIZE){
        printf("\nerror writing output file, error:%i\n", errno);
        exit(1);
    }else{
        diff = ((float) (t2-t1) / CLOCKS_PER_SEC) * 1000;
        printf("\nWriting into random_file terminated after %f
            miliseconds. size: %lu bytes, name: random_data" \
            , diff, BUF_SIZE);
    }
```

```
    close(fd_out);
#endif

// Data Encryption
    int result = 0;
    int iter;
    float avg_time = 0;
#ifdef AES_GCM
    printf("\nUsing AES-GCM with %i bit lenght", AES_KEY_LEN_BYTES
        *8);
    //printf("\nBuffer content before encryption: %s\n",
        rand_buffer);
    generate_key(key, AES_KEY_LEN_BYTES);
    //printf("\nkey: %s", key);
    generate_key(iv, AES_IV_LEN_BYTES);
    //printf("\niv: %s", iv);
    mbedtls_gcm_init(&gcm);

    result = mbedtls_gcm_setkey(&gcm, MBEDTLS_CIPHER_ID_AES, key,
       AES_KEY_LEN_BYTES*8);
    if (result != 0){
        printf("\nerror in mbedtls_gcm_setkey, return value:%i\n",
            result);
        exit(1);
    }
    for (iter = 1; iter <= NUM_ITERATIONS; iter++){
        t1 = clock();
        result = mbedtls_gcm_crypt_and_tag( &gcm,
           MBEDTLS_GCM_ENCRYPT , \
                BUF_SIZE, iv, AES_IV_LEN_BYTES, NULL, 0,
                    rand_buffer,                \
                rand_buffer, AES_IV_LEN_BYTES, iv);
        t2 = clock();
        if (result != 0){
            printf("\nerror in mbedtls_gcm_crypt_and_tag, return
               value:%i\n", result);
            exit(1);
        }
        diff = ((float) (t2-t1) / CLOCKS_PER_SEC) * 1000;
        avg_time = avg_time + diff;
    }
    avg_time = avg_time / NUM_ITERATIONS;
    printf("\nLast encryption iteration terminated after %f
        miliseconds." , diff);
    printf("\nAverage time in %i iterations: %f miliseconds",
       NUM_ITERATIONS, avg_time);
    //printf("\nBuffer content after encryption: %s\n",
        rand_buffer);
#endif
```

```
#ifdef SHA_256
    printf("\nUsing SHA-256");
    //Initialize SHA-256 context
    mbedtls_sha256_init(&sha);
    //SHA-256 context setup, 0 = use SHA256, 1 = use SHA224
    mbedtls_sha256_starts(&sha, 0 );
    for (iter = 1; iter <= NUM_ITERATIONS; iter++){
        //SHA-256 process buffer
        t1 = clock();
        mbedtls_sha256_update(&sha, rand_buffer, BUF_SIZE);
        //SHA-256 final digest
        mbedtls_sha256_finish(&sha, hash);
        t2 = clock();
        diff = ((float) (t2-t1) / CLOCKS_PER_SEC) * 1000;
        avg_time = avg_time + diff;
    }
    avg_time = avg_time / NUM_ITERATIONS;
    printf("\nLast hash: \" %s \", generated in %f miliseconds.",
       hash, diff);
    printf("\nAverage time in %i iterations: %f miliseconds",
       NUM_ITERATIONS, avg_time);
#endif
    printf("\nProgram terminated\n\n");
    return 0;
}
```

Listing C.1: AES implementation with mbed TLS

D. RSA Implementation with MatrixSSL

```
#include <unistd.h> /* sleep */
#include "crypto/cryptoImpl.h"

#if defined(USE_RSA) && defined(USE_PRIVATE_KEY_PARSING)

typedef void pkaCmdInfo_t;

/* OPERATIONS TO TEST */
# define SIGN_OP     /* Private encrypt operations */
# define VERIFY_OP   /* Public decrypt operations */

# define ENCRYPT_OP /* Public encrypt operations */
# define DECRYPT_OP /* Private decrypt operations */

/* KEY SIZES TO TEST */
#   define DO_2048
#   define DO_4096

#   define ITER_2048    1000
#   define ITER_4096    1000
/* Another pass with different keys */
/* #define INCLUDE_SECOND_SET */

# define PS_OH (sizeof(psPool_t))

/*
    Tuned to smallest K for each key size and optimization setting
        .
    Private key operations take about 1/2 the RAM when optimized
        for size
 */
# ifdef PS_PUBKEY_OPTIMIZE_FOR_SMALLER_RAM
#   define POOL_SIGN_512        (1 * 1024) + PS_OH
#   define POOL_VERIFY_512      (2 * 1024) + PS_OH
#   define POOL_ENCRYPT_512     (2 * 1024) + PS_OH
#   define POOL_DECRYPT_512     (1 * 1024) + PS_OH
#   define POOL_MISC_512        (3 * 1024) + PS_OH
```

```
#  define POOL_SIGN_1024        (2 * 1024) + PS_OH
#  define POOL_VERIFY_1024      (3 * 1024) + PS_OH
#  define POOL_ENCRYPT_1024     (3 * 1024) + PS_OH
#  define POOL_DECRYPT_1024     (2 * 1024) + PS_OH
#  define POOL_MISC_1024        (4 * 1024) + PS_OH

#  define POOL_SIGN_2048        (3 * 1024) + PS_OH
#  define POOL_VERIFY_2048      (5 * 1024) + PS_OH
#  define POOL_ENCRYPT_2048     (5 * 1024) + PS_OH
#  define POOL_DECRYPT_2048     (3 * 1024) + PS_OH
#  define POOL_MISC_2048        (7 * 1024) + PS_OH

#  define POOL_SIGN_4096        (6 * 1024) + PS_OH
#  define POOL_VERIFY_4096      (8 * 1024) + PS_OH
#  define POOL_ENCRYPT_4096     (8 * 1024) + PS_OH
#  define POOL_DECRYPT_4096     (6 * 1024) + PS_OH
#  define POOL_MISC_4096        (12 * 1024) + PS_OH

# else /* PS_PUBKEY_OPTIMIZE_FOR_FASTER_SPEED */

#  define POOL_SIGN_512         (3 * 1024) + PS_OH
#  define POOL_VERIFY_512       (2 * 1024) + PS_OH
#  define POOL_ENCRYPT_512      (2 * 1024) + PS_OH
#  define POOL_DECRYPT_512      (3 * 1024) + PS_OH
#  define POOL_MISC_512         (4 * 1024) + PS_OH

#  define POOL_SIGN_1024        (5 * 1024) + PS_OH
#  define POOL_VERIFY_1024      (3 * 1024) + PS_OH
#  define POOL_ENCRYPT_1024     (3 * 1024) + PS_OH
#  define POOL_DECRYPT_1024     (5 * 1024) + PS_OH
#  define POOL_MISC_1024        (6 * 1024) + PS_OH

#  define POOL_SIGN_2048        (7 * 1024) + PS_OH
#  define POOL_VERIFY_2048      (5 * 1024) + PS_OH
#  define POOL_ENCRYPT_2048     (5 * 1024) + PS_OH
#  define POOL_DECRYPT_2048     (7 * 1024) + PS_OH
#  define POOL_MISC_2048        (9 * 1024) + PS_OH

#  define POOL_SIGN_4096        (13 * 1024) + PS_OH
#  define POOL_VERIFY_4096      ( 8 * 1024) + PS_OH
#  define POOL_ENCRYPT_4096     ( 8 * 1024) + PS_OH
#  define POOL_DECRYPT_4096     (13 * 1024) + PS_OH
#  define POOL_MISC_4096        (16 * 1024) + PS_OH
# endif

# ifdef DO_512
#  include "rsa3e512.h"
#  include "rsa17e512.h"
#  include "rsa257e512.h"
```

```
#  include "rsa65537e512.h"
# endif /* DO_512 */

# ifdef DO_1024
#  include "rsa3e1024.h"
#  include "rsa17e1024.h"
#  include "rsa257e1024.h"
#  include "rsa65537e1024.h"
# endif /* DO_1024 */

# ifdef DO_2048
#  include "rsa3e2048.h"
#  include "rsa17e2048.h"
#  include "rsa257e2048.h"
#  include "rsa65537e2048.h"
# endif /* DO_2048 */

# ifdef DO_4096
#  include "rsa3e4096.h"
#  include "rsa17e4096.h"
#  include "rsa257e4096.h"
#  include "rsa65537e4096.h"
# endif /* DO_4096 */

# ifdef INCLUDE_SECOND_SET
#  ifdef DO_512
#   include "rsa3e512_1.h"
#   include "rsa17e512_1.h"
#   include "rsa257e512_1.h"
#   include "rsa65537e512_1.h"
#  endif /* DO_512 */

#  ifdef DO_1024
#   include "rsa3e1024_1.h"
#   include "rsa17e1024_1.h"
#   include "rsa257e1024_1.h"
#   include "rsa65537e1024_1.h"
#  endif /* DO_1024 */

#  ifdef DO_2048
#   include "rsa3e2048_1.h"
#   include "rsa17e2048_1.h"
#   include "rsa257e2048_1.h"
#   include "rsa65537e2048_1.h"
#  endif /* DO_2048 */

#  ifdef DO_4096
#   include "rsa3e4096_1.h"
#   include "rsa17e4096_1.h"
```

```
#   include "rsa257e4096_1.h"
#   include "rsa65537e4096_1.h"
#  endif /* DO_4096 */
# endif  /* INCLUDE_SECOND_SET */

typedef struct
{
    char *name;
    const unsigned char *key;
    uint32 len;
    int32 iter;
    int32 poolSign;
    int32 poolVerify;
    int32 poolEncrypt;
    int32 poolDecrypt;
    int32 poolMisc;
} keyList_t;

# ifdef USE_HIGHRES_TIME
#  define psDiffMsecs(A, B, C) (unsigned long long) psDiffUsecs(A,
    B)
#  define TIME_UNITS "    %lld Microsekunden"
#  define PER_SEC(A) ((A) ? (1000000 / (A)) : 0)
# else
#  define TIME_UNITS "    %d msecs"
#  define PER_SEC(A) ((A) ? (1000 / (A)) : 0)
# endif

/*
    Add an iteration count so we don't have to run the large keys
       so many times
 */
static keyList_t keys[] = {
# ifdef DO_512
    { "rsa3e512",         rsa3e512,          sizeof(rsa3e512),
              ITER_512,           POOL_SIGN_512,
      POOL_VERIFY_512, POOL_ENCRYPT_512, POOL_DECRYPT_512,
         POOL_MISC_512 },
    { "rsa17e512",        rsa17e512,         sizeof(rsa17e512),
              ITER_512,           POOL_SIGN_512,
      POOL_VERIFY_512, POOL_ENCRYPT_512, POOL_DECRYPT_512,
         POOL_MISC_512 },
    { "rsa257e512",       rsa257e512,        sizeof(rsa257e512),
              ITER_512,           POOL_SIGN_512,
      POOL_VERIFY_512, POOL_ENCRYPT_512, POOL_DECRYPT_512,
         POOL_MISC_512 },
    { "rsa65537e512",     rsa65537e512,      sizeof(rsa65537e512),
              ITER_512,
      POOL_SIGN_512, POOL_VERIFY_512, POOL_ENCRYPT_512,
```

```
        POOL_DECRYPT_512,
      POOL_MISC_512 },
# endif
# ifdef DO_1024
    { "rsa3e1024",        rsa3e1024,          sizeof(rsa3e1024),
              ITER_1024,          POOL_SIGN_1024,
      POOL_VERIFY_1024, POOL_ENCRYPT_1024, POOL_DECRYPT_1024,
      POOL_MISC_1024 },
    { "rsa17e1024",       rsa17e1024,         sizeof(rsa17e1024),
              ITER_1024,          POOL_SIGN_1024,
      POOL_VERIFY_1024, POOL_ENCRYPT_1024, POOL_DECRYPT_1024,
      POOL_MISC_1024 },
    { "rsa257e1024",      rsa257e1024,        sizeof(rsa257e1024),
              ITER_1024,          POOL_SIGN_1024,
      POOL_VERIFY_1024, POOL_ENCRYPT_1024, POOL_DECRYPT_1024,
      POOL_MISC_1024 },
    { "rsa65537e1024",    rsa65537e1024,      sizeof(rsa65537e1024),
             ITER_1024,
      POOL_SIGN_1024, POOL_VERIFY_1024, POOL_ENCRYPT_1024,
        POOL_DECRYPT_1024,
      POOL_MISC_1024 },
# endif
# ifdef DO_2048

    { "rsa257e2048",      rsa257e2048,        sizeof(rsa257e2048),
              ITER_2048,          POOL_SIGN_2048,
      POOL_VERIFY_2048, POOL_ENCRYPT_2048, POOL_DECRYPT_2048,
      POOL_MISC_2048 },

# endif
# ifdef DO_4096

    { "rsa257e4096",      rsa257e4096,        sizeof(rsa257e4096),
              ITER_4096,          POOL_SIGN_4096,
      POOL_VERIFY_4096, POOL_ENCRYPT_4096, POOL_DECRYPT_4096,
      POOL_MISC_4096 },

# endif
# ifdef INCLUDE_SECOND_SET
#  ifdef DO_512
    { "rsa3e512_1",       rsa3e5121,          sizeof(rsa3e5121),
              ITER_512,           POOL_SIGN_512,
      POOL_VERIFY_512, POOL_ENCRYPT_512, POOL_DECRYPT_512,
        POOL_MISC_512 },
    { "rsa17e512_1",      rsa17e5121,         sizeof(rsa17e5121),
              ITER_512,           POOL_SIGN_512,
      POOL_VERIFY_512, POOL_ENCRYPT_512, POOL_DECRYPT_512,
        POOL_MISC_512 },
    { "rsa257e512_1",     rsa257e5121,        sizeof(rsa257e5121),
```

```
            ITER_512,           POOL_SIGN_512,
      POOL_VERIFY_512, POOL_ENCRYPT_512, POOL_DECRYPT_512,
          POOL_MISC_512 },
    { "rsa65537e512_1",  rsa65537e5121,      sizeof(rsa65537e5121),
            ITER_512,
      POOL_SIGN_512, POOL_VERIFY_512, POOL_ENCRYPT_512,
          POOL_DECRYPT_512,
      POOL_MISC_512 },
#   endif
#   ifdef DO_1024
    { "rsa3e1024_1",     rsa3e10241,         sizeof(rsa3e10241),
             ITER_1024,          POOL_SIGN_1024,
      POOL_VERIFY_1024, POOL_ENCRYPT_1024, POOL_DECRYPT_1024,
      POOL_MISC_1024 },
    { "rsa17e1024_1",    rsa17e10241,        sizeof(rsa17e10241),
            ITER_1024,
      POOL_SIGN_1024, POOL_VERIFY_1024, POOL_ENCRYPT_1024,
          POOL_DECRYPT_1024,
      POOL_MISC_1024 },
    { "rsa257e1024_1",   rsa257e10241,       sizeof(rsa257e10241),
            ITER_1024,
      POOL_SIGN_1024, POOL_VERIFY_1024, POOL_ENCRYPT_1024,
          POOL_DECRYPT_1024,
      POOL_MISC_1024 },
    { "rsa65537e1024_1", rsa65537e10241,     sizeof(rsa65537e10241)
        ,     ITER_1024,
      POOL_SIGN_1024, POOL_VERIFY_1024, POOL_ENCRYPT_1024,
          POOL_DECRYPT_1024,
      POOL_MISC_1024 },
#   endif
#   ifdef DO_2048
    { "rsa3e2048_1",     rsa3e20481,         sizeof(rsa3e20481),
             ITER_2048,          POOL_SIGN_2048,
      POOL_VERIFY_2048, POOL_ENCRYPT_2048, POOL_DECRYPT_2048,
      POOL_MISC_2048 },
    { "rsa17e2048_1",    rsa17e20481,        sizeof(rsa17e20481),
            ITER_2048,
      POOL_SIGN_2048, POOL_VERIFY_2048, POOL_ENCRYPT_2048,
          POOL_DECRYPT_2048,
      POOL_MISC_2048 },
    { "rsa257e2048_1",   rsa257e20481,       sizeof(rsa257e20481),
            ITER_2048,
      POOL_SIGN_2048, POOL_VERIFY_2048, POOL_ENCRYPT_2048,
          POOL_DECRYPT_2048,
      POOL_MISC_2048 },
    { "rsa65537e2048_1", rsa65537e20481,     sizeof(rsa65537e20481)
        ,     ITER_2048,
      POOL_SIGN_2048, POOL_VERIFY_2048, POOL_ENCRYPT_2048,
          POOL_DECRYPT_2048,
```

```
      POOL_MISC_2048 },
#  endif
#  ifdef DO_4096
    { "rsa3e4096_1",      rsa3e40961,         sizeof(rsa3e40961),
              ITER_4096,          POOL_SIGN_4096,
      POOL_VERIFY_4096, POOL_ENCRYPT_4096, POOL_DECRYPT_4096,
      POOL_MISC_4096 },
    { "rsa17e4096_1",     rsa17e40961,        sizeof(rsa17e40961),
             ITER_4096,
      POOL_SIGN_4096, POOL_VERIFY_4096, POOL_ENCRYPT_4096,
          POOL_DECRYPT_4096,
      POOL_MISC_4096 },
    { "rsa257e4096_1",    rsa257e40961,       sizeof(rsa257e40961),
            ITER_4096,
      POOL_SIGN_4096, POOL_VERIFY_4096, POOL_ENCRYPT_4096,
          POOL_DECRYPT_4096,
      POOL_MISC_4096 },
    { "rsa65537e4096_1", rsa65537e40961,     sizeof(rsa65537e40961)
        ,     ITER_4096,
      POOL_SIGN_4096, POOL_VERIFY_4096, POOL_ENCRYPT_4096,
          POOL_DECRYPT_4096,
      POOL_MISC_4096 },
#  endif
# endif /* INCLUDE_SECOND_SET */
    { NULL,                 NULL,               0 }
};

/**
    ******************************************************************************
    */
/*
    Main
 */

# ifdef STATS
#  include <fcntl.h>
#  ifdef USE_HIGHRES_TIME
#   define TIME_STRING "\t%lld"
#  else
#   define TIME_STRING "\t%d"
#  endif
# endif

int main(int argc, char **argv)
{
    psPool_t *pool, *misc;
    psRsaKey_t privkey;
    psSize_t keysize;
    unsigned char *in, *out, *savein, *saveout;
```

```
    psTime_t start, end;
    uint32 iter, i = 0;
    int32 t;
    pkaCmdInfo_t *pkaInfo;

# ifdef STATS
    FILE *sfd;
# endif

    pool = misc = NULL;
    if (psCryptoOpen(PSCRYPTO_CONFIG) < PS_SUCCESS)
    {
        _psTrace("Failed to initialize library: psCryptoOpen
            failed\n");
        return -1;
    }

    pkaInfo = NULL;

    /* Verify time sanity */
    psGetTime(&start, NULL);
    sleep(1);
    psGetTime(&end, NULL);
    printf("Time sanity, 1 second = " TIME_UNITS "\n", psDiffMsecs
        (start, end, NULL));

    _psTraceStr("Aufgabe2 Test mit RSA 2048 bit und 4096 bit, je
        1000 Iterationen...\n", NULL);
# ifdef STATS
    if ((sfd = fopen("perfstat.txt", "w")) == NULL)
    {
        return PS_FAILURE;
    }
#  ifdef USE_HIGHRES_TIME
    fprintf(sfd, "Key\tSign(usec)\tVerify\tEncrypt\tDecrypt\n");
#  else
    fprintf(sfd, "Key\tSign(msec)\tVerify\tEncrypt\tDecrypt\n");
#  endif
# endif /* STATS */

    while (keys[i].key != NULL)
    {
        _psTraceStr("Test %s...\n", keys[i].name);
# ifdef STATS
        fprintf(sfd, "%s", keys[i].name);
# endif
        psRsaInitKey(misc, &privkey);
        if (psRsaParsePkcs1PrivKey(misc, keys[i].key, keys[i].len,
            &privkey) < 0)
```

```
        {
            _psTrace("  FAILED TO PARSE PRIVATE KEY\n");
            continue;
        }
        /* psRsaParsePkcs1PrivKeyOld(misc, keys[i].key, keys[i].
            len, &privkey); */
        keysize = psRsaSize(&privkey);
        savein = in = psMalloc(misc, keysize);
        psGetEntropy(in, keysize, NULL);
        saveout = out = psMalloc(misc, keysize);

# ifdef SIGN_OP
        iter = 0;
        psGetTime(&start, NULL);
        while (iter < keys[i].iter)
        {
            if (psRsaEncryptPriv(pool, &privkey, in, keysize - 16,
                    out, keysize, pkaInfo) < 0)
            {
                _psTrace("  FAILED SIGNATURE OPERATION\n");
            }
/*
            The idea here is that I wanted the encryption to
                happen over
            an ever-changing value without adding any timing
                overhead by
            going out and getting more random data (or whatever).
                 So just
            passing the output as the basis for the input each
                time.
 */
            in = out;
/**/
            if (iter % 2)
            {
                out = saveout;
            }
            else
            {
                out = savein;
            }
            iter++;
        }
        psGetTime(&end, NULL);

        _psTraceInt(TIME_UNITS "/sig ",
            t = psDiffMsecs(start, end, NULL) / keys[i].iter);
        _psTraceInt("(%d pro Sekunde)\n", PER_SEC(t));
```

```
#  ifdef STATS
        fprintf(sfd, TIME_STRING, t);
#  endif
# endif /* SIGN_OP */

# ifdef VERIFY_OP
        static const unsigned char sigdata[] = "Test message to be
            signed - at least 28 bytes";
        memset(in, 0x0, keysize);
        memcpy(in, sigdata, sizeof(sigdata));
        if (psRsaEncryptPriv(misc, &privkey, in, sizeof(sigdata),
            out, keysize, pkaInfo) < 0)
        {
            _psTrace("  FAILED VERIFY PREP\n");
        }
        memset(in, 0x0, keysize);

        psGetTime(&start, NULL);
        /* coverity[swapped_arguments] */
        if (psRsaDecryptPub(pool, &privkey, out, keysize, in,
            sizeof(sigdata), pkaInfo) < 0)
        {
            _psTrace("  FAILED VERIFY OPERATION\n");
        }
        psGetTime(&end, NULL);
        if (memcmp(in, sigdata, sizeof(sigdata)) != 0)
        {
            _psTrace("  FAILED VERIFY VERIFY\n");
        }
        _psTraceInt(TIME_UNITS "/verify ", t = psDiffMsecs(start,
            end, NULL));
        _psTraceInt("(%d per sec)\n", PER_SEC(t));
#  ifdef STATS
        fprintf(sfd, TIME_STRING, t);
#  endif
# endif /* VERIFY_OP */

# ifdef ENCRYPT_OP
        iter = 0;
        psGetEntropy(in, keysize, NULL);
        psGetTime(&start, NULL);
        while (iter < keys[i].iter)
        {
            if (psRsaEncryptPub(pool, &privkey, in, keysize - 16,
                    out, keysize, pkaInfo) < 0)
            {
                _psTrace("  FAILED ENCRYPT OPERATION\n");
            }
/*
```

```
            The idea here is that we wanted the encryption to
                happen over
            an ever-changing value without adding any timing
                overhead by
            going out and getting more random data (or whatever).
                 So just
            passing the output as the basis for the input each
                time.
 */
            in = out;
/**/
            if (iter % 2)
            {
                out = saveout;
            }
            else
            {
                out = savein;
            }
            iter++;
        }
        psGetTime(&end, NULL);
        _psTraceInt(TIME_UNITS "/encrypt ",
            t = psDiffMsecs(start, end, NULL) / keys[i].iter);
        _psTraceInt("(%d per sec)\n", PER_SEC(t));
#  ifdef STATS
        fprintf(sfd, TIME_STRING, t);
#  endif
# endif /* ENCRYPT_OP */

# ifdef DECRYPT_OP
/**/
        if (in == out)
        {
            out = saveout;
        }
        memset(in, 0x0, keysize);
        memcpy(in, "hello", 5);
        if (psRsaEncryptPub(misc, &privkey, in, 5, out, keysize,
            pkaInfo) < 0)
        {
            _psTrace("  FAILED VERIFY PREP\n");
        }
        memset(in, 0x0, keysize);

        psGetTime(&start, NULL);
        /* coverity[swapped_arguments] */
        if (psRsaDecryptPriv(pool, &privkey, out, keysize, in, 5,
            pkaInfo) < 0)
```

```
        {
            _psTrace("  FAILED DECRYPT OPERATION\n");
        }
        psGetTime(&end, NULL);
        if (memcmp(in, "hello", 5) != 0)
        {
            _psTrace("  FAILED DECRYPT VERIFY\n");
        }
        _psTraceInt(TIME_UNITS "/decrypt ", t = psDiffMsecs(start,
            end, NULL));
        _psTraceInt("(%d per sec)\n", PER_SEC(t));
#  ifdef STATS
        fprintf(sfd, TIME_STRING "\n", t);
#  endif
# endif /* DECRYPT_OP */

        psFree(savein, misc);
        psFree(saveout, misc);
        psRsaClearKey(&privkey);
        i++;
    }

# ifdef STATS
    fclose(sfd);
# endif
# ifdef WIN32
    _psTrace("Press any key to close");
    getchar();
# endif
    _psTraceStr("Test abgeschlossen\n", NULL);
    psCryptoClose();
    return 0;
}
#else /* --> !USE_RSA || !USE_PRIVATE_KEY_PARSING*/
int main(int argc, char **argv)
{
# ifndef USE_RSA
    _psTrace("Please enable USE_RSA for this test\n");
# endif /* USE_RSA */
# ifndef USE_PRIVATE_KEY_PARSING
    _psTrace("Please enable USE_PRIVATE_KEY_PARSING for this test\
        n");
# endif /* USE_PRIVATE_KEY_PARSING */
    return 1;
}
#endif /* USE_RSA && USE_PRIVATE_KEY_PARSING */
```

Listing D.1: RSA implementation with MatrixSSL

Bibliography

[Bora] *BorinSSL*. June 25, 2017. URL: https://boringssl.googlesource.com/boringssl/.

[Borb] *Google kündigt OpenSSL-Fork „BoringSSL“ an*. June 25, 2017. URL: http://www.zdnet.de/88196490/google-kuendigt-openssl-fork-boringssl/?inf_by=594f7178681db839228b4a9f.

[Borc] *OpenSSL: Generating an RSA Key From the Command Line*. June 25, 2017. URL: https://rietta.com/blog/2012/01/27/openssl-generating-rsa-key-from-command/.

[Bord] *Porting from OpenSSL to BoringSSL*. June 25, 2017. URL: https://boringssl.googlesource.com/boringssl/+/HEAD/PORTING.md.

[Liba] *LibreSSL*. June 25, 2017. URL: https://www.libressl.org/.

[Libb] *LibreSSL Goals*. June 25, 2017. URL: https://www.libressl.org/goals.html.

[Libc] *libressl-portable/portable*. June 25, 2017. URL: https://github.com/libressl-portable/portable.

[Libd] *Linux: LibreSSL*. June 25, 2017. URL: http://penzin.net/libressl.html.

[Libe] *pem_read_rsapublickey(3) - Linux man page*. June 25, 2017. URL: https://linux.die.net/man/3/pem_read_rsapublickey.

[Libf] *portable/README.windows*. June 25, 2017. URL: https://github.com/libressl-portable/portable/blob/master/README.windows.

[Libg] *The Heartbleed Bug*. June 25, 2017. URL: http://heartbleed.com/.

[Libh] *What's wrong with the OpenSSL Api?* June 25, 2017. URL: https://www.openbsd.org/papers/libtls-fsec-2015/mgp00007.html.

[Mata] *Download mbed TLS and PolarSSL*. June 29, 2017. URL: https://tls.mbed.org/download.

[Matb] *How to encrypt and decrypt with RSA*. June 29, 2017. URL: https://tls.mbed.org/kb/how-to/encrypt-and-decrypt-with-rsa.

[Matc] *Intel® Core™ i3-370M Prozessor*. June 29, 2017. URL: https://ark.intel.com/de/products/49020/Intel-Core-i3-370M-Processor-3M-cache-2_40-Ghz.

[Matd] *jedisct1/libsodium*. June 29, 2017. URL: https://github.com/jedisct1/libsodium.

[Mate] *MatrixSSL 3.9.3 Open*. June 29, 2017. URL: https://github.com/matrixssl/matrixssl/releases/tag/3-9-3-open.

[Matf] *RSA Key Pair generator*. June 29, 2017. URL: https://tls.mbed.org/kb/cryptography/rsa-key-pair-generator.

[Matg] *The GnuTLS Transport Layer Security Library*. June 29, 2017. URL: http://www.gnutls.org/download.html.

[Opea] *AES NI*. June 29, 2017. URL: http://www.thinkwiki.org/wiki/AES_NI.

[Opeb] *Changelog for VirtualBox 5.0*. June 23, 2017. URL: https://www.virtualbox.org/wiki/Changelog-5.0.

[Opec] *Intel® Data-Protection-Technik mit AES-NI und Secure Key*. June 23, 2017. URL: https://www.intel.de/content/www/de/de/architecture-and-technology/advanced-encryption-standard--aes-/data-protection-aes-general-technology.html.

[Oped] *Main Page*. June 22, 2017. URL: https://wiki.openssl.org/index.php/Main_Page.

[Opee] *Manpages*. June 27, 2017. URL: https://www.openssl.org/docs/manpages.html.

[Opef] *OpenSSL*. June 16, 2017. URL: https://www.openssl.org/.

[Opeg] *openssl/openssl*. June 17, 2017. URL: https://github.com/openssl/openssl.

[Opeh] *Where is the documentation?* June 29, 2017. URL: https://www.openssl.org/docs/faq.html#MISC2.

[Ris13] Ivan Ristić. *OpenSSL Cookbook*. 2013.

[Sch13] Klaus Schmeh. *Kryptografie*. 2013.