Appendix to the
CrypTool Book: Learning and Experiencing Cryptography with CrypTool and SageMath

Short Introduction into the CLI of OpenSSL

Bernhard Esslinger

Appendix to the „CrypTool Book“:
Learning and Experiencing Cryptography with CrypTool and SageMath

Short Introduction into the CLI OpenSSL

Bernhard Esslinger

https://www.cryptool.org

Saturday 29th June, 2024 (17:57:00)

Suggestion for referencing with Bib(La)TEX:

```
@article{Esslinger:ctb_2024_app_openssl_en,
  editor    = {Bernhard Esslinger},
  title     = {{A}ppendix to the {C}ryp{T}ool {B}ook:
               {L}earning and {E}xperiencing {C}ryptography
               with {C}ryp{T}ool and {S}age{M}ath:
               {S}hort {I}ntroduction into the {CLI} {OpenSSL}},
  publisher = {CrypTool Project},
  year      = {2024}
}
```

Typesetting software: LaTeX
Version control software: Git
Support with translations: DeepL Translator by DeepL.com and ChatGPT 4 by openai.com
Supporting the generation of Tikz diagram: Claude 3.5 Sonnet by claude.ai

Table of Contents

1 Short Introduction into the CLI `openssl`

(Bernhard Esslinger. Last update: Jun 2024)

OpenSSL[1] is a very widespread free open-source crypto library. It is e. g. in productive use for providing TLS or SSL functionality on web servers. OpenSSL is the first open-source program which is certified by FIPS 140-2.

But with the command line program you not only can map the TLS protocol and manage certificates, but also execute many cryptographic functions in a standalone manner (but only for productive parameter sets, not for small values like those used in didactic learning programs). Under Unix, OpenSSL is usually installed by default.[2]

OpenSSL also contains the command line interface (CLI) called `openssl` that allows using this functionality directly on many operating systems.[3]

In the following we implement many examples from cryptography with the CLI `openssl` and where necessary provide supplementing Python scripts. The code is executable and was tested with the OpenSSL versions 1.1.1 (under Ubuntu 20.04) and 3.0 (under Ubuntu 22.04). About OpenSSL in a browser see Section 1.14 on page 27.

Checking your OpenSSL version

To get the version number of the OpenSSL version installed at your side you can use one of following three commands:

```
$ openssl version
$ python -c "import ssl; print(ssl.OPENSSL_VERSION)"
$ openssl
OpenSSL> version
```

The result is always the same. For example:

```
OpenSSL 3.3.0 9 Apr 2024 (Library: OpenSSL 3.3.0 9 Apr 2024)
```

To get complete version information including the used compiler flags during building OpenSSL, use the option `-a`. This version output looks different on different operating systems. It is most detailed under Linux. Here is the output from a WebAssembly version of OpenSSL in CTO:

```
$ openssl version -a
OpenSSL 3.3.0 9 Apr 2024 (Library: OpenSSL 3.3.0 9 Apr 2024)
built on: Tue Apr 16 06:02:16 2024 UTC
platform: linux-x32
options:  bn(64,64)
compiler: /emsdk/upstream/emscripten/emcc -mx32 -Wall -O3 -DOPENSSL_USE_NODELETE -DL_ENDIAN -▸
 ▸DOPENSSL_BUILDING_OPENSSL -DNDEBUG
OPENSSLDIR: "/usr/local/ssl"
ENGINESDIR: "/usr/local/libx32/engines-3"
MODULESDIR: "/usr/local/libx32/ossl-modules"
Seeding source: os-specific
CPUINFO: N/A
```

[1] Also see https://www.openssl.org/ and https://en.wikipedia.org/wiki/OpenSSL. We provide further commented links at the end of this appendix on page 34.

[2] You can update OpenSSL via `sudo apt-get install openssl`, but mind that this pre-packed version can have major differences compared to the newest stable version that you have to build from source code (when writing the first version of this appendix, we got with `apt-get` under Ubuntu 18.04 the OpenSSL major version 1.1.1 dated from September 11, 2019, whereas the last stable version had been the minor version 1.1.1g from April 4, 2020. In April 2024, Ubuntu 22.04 delivered by default the version OpenSSL 3.0.2 from Mar 15, 2022, despite OpenSSL 3.3.0 was already available).

[3] In contrast to the command line tool `gpg` from GNU Privacy Guard (https://en.wikipedia.org/wiki/GNU_Privacy_Guard), which is also very popular, `openssl` also allows calls that go into great detail. At `gpg`, the focus is only on the cipher suites that are in practical use. To our knowledge, simply encrypting exactly one block without padding as in OpenSSL example 1.11.1 of the book does not work with `gpg`. On the other hand, also with OpenSSL, you can call for example RSA and TLS only with certain minimum sizes (such as 512 bit for the length of the RSA modulus).

1.1 Print all openssl commands

A list of available commands is shown when calling

```
$ openssl help
```

The output that follows this command is divided into three groups: the standard commands, the message digest commands and the cipher commands[4].

```
Standard commands
asn1parse         ca                ciphers           cms
crl               crl2pkcs7         dgst              dhparam
dsa               dsaparam          ec                ecparam
enc               engine            errstr            gendsa
genpkey           genrsa            help              list
nseq              ocsp              passwd            pkcs12
pkcs7             pkcs8             pkey              pkeyparam
pkeyutl           prime             rand              rehash
req               rsa               rsautl            s_client
s_server          s_time            sess_id           smime
speed             spkac             srp               storeutl
ts                verify            version           x509

Message Digest commands (see the 'dgst' command for more details)
blake2b512        blake2s256        gost              md4
md5               rmd160            sha1              sha224
sha256            sha3-224          sha3-256          sha3-384
sha3-512          sha384            sha512            sha512-224
sha512-256        shake128          shake256          sm3

Cipher commands (see the 'enc' command for more details)
aes-128-cbc       aes-128-ecb       aes-192-cbc       aes-192-ecb
aes-256-cbc       aes-256-ecb       aria-128-cbc      aria-128-cfb
aria-128-cfb1     aria-128-cfb8     aria-128-ctr      aria-128-ecb
aria-128-ofb      aria-192-cbc      aria-192-cfb      aria-192-cfb1
aria-192-cfb8     aria-192-ctr      aria-192-ecb      aria-192-ofb
aria-256-cbc      aria-256-cfb      aria-256-cfb1     aria-256-cfb8
aria-256-ctr      aria-256-ecb      aria-256-ofb      base64
bf                bf-cbc            bf-cfb            bf-ecb
bf-ofb            camellia-128-cbc  camellia-128-ecb  camellia-192-cbc
camellia-192-ecb  camellia-256-cbc  camellia-256-ecb  cast
cast-cbc          cast5-cbc         cast5-cfb         cast5-ecb
cast5-ofb         des               des-cbc           des-cfb
des-ecb           des-ede           des-ede-cbc       des-ede-cfb
des-ede-ofb       des-ede3          des-ede3-cbc      des-ede3-cfb
des-ede3-ofb      des-ofb           des3              desx
rc2               rc2-40-cbc        rc2-64-cbc        rc2-cbc
rc2-cfb           rc2-ecb           rc2-ofb           rc4
rc4-40            seed              seed-cbc          seed-cfb
seed-ecb          seed-ofb          sm4-cbc           sm4-cfb
sm4-ctr           sm4-ecb           sm4-ofb
```

All commands have options. To show them, enter something like:

```
$ openssl x509 -help
$ openssl enc -help
$ openssl sha3-512 -help
```

[4] However, not included in the cipher list is e. g. the encryption procedure ChaCha20, which in openssl you can only call from within cipher suites and in combination with MAC Poly1305.

1.2 Symmetric encryption with openssl

The symmetric cipher AES needs as input a message m, a session key k, and an initialization vector (IV).[5]

There are several ways in which AES can be invoked with OpenSSL. Two variants are presented here: On the one hand the parameters are explicitly transferred, on the other hand they are derived internally from the transferred password (OpenSSL uses a built-in key derivation function such as PBKDF2). Using a password and a key derivation function is the usual way.

The session key can be placed directly on the command line or read from a file. There are even three options for the password: It can be given directly on the command line or read from a file, or you will be prompted for a password.

1.2.1 Generate a random session key to be used for AES

First, we generate a 256-bit key to be stored in `symkey.bin`. The argument 32 means to get a length of 32 bytes which is 256 bit.

```
$ openssl rand -out symkey.bin 32
```

Then we generate a random IV of length 16 byte = 128 bit = 32 hex digits. Sample output:

```
$ openssl rand -hex 16
cbff22e0ae27abf4892c1437c54dc4ca
```

The values for session key and IV created in advance are only needed in the first variant, where the parameters are explicitly transferred to OpenSSL. In the first variant, both the session key and the IV have to given explicitly.

1.2.2 Encrypt with AES using a random session key (variant 1)

Encrypt the file `message.txt` with AES-256 in cfb8 mode using IV from above as additional input. The output is a file containing the ciphertext which is called `message.txt.enc`. The option `-p` makes OpenSSL show in addition which parameters this command used. The session key is given after `-K` – once directly as hex value and once after it is read from the file `symkey.bin` and stored in a shell variable.

```
$ openssl enc -aes-256-cfb8 -in message.txt -out message.txt.enc -K F8ED266E40B5534E04EC3F0C688B84E9168E8DF8C64F1▸
 ▸598C3D905DD68BBD860 -iv cbff22e0ae27abf4892c1437c54dc4ca  -p

$ variable=$(xxd -p -c 1000 symkey.bin)
$ openssl aes-256-cfb8 -in message.txt -out message.txt.enc -K $variable -iv cbff22e0ae27abf4892c1437c54dc4ca -p

salt=0000000000000000
key=F8ED266E40B5534E04EC3F0C688B84E9168E8DF8C64F1598C3D905DD68BBD860
iv =CBFF22E0AE27ABF4892C1437C54DC4CA
```

1.2.3 Encrypt with AES using a password (variant 2)

Encrypt the file `message.txt` like above but use a password instead of explicit parameters. This is the recommended approach. The command gets the password by one of three different ways: The command prompts for the password, or it is entered after the option `-pass pass:` on the command line, or it is read out after the option `-pass file:` from the file `symkey.bin`.[6] This file has only been created shortly beforehand (like in 1.2.1) to act as random password file. An option `-iv` is optional in all cases.

[5] You also can use `-iter` to specify the iteration count used by the key derivation function (KDF). In addition, as default, the KDF uses a salt value which can be switched off with `-nosalt`.

[6] Old options to read a password are `-k` and `-kfile`. According to `https://wiki.openssl.org/index.php/Enc` these options are depricated and should be substituted by `-pass pass:` and `-pass file:`.

```
$ openssl enc -aes-256-cfb8 -in message.txt -out message.txt.enc -pbkdf2

$ openssl aes-256-cfb8 -in message.txt -out message.txt.enc -pass pass:foobar -pbkdf2

$ openssl rand -out symkey.bin 32
$ openssl aes-256-cfb8 -in message.txt -out message.txt.enc -pass file:symkey.bin -pbkdf2
```

You can also read about this in the OpenSSL online documentation: https://wiki.openssl.org/index.php/Enc

1.2.4 Decrypt a file using AES-256

Decrypt the file `message.txt.enc` using AES-256. To do this, the additional option `-d` is needed. As output, we get the file `message.txt.enc.dec` that contains the original message.

```
$ openssl aes-256-cfb8 -d -in message.txt.enc -out message.txt.enc.dec -pbkdf2
```

The files `message.txt` and `message.txt.enc.dec` must then be equal.[7,8]

1.3 Asymmetric encryption with `openssl`: key generation

For the asymmetric encryption method RSA, each participant first needs a key pair that consists of a public and a private key. Then both sides have to exchange their public key. This creates the necessary "public-key infrastructure".

The sender then encrypts with the recipient's public key, and the recipient decrypts with his private key.

1.3.1 Generate a private RSA key of length 2048 bit

The length of the modulus[9] is considered to be the length of the RSA key.

The following command generates the private key and stores it in PKCS#8 format[10] into the file `privatekey.pem` (base64 encoded). This format consists not only of the key but contains also the OID[11] defining the key type.[12,13]

```
$ openssl genpkey -algorithm RSA -pkeyopt rsa_keygen_bits:2048 -out privatekey.pem
```

An alternative[14] is:

```
$ openssl genrsa -out rsa.key 2048
```

[7] It doesn't matter whether the option `enc -aes` or the option `aes` for `openssl` is used, that is, `aes` *without* the leading "-", cf. listing on p. 12.

[8] For the day-to-day usage of an end user, for symmetric encrypting/decrypting GnuPG or Veracrypt are recommended instead of openssl, because those programs are either integrated in (email) applications or directly do the hard drive encryption on various operating systems.

[9] For singular, we use "modulus" and for plural "moduli". Wikipedia does it in the same way e. g. for "RSA modulus" and "RSA moduli" in https://en.wikipedia.org/wiki/RSA_(cryptosystem).

[10] PKCS stands for "public-key cryptography standards", see RFC 5208 (https://datatracker.ietf.org/doc/html/rfc5208).

[11] Object Identifier, see https://www.ietf.org/rfc/rfc3447.txt

[12] Small keys like in student exercises are not allowed in `openssl`. OpenSSL 1.1.1 requires at least 512 bit for an RSA key.

[13] `genrsa` also has an option to use the number 3 instead of 65537 as public exponent. And with the option `-primes` you can generate a so-called "multi-prime" RSA key. See the documentation at https://www.openssl.org/docs/manmaster/man1/genrsa.html.

[14] On the website of OpenSSL the user is actually encouraged **not** to use this alternative, see https://www.openssl.org/docs/manmaster/man1/openssl-genpkey.html: "NOTES: The use of the genpkey program is encouraged over the algorithm specific utilities because additional algorithm options and ENGINE provided algorithms can be used."

Those two calls are equivalent: `openssl genpkey -algorithm RSA` and `openssl genrsa` both generate a file in the base64-encoded, platform-independent PEM format[15].

You also can AES-encrypt the private key at once by adding the option `-aes256`. The passphrase (4 to 1023 characters) can be delivered either via prompt or at the CLI:

a) Here the pass phrase is prompted and has to be entered twice:

```
$ openssl genpkey -algorithm RSA -pkeyopt rsa_keygen_bits:2048 -out privatekey2.pem -aes256
```

b) Here the pass phrase ("test") is delivered directly at the command line (no prompt):

```
$ openssl genpkey -algorithm RSA -pkeyopt rsa_keygen_bits:2048 -out privatekey2.pem -aes256 -pass pass:test
```

1.3.2 OpenSSL file "private key"

In mathematics the private RSA key consists exactly of the modulus `n` and the decryption exponent `d`. This is correct.

For implementation reasons, in `openssl` the file called "private key" contains all relevant information created during key generation: so not only `n` and `d`, but also the values of the public key (`n` and the encryption exponent `e`), the two primes `p` and `q`, the length of the modulus, and three further values which can be used to speed up the decryption.[16]

The actual content of the private key can be shown with the following command:[17,18]

```
$ openssl pkey -in privatekey.pem -text
-----BEGIN PRIVATE KEY-----
MIIEwAIBADANBgkqhkiG9w0BAQEFAASCBKowggSmAgEAAoIBAQDNKnRwj+TRx5ru
icDrwlBUU0UdP9HJOxyFjzsuy/OvwqOKYVLABec97N/4fFEmvDvoAyMHZWCReO3c
...
-----END PRIVATE KEY-----
Private-Key: (2048 bit)
modulus:
    00:cd:2a:74:70:8f:e4:d1:c7:9a:ee:89:c0:eb:c2:
    50:54:53:45:1d:3f:d1:c9:3b:1c:85:8f:3b:2e:cb:
    ...
    09:f8:4f:e8:9b:65:b7:78:87:5f:2f:73:49:f4:6c:
    e6:17
publicExponent: 65537 (0x10001)
privateExponent:
    00:b9:af:3c:e7:4b:34:3b:30:be:66:39:c2:a3:1d:
    a0:7a:51:4a:f2:27:fa:84:77:bd:5e:9b:bd:62:a3:
    ...
    33:d5:fe:40:9c:04:90:71:91:a5:cd:99:10:f9:dd:
    be:c1
prime1:
    00:f0:7e:2c:8e:03:5a:55:c3:47:3a:98:e4:8e:d7:
    ...
    b1:9a:76:02:2b:bc:e7:f6:11
prime2:
    00:da:65:22:d4:24:d6:08:da:cc:64:38:44:56:0e:
    ...
    27:ff:f5:ea:01:d0:40:51:a7
exponent1:
```

[15] PEM stands for *privacy enhanced mail*, cf. https://datatracker.ietf.org/doc/html/rfc1421.

[16] More information about these additional exponents can be found in the "Example" section of the RSA article in Wikipedia.

- d = privateExponent
- n = modulus
- e = publicExponent
- p = prime1
- q = prime2
- d_p = exponent1
- d_q = exponent2
- q_inv = coefficient

[17] If you add the option `-noout`, you get the same output without the base64 preamble between `BEGIN PRIVATE KEY` and `END PRIVATE KEY`.

[18] How to get those values as integers in decimal notation we show in Section 1.6 on page 12.

```
    00:da:bb:77:9c:a6:b2:07:e4:f7:a8:f5:1c:94:4a:
    ...
    a0:eb:82:bf:90:b6:5d:27:71
exponent2:
    00:bd:42:8b:ee:40:85:e3:62:89:62:08:88:df:f2:
    ...
    08:73:fe:ce:6e:07:e4:d5:5d
coefficient:
    00:ce:bc:f6:f1:fd:64:c3:8f:36:7c:c9:da:64:3a:
    ...
    45:2a:a9:f6:bb:da:f0:12:cd
```

1.3.3 OpenSSL file "public key"

When generating the key `openssl` generates only the private key that we had saved as `privatekey.pem`. This file also contains all the information for the public key. Therefore, with `openssl` (but not generally with RSA) we can "extract" the public key from the private key file.[19]

To complete the key pair, we now generate the public key from the private key (only the public key will be made public). We have to specify a different name for the public key.[20]

```
$ openssl pkey -in privatekey.pem -out publickey.pem -pubout
```

With the following command you can see the details of the public key (it only contains the length of the modulus, its value in hex and the public exponent):[21]

```
$ openssl pkey -in publickey.pem -pubin -text -noout
Public-Key: (2048 bit)
Modulus:
    00:cd ...
Exponent: 65537 (0x10001)
```

If one is interested only in the modulus, you can use:

```
$ openssl rsa -in publickey.pem -pubin -noout -modulus
Modulus=CD2A74708FE4D1C79AEE89 ... F46CE617
```

Using the same option also allows getting the modulus from the private key:

```
$ openssl rsa -in privatekey.pem -noout -modulus
```

1.4 Asymmetric encryption with `openssl`: still no hybrid encryption

The key pair generated in Section 1.3 is now used here to encrypt a file.

[19] But keep in mind that you cannot derive a private key from a public key, even when using `openssl`.

[20] It is always better to use the internal option `-out` instead of redirecting `stdout` to a file:
`openssl pkey -in privatekey.pem -pubout > publickey.pem`
As an alternative there is a deprecated command: `$ openssl rsa -in privatekey.pem -out publickey.pem -pubout`

[21] How to get all values of a key as a decimal integer is shown in Section 1.6 on page 12. Here are two ways how to convert hex numbers into decimal numbers:
- in Python:
  `s="6a48f82d8e828ce82b82"; i = int(s, 16); s=str(i); i; s`
- on the command line via the BC tool:
  ```
  $ echo "ibase=16; 6A48F82D8E828CE82B82"|bc
  501916895863876199394178
  ```

1.4.1 Encryption with RSA (no textbook RSA)

We want to encrypt a file called `message.txt` with `openssl` and use the public key in the file `publickey.pem`. Our encrypted file is called `message.txt.rsaenc`.[22,23]

```
$ openssl pkeyutl -encrypt -inkey publickey.pem -pubin -in message.txt -out message.txt.rsaenc
```

1.4.2 Decryption with RSA (no textbook RSA)

We want to decrypt the file `message.txt.rsaenc` with `openssl` using the according private key from `privatekey.pem`. We call our decrypted file `message.decrypted.txt`.

```
$ openssl pkeyutl -decrypt -inkey privatekey.pem -in message.txt.rsaenc -out message.decrypted.txt
```

1.5 Hybrid encryption with `openssl`

Hybrid encryption is a combination of asymmetric and symmetric encryption. The sender chooses a random symmetric key (called a session key). This session key is used to symmetrically (e. g. with AES) encrypt the (potentially very long) data to be protected. The session key is then encrypted asymmetrically (e. g. with RSA) with the recipient's public key. This procedure solves the key distribution problem and maintains the speed advantage of symmetric encryption. Also see https://en.wikipedia.org/wiki/Hybrid_cryptosystem.

Hybrid scheme variants are used e. g. for secure email and HTTPS.

Preliminary remark

All commands in this Subsection 1.5 are contained in the shell script `hybrid-openssl-enc-dec.sh` for direct execution. You can find its listing (1.2) at the end of this section or on the website: https://www.cryptool.org/en/ctbook/openssl

In our scenario Alice and Bob communicate.

[22] In general, for encrypting with RSA the length of `message.txt` must be less than the size of the RSA modulus – if the message file is not split into smaller parts. However, `openssl` uses as default RSA-OAEP instead of textbook RSA, so the input has to be even smaller. OAEP stands for "Optimal Asymmetric Encryption Padding" and should – according to RFC 8017 (https://datatracker.ietf.org/doc/html/rfc8017) – be used to add a component of randomness to the deterministic RSA algorithm. When doing so the result has a small overhead which reduces the maximal encryptable message size (which is possible in the case of textbook RSA) by some bytes. Details can be found e. g. on `crypto.stackexchange.com` (https://crypto.stackexchange.com/questions/42097/what-is-the-maximum-size-of-the-plaintext-message-for-rsa-oaep). There you can find especially this post from 2017 (by Maarten Bodewes): *"For PKCS#1 v1.5 padding, for encryption the overhead is simply 11 bytes (three bytes static values and 8 bytes filled with random numbers 1...255)."*
Example: If the modulus of Bob's public key has the length `2048bit` you can encrypt a file of length $245\text{ B} = (256 - 11)\text{B} = 1960$ bit, but not a file of length $(256 - 10)\text{B} = 246\text{ B} = 1968$ bit. In the following listing we generate two random messages, one with 245 B (we call it `x245`, per default in binary format) and one with 246 B (called `y246`, also in binary format) using the command `rand`. Then we (at least try to) encrypt both messages:

```
$ openssl rand -out x245 245
$ openssl rand -out y246 246
$ openssl genrsa -out key.pem 2048
$ openssl pkey -in key.pem -out pubkey.pem -pubout
$ openssl pkeyutl -encrypt -pubin -inkey pubkey.pem -in x245 -out x245enc
# no error
$ openssl pkeyutl -encrypt -pubin -inkey pubkey.pem -in y246 -out y246enc
RSA operation error
4354178496:error:0406D06E:rsa routines:RSA_padding_add_PKCS1_type_2:data too large for key size:crypto/rsa/rsa_pk
  1.c:125:
```

[23] How to perform the insecure textbook RSA for didactic purposes with `openssl` is described in Section 1.9.

1.5.1 Pre-tasks at the receiver Bob: key generation (RSA)

Bob generates his key pair and saves his private key into a file. Then he generates his public key from his private key.

```
$ openssl genpkey -algorithm RSA -pkeyopt rsa_keygen_bits:2048
  -out privatekey.Bob.pem
$ openssl pkey -in privatekey.Bob.pem -out publickey.Bob.pem -pubout
```

Bob sends his public key to Alice.

1.5.2 Encryption: Three tasks at the site of sender Alice

1. Alice creates a random session key and stores it in the file `symkey.bin`.

```
openssl rand -out symkey.bin  32
```

2. Alice encrypts the session key `symkey.bin` using RSA with Bob's public key.

```
$ openssl pkeyutl -encrypt -pubin -inkey publickey.Bob.pem -in symkey.bin -out symkey.bin.rsa-encrypted-for-bob
```

3. Alice encrypts her message with AES using `symkey.bin` as session key.

```
$ openssl enc -aes-256-cfb8 -in message.txt -out message.txt.enc -pass file:symkey.bin -pbkdf2
```

Alice sends two things to Bob:

- the with the session key symmetrically encrypted message as file "message.txt.enc", and
- the sessionskey asymmetrically encrypted with Bob's public key as file: `symkey.bin.rsa-encrypted-for-bob`

Alice should remove the session key afterward.

1.5.3 Decryption: Two tasks at the site of the receiver Bob

1. Bob decrypts the session key with his private RSA key.

```
openssl pkeyutl -decrypt -inkey privatekey.Bob.pem -in symkey.bin.rsa-encrypted-for-bob -out symkey-decrypted.bin
```

2. Bob uses the session key to decrypt the actual data (message) with AES.

```
openssl aes-256-cfb8 -d -in message.txt.enc -out message.txt.enc.dec  -pass file:symkey.bin.rsa-encrypted-for-bob
  -pbkdf2
```

1.6 Show all keys of a private PEM file as decimal numbers (using an own Python script)

The script `convert-pem-dec.py` converts the found hex numbers in a private or public PEM file into decimal numbers. To do so, it searches the text form of the corresponding text form of the key file for the occurrence of the strings "RSA private key" and "RSA public key".

Note: The private PEM file generated by OpenSSL is often also called "private key" in OpenSSL. See Subsection 1.3.2.

Call for a private PEM file:

```
$ openssl pkey -in privatekey.Bob.pem -noout -text > privatekey.Bob.txt
$ ./convert-pem-dec.py privatekey.Bob.txt
```

The same goes for the public PEM file with the following call:

```
$ openssl pkey -in publickey.Bob.pem -pubin -noout -text > publickey.Bob.txt
$ ./convert-pem-dec.py publickey.Bob.txt -pubin
```

Here a possible output of a public key (modulus N, exponent e):

```
$ ls -l *.Bob.txt
-rw-r--r-- 1 bet bet 4010 Jun 13 23:45 privatekey.Bob.txt
-rw-r--r-- 1 bet bet  922 Jun 13 23:50 publickey.Bob.txt

$ cat publickey.Bob.txt
RSA Public-Key: (2048 bit)
Modulus:
    00:cb:e8:77:13:ac:77:67:de:bc:57:b4:5c:73:4b:
    99:49:92:92:71:95:8a:b8:3b:a3:c5:9b:30:e7:b0:
    4d:5a:b7:e5:af:ea:b0:92:43:db:7c:39:90:c6:dd:
    96:a0:de:2d:61:4d:73:51:fa:9c:35:c0:76:1b:ac:
    2c:b1:da:e5:f9:26:22:0d:2d:c4:56:5f:7e:aa:84:
    87:0e:84:a5:11:68:47:9a:5b:d7:6b:e8:2a:b0:79:
    35:8d:cf:17:d7:56:42:f8:57:39:c9:96:23:1c:b0:
    9c:19:c2:3e:62:67:de:06:b7:66:65:b0:70:6d:5a:
    b7:2b:20:13:df:5a:a0:65:fd:59:bd:7f:d9:71:fd:
    01:3e:e8:43:b3:6f:56:14:54:a6:77:b6:3c:e5:19:
    46:6b:0d:7b:10:ca:72:b8:2a:41:05:9e:5c:73:d1:
    e2:46:b6:b3:dc:4a:21:a1:af:08:80:ae:2c:9f:52:
    68:83:c1:7b:04:05:ab:13:f3:e9:fb:ca:8e:9a:d6:
    11:29:28:ac:aa:dd:7b:58:e1:4b:2e:d2:53:fd:f0:
    e4:1d:96:90:07:cd:5c:a0:a6:e6:fe:f0:8c:c5:72:
    73:e3:10:73:16:21:80:31:90:9c:eb:07:d0:98:47:
    16:77:78:bb:00:0b:f2:e6:61:24:78:b2:39:00:bb:
    76:63
Exponent: 65537 (0x10001)

$ ./convert-pem-dec.py publickey.Bob.txt -pubin
Modulus:
    n =25741008743014329602130075944926942393815840463882852482920558426679866327952824599118760618782596588503204928822353751318706500956045565322281328491894883011550481252689083876294733091597440056392676782832418868550497092915089476440069013656369919580674215699486371723448625643678604815254035894944521623776040635114105544615730756280024048599158725479299023271801879702500078406803579754940027584133355614742986888643828833692155604696172500660504301425627619980135474110408212153438671573134485043911241953958498315793466069332534088446495105840432581716757579981700370832452879603918850204752639915161466932182 7
Exponent:
    e = 65537
```

1.7 Show keys of a PEM file as decimal numbers (via RsaCtfTool)

The so-called "RSA attack tool" RsaCtfTool[24] is mainly used for CTFs[25] in order to retrieve a private key from a weak public key or to decrypt a ciphertext decrypted with textbook RSA. It also can be used to dump the parameters (numbers) from a key in PEM or DER format.

RsaCtfTool also recognizes the PEM format used by OpenSSL to store asymmetric keys.

The 2 commands from above

```
$ openssl pkey -in privatekey.pem -noout -text > privatekey.txt
```

[24] https://github.com/RsaCtfTool/RsaCtfTool

[25] https://en.wikipedia.org/wiki/Capture_the_flag_(cybersecurity)

`$ ./convert-pem-dec.py privatekey.txt`
are equivalent to the following command:
`$ ./RsaCtfTool/RsaCtfTool.py --dumpkey --key privatekey.pem --private`

1.8 Overview of previous OpenSSL commands (as list and in shell script)

In the previous examples, a random session key, a random IV, and RSA key pairs for Alice and Bob were generated. Then an encrypted file was generated from Alice for Bob (once only RSA was used, once hybrid encryption was used). Finally, the values of the parameters in the PEM files were output in hex and decimal. This is summarized in the following two examples: The script in the OpenSSL example 1.2 does a little more than the command list in the OpenSSL example 1.1.

OpenSSL Example 1.1: The previous tasks as a call sequence

```
echo "sultans of swing meet sympathy for the devil" > message.txt

openssl rand -out symkey.bin 32
openssl rand -hex 16  > IV.hex
cat IV.hex
openssl enc -aes-256-cfb8 -in message.txt -out message.txt.enc -pass file:symkey.bin -pbkdf2 -iv "$(cat IV.hex)"
openssl enc -d -aes-256-cfb8 -in message.txt.enc -out message.txt.enc.dec -pass file:symkey.bin -pbkdf2 -iv "$(cat IV.hex)"

openssl genpkey -algorithm RSA -pkeyopt rsa_keygen_bits:4096 -out privatekey.pem
openssl pkey -in privatekey.pem -text
openssl pkey -in privatekey.pem -out publickey.pem -pubout
openssl pkey -in publickey.pem -pubin -text

openssl pkeyutl -encrypt -inkey publickey.pem -pubin -in message.txt -out message.txt.rsaenc
openssl pkeyutl -decrypt -inkey privatekey.pem -in message.txt.rsaenc -out message.decrypted.txt

openssl genpkey -algorithm RSA -pkeyopt rsa_keygen_bits:2048 -out privatekey.Bob.pem
openssl pkey -in privatekey.Bob.pem -out publickey.Bob.pem -pubout

echo "- A creates a random symmetric password and stores it in the file symkey.bin"
openssl rand -out symkey.bin  32
openssl pkeyutl -encrypt -pubin -inkey publickey.Bob.pem -in symkey.bin -out symkey.bin.rsa-encrypted-for-bob
openssl enc -aes-256-cfb8 -in message.txt -out message.txt.enc -pass file:symkey.bin -pbkdf2

echo "- B decrypts the random symmetric password and stores it in the file symkey.bin.rsa-encrypted-for-bob.dec"
openssl pkeyutl -decrypt -inkey privatekey.Bob.pem -in symkey.bin.rsa-encrypted-for-bob -out symkey.bin.rsa-encrypted-for-bob.dec
openssl aes-256-cfb8 -d -in message.txt.enc -out message.txt.enc.dec -pass file:symkey.bin.rsa-encrypted-for-bob.dec -pbkdf2
openssl pkey -in privatekey.Bob.pem -noout -text > privatekey.Bob.txt
./convert-pem-dec.py privatekey.Bob.txt
```

OpenSSL Example 1.2: Shell script `hybrid-openssl-enc-dec.sh`

```
#!/bin/bash
# BE 2024-04-13 for OpenSSL 3
#
# Apply some typical OpenSSL commands.

echo
openssl version


# create a message
echo "sultans of swing meet sympathy for the devil" > message.txt



printf "\r\n### (1) Symmetric Encryption and Decryption with OpenSSL\n"

echo "- Generate a random symmetric key (= sessionkey) to be used for AES (32 byte = 256 bit)"
openssl rand -out symkey.bin 32

echo "- Generate a random initialization vector (IV) of length 16 byte = 128 bit = 64 hex digits"
openssl rand -hex 16  > IV.hex
cat IV.hex

echo "- Encrypt file with AES using the session key (here IV is read from file)"
openssl enc -aes-256-cfb8 -in message.txt -out message.txt.enc -k file:symkey.bin -pbkdf2 -iv "$(cat IV.hex)"

echo "- Decrypt file using AES. Needs additional option -d"
openssl enc -d -aes-256-cfb8 -in message.txt.enc -out message.txt.enc.dec -k file:symkey.bin -pbkdf2 -iv "$(cat IV.hex)"

echo "- Check AES correctness: message.txt ?= message.txt.enc.dec"
cmp -s "message.txt" "message.txt.enc.dec"
CMPRESULT=$?
if [ $CMPRESULT -eq 0 ]; then
  echo "files are equal"
  cat message.txt
```

OpenSSL Example 1.2 ctd.

```
elif [ $CMPRESULT -eq 1 ]; then
  echo "files are not equal"
  cat message.txt
  cat message.txt.enc.dec
else
  echo "file cmp error"
fi



printf "\r\n### (2) Asymmetric Encryption and Decryption with OpenSSL: Key Generation\n"

echo "- Create an RSA 4096 bit key pair. The private keys is stored in PKCS#8 format in the file privatekey.pem."
openssl genpkey -algorithm RSA -pkeyopt rsa_keygen_bits:4096 -out privatekey.pem

# You also can AES-encrypt the privatekey at once.
# - Here the pass phrase (4 to 1023 characters) is prompted and has to be entered twice:
#   openssl genpkey -algorithm RSA -pkeyopt rsa_keygen_bits:4096 -out privatekey2.pem -aes256
# - Here the pass phrase ("test") is delivered in the script itself (no prompt):
# openssl genpkey -algorithm RSA -pkeyopt rsa_keygen_bits:4096 -out privatekey2.pem -aes256 -pass pass:test

echo "- Look at the details of the private key file, which in openssl also contains the public values and some more values for faster calculations (output only ▸
 ▸in base64 and/or hex)"
openssl pkey -in privatekey.pem -text

echo "- Create the public key (from the private key)"
openssl pkey -in privatekey.pem -out publickey.pem -pubout

echo "- View the details of the public key (it only contains n and e as usual in maths)"
openssl pkey -in publickey.pem -pubin -text



printf "\r\n### (3) Asymmetric Encryption and Decryption with OpenSSL: Textbook RSA\n"

echo "- Encrypt a file called message.txt via RSA and public key"
openssl pkeyutl -encrypt -inkey publickey.pem -pubin -in message.txt -out message.txt.rsaenc

echo "- Decrypt file with the RSA and private key (privatekey.pem)"
openssl pkeyutl -decrypt -inkey privatekey.pem -in message.txt.rsaenc -out message.decrypted.txt

echo "- Check correctness: message.txt ?= message.decrypted.txt"
cmp -s "message.txt" "message.decrypted.txt"
CMPRESULT=$?
if [ $CMPRESULT -eq 0 ]; then
  echo "files are equal"
  cat message.txt
elif [ $CMPRESULT -eq 1 ]; then
  echo "files are not equal"
  cat message.txt
  cat message.decrypted.txt
else
  echo "file cmp error"
fi



printf "\n\n### (4) Hybrid Encryption Using RSA for Sessionkey and AES for (Long) Message File\n"
echo
echo "# Pre-tasks at the receiver Bob (B): Key generation (RSA)"

echo "- B creates his keypair and stores the private key in a file"
openssl genpkey -algorithm RSA -pkeyopt rsa_keygen_bits:2048 -out privatekey.Bob.pem

echo "- B creates a public key from his private key"
openssl pkey -in privatekey.Bob.pem -out publickey.Bob.pem -pubout

# echo "- view the details on the public key with this command"
# $ openssl pkey -in publickey.Bob.pem -pubin -text

echo "- B sends his public key to Alice (A)"


echo
echo "# Tasks at the sender Alice: Encryption"
# Prerequisite: message.txt given

echo "- A creates a random symmetric key and stores it in the file symkey.bin"
openssl rand -out symkey.bin  32

echo "- A encrypts symkey.bin using RSA with B's public key"
openssl pkeyutl -encrypt -pubin -inkey publickey.Bob.pem -in symkey.bin -out symkey.bin.rsa-encrypted-for-bob

echo "- A encrypts message.txt with AES using symkey.bin as sessionkey"
# Im ff. keinen IV anggb: -iv 021afbb5928ac15fa4503d90959ed139
openssl enc -aes-256-cfb8 -in message.txt -out message.txt.enc -k file:symkey.bin -pbkdf2

# remove the sessionkey afterwards
# rm symkey.bin


echo
echo "# Tasks at the receiver Bob: Decryption"

echo "- B decrypts the sessionkey with his private key"
openssl pkeyutl -decrypt -inkey privatekey.Bob.pem -in symkey.bin.rsa-encrypted-for-bob -out symkey-decrypted.bin

echo "- B uses the sessionkey to decrypt the data with AES"
```

OpenSSL Example 1.2 ctd.

```
openssl aes-256-cfb8 -d -in message.txt.enc -out message.txt.enc.dec -k file:symkey.bin -pbkdf2

# remove the sessionkey afterwards
# rm symkey.bin


echo
echo "- Check correctness: message.txt ?= message.txt.enc.dec"
cmp -s "message.txt" "message.txt.enc.dec"
CMPRESULT=$?
if [ $CMPRESULT -eq 0 ]; then
  echo "files are equal"
  cat message.txt
elif [ $CMPRESULT -eq 1 ]; then
  echo "files are not equal"
  cat message.txt
  cat message.txt.enc.dec
else
  echo "file cmp error"
fi



printf "\n\n### (5) Show Numbers of Bob's Private Key as Decimals (Using a Python Script)\n"
openssl pkey -in privatekey.Bob.pem -noout -text > privatekey.Bob.txt
./convert-pem-dec.py privatekey.Bob.txt



printf "\n\n### (6) Generate Random Data\n"

echo "- Generate 32 byte = 256-bit and write them to a file called testkey.bin"
openssl rand -out testkey.bin 32

echo "- Generate 100 byte of random data and write to a file called file.bin"
openssl rand -out file.bin 100

echo "- Generate 100 byte of random data in hexadecimal and redirect output to file.hex"
openssl rand -hex 100 > file.hex

echo "- Generate 100 byte of random data in base64 and redirect output to file.b64"
openssl rand -base64 100 > file.b64

echo "- Show different file sizes"
ls -l file.*



printf "\n\n### (7) Generate Prime Numbers\n"

echo "- Generate a random 4096-bit prime number (takes about 3 sec)"
openssl prime -generate -bits 4096

echo "- Generate small primes (doesn't create all of the given bit length)"
openssl prime -generate -bits 2
openssl prime -generate -bits 3
openssl prime -generate -bits 4
openssl prime -generate -bits 5
openssl prime -generate -bits 6
openssl prime -generate -bits 7
openssl prime -generate -bits 8

echo

: <<KOMMENTARIO
Files initially required in an empty directory:
-rwx--x--x 1 bet bet 3222 Jun  1 17:35 convert-pem-dec.py
-rwxr-xr-x 1 bet bet 8051 Jun  2 04:14 hybrid-openssl-enc-dec.sh

Files created during running the script hybrid-openssl-enc-dec.sh:
-rw-r--r-- 1 bet bet  139 Jun  2 04:18 file.b64
-rw-r--r-- 1 bet bet  100 Jun  2 04:18 file.bin
-rw-r--r-- 1 bet bet  201 Jun  2 04:18 file.hex
-rw-r--r-- 1 bet bet   33 Jun  2 04:18 IV.hex
-rw-r--r-- 1 bet bet   45 Jun  2 04:18 message.decrypted.txt
-rw-r--r-- 1 bet bet   45 Jun  2 04:18 message.txt
-rw-r--r-- 1 bet bet   61 Jun  2 04:18 message.txt.enc
-rw-r--r-- 1 bet bet   45 Jun  2 04:18 message.txt.enc.dec
-rw-r--r-- 1 bet bet  512 Jun  2 04:18 message.txt.rsaenc
-rw------- 1 bet bet 1704 Jun  2 04:18 privatekey.Bob.pem
-rw-r--r-- 1 bet bet 4013 Jun  2 04:18 privatekey.Bob.txt
-rw------- 1 bet bet 3272 Jun  2 04:18 privatekey.pem
-rw-r--r-- 1 bet bet  451 Jun  2 04:18 publickey.Bob.pem
-rw-r--r-- 1 bet bet  800 Jun  2 04:18 publickey.pem
-rw-r--r-- 1 bet bet   32 Jun  2 04:18 symkey.bin
-rw-r--r-- 1 bet bet  256 Jun  2 04:18 symkey.bin.rsa-encrypted-for-bob
-rw-r--r-- 1 bet bet   32 Jun  2 04:18 symkey-decrypted.bin
-rw-r--r-- 1 bet bet   32 Jun  2 04:18 testkey.bin

KOMMENTARIO
```

1.9 Textbook RSA with openssl and own Python script

Preparations

In the following we use the Python script `int2bin.py`. It transforms a natural number into binary format which `openssl` uses for a specific length. This length must be equal to the length of the RSA modulus in the public key that is uses. The following command shows the bit length (key size) of the modulus:

```
$ openssl pkey -in publickey.pem -pubin -noout -text
RSA Public-Key: (4096 bit)
...
```

Step 1: Transform natural number into openssl format

The following command stores the number `m=25` in the same length as the RSA modulus to be used (4096 bit). It is stored into the file "message.bin" which has a length of 4096 bit $=$ 512 B:

```
$ ./int2bin.py -enc  25  4096  message.bin
$ ls -l message.bin
-rw-r--r-- 1 bet bet 512 Jun 17 21:31 message.bin
```

Step 2: Encrypt the natural number with openssl

To get a result of the encryption without padding, we need the option `-raw`. The result is saved to the file "message.bin.enc".

```
$ openssl pkeyutl -encrypt -raw -inkey publickey.pem -pubin -in message.bin -out message.bin.enc
```

Step 3: Transform the encrypted number into a natural number

The Python script `int2bin.py` converts the result of `openssl` back to a natural number `c`:

```
$ ./int2bin.py -dec message.bin.enc
252491 ... 316805
```

Step 4: Decrypt the encrypted number again with openssl

The result of `openssl` is the decrypted number. It is saved to the file "message.bin.enc.dec".

```
$ openssl pkeyutl -decrypt -raw -inkey privatekey.pem -in message.bin.enc -out message.bin.enc.dec
$ ls -l message.bin.enc.dec
-rw-r--r-- 1 bet bet 512 Jun 15 23:42 message.bin.enc.dec
```

Step 5: Convert the decrypted number into a natural number

The Python script `int2bin.py` converts the result of `openssl` back into an integer `m`:

```
$ ./int2bin.py -dec message.bin.enc.dec
25
```

Step 6: Follow the whole process with the Python function pow()

Using `openssl` and the Python script `convert-pem-dec.py` we extract the RSA parameters n, e and d from the private key.

```
$ openssl pkey -in privatekey.pem -noout -text > privatekey.txt
$ ./convert-pem-dec.py privatekey.txt
n = 793291 ... 187789
e = 65537
d = 214891 ... 510429
```

We could calculate the powers like usual in Python:

```
>>> m=25
>>> c = m**e % n
>>> m = c**d  % n
```

However, it's important to use instead the `pow()` function in order to avoid the decryption operation taking too long or just stopping without result.[26]

```
>>> c = pow(m, e, n)
>>> m = pow(c, d, n)
```

Comment

All commands of this Subsection 1.9 are contained in the shell script OpenSSL example 1.3. You can find this script here: https://www.cryptool.org/en/ctbook/openssl.

OpenSSL Example 1.3: Textbook RSA with `openssl` and Python script (commented in detail in `zahl-direkt-openssl-enc-dec.sh`)

```
NUMBER=25
KEYSIZE=4096
openssl pkey -in publickey.pem -pubin -noout -text
./int2bin.py -enc  $NUMBER   $KEYSIZE  message.bin
openssl pkeyutl -encrypt -raw -inkey publickey.pem -pubin -in message.bin -out message.bin.enc
./int2bin.py -dec message.bin.enc
openssl pkeyutl -decrypt -raw -inkey privatekey.pem -in message.bin.enc -out message.bin.enc.dec
./int2bin.py -dec message.bin.enc.dec
openssl pkey -in privatekey.pem -noout -text > privatekey.txt
./convert-pem-dec.py privatekey.txt
```

1.10 Generate random numbers

Here are some commands for generating random data with different length and format:

1) Generate 32 B = 256 bit and write them to a file called testkey.bin:

```
openssl rand -out testkey.bin 32
```

2) Generate 100 byte of random data in hexadecimal and redirect output to `file.hex`:

```
openssl rand -hex 100 > file.hex
```

3) Generate 100 byte of random data in base64 and redirect output to `file.b64`:

```
$ openssl rand -base64 100 > file.b64
$ more file.b64
32zjqFBi1jl3IyHkXoZ2PMLb7kWEO4Qqsqc9pja3zb1xQEUjYOaWSG5c9ZpBD7UE
18K0M4ZyVBKxmN1ywCjjMIxqRqoFmpV87GvRshfCxhX7s5DyCSX1Vznr00WjTEMS
Gkz9kg==
```

[26] a) SageMath expects ^ instead of ** as power operator.
b) SageMath informs when decrypting without `pow()`:
`exponent must be at most 9223372036854775807`
c) Decrypting can only be done fast if you choose a finite integer ring from the start, which can also be done with `pow()`: The result appears immediately for both Python and SageMath: 25 (in Python the output is `25L`)
d) `pow()` is so fast, that for a 4096-bit n there is no need at the command line to use CRT. See https://en.wikipedia.org/wiki/RSA_(cryptosystem): "Practical implementations use the Chinese remainder theorem to speed up the calculation using modulus of factors (mod pq using mod p and mod q)."

The file sizes (in byte) are different:

```
$ ls -l testkey.bin file.hex file.b64
-rw-r--r-- 1 bet bet  32 Jun  8 19:41 testkey.bin
-rw-r--r-- 1 bet bet 201 Jun  2 04:45 file.hex
-rw-r--r-- 1 bet bet 139 Jun  2 04:45 file.b64
```

1.11 Generate prime numbers with OpenSSL

OpenSSL is used productively to generate large prime numbers. For a prime of length 4096 bit a modern PC needs less than a second. [27]

```
$ openssl prime -generate -bits 4096
```

It is also possible to generate smaller primes, but the random number generator of OpenSSL seems to not cover all primes of a specific given bit length (for 2, 3 and 4 bit we always get the same prime).[28]

```
$ openssl prime -generate -bits 2
3
$ openssl prime -generate -bits 3
7
$ openssl prime -generate -bits 4
13
$ openssl prime -generate -bits 5
29 or 31
$ openssl prime -generate -bits 6
53, 59, 61
$ openssl prime -generate -bits 7
127, 113, 103, 101, 107, 109, 97
$ openssl prime -generate -bits 8
211, 227, 239, 223, 251, ...
```

In addition to generate pseudo random primes, OpenSSL also can check primality. The output shows the input first in hex, then decimal:

```
$ openssl prime 17
11 (17) is prime
$ openssl prime 717
2CD (717) is not prime
```

OpenSSL Example 1.4: Generate random numbers and primes (also see 1.2 `hybrid-openssl-enc-dec.sh`)

```
openssl rand -out testkey.bin 32
openssl rand -out file.bin 100
openssl rand -hex 100 > file.hex
openssl rand -base64 100 > file.b64
openssl prime -generate -bits 4096
openssl prime 17
```

[27] If one needs a hex output instead of decimal, this can be done by appending the option `-hex`:
`openssl prime -generate -hex -bits 200`
With the option `-safe` you also can generate so-called "safe" or "strong" prime numbers. These are not necessarily needed for RSA, but for the Diffie-Hellman key exchange.
The list of all options can be found in the original documentation: https://www.openssl.org/docs/manmaster/man1/openssl-prime.html

[28] We don't know why the primes 5, 11, 17, 19, 23, 37, 41, 43, 47, 67, 71, 73, 79, 83, 89, 131, 137, 139, 149, 151, 157, 163, 167, 173, 179, 181, 191, 193, 197, 199, ... never appeared with OpenSSL 1.1.1.

1.12 Comparing cipher speeds with openssl

With OpenSSL you can test and compare the speed of different ciphers on your PC.

Test a single cipher[29]

```
$ openssl speed -evp aes-256-gcm
Doing aes-256-gcm for 3s on 16 size blocks: 121185279 aes-256-gcm's in 3.00s
Doing aes-256-gcm for 3s on 64 size blocks: 81302940 aes-256-gcm's in 3.00s
...
$ openssl speed aes-256-ige
...
$ openssl speed aes-256-cbc
...
$ openssl speed md4
Doing md4 for 3s on 16 size blocks: 24896959 md4's in 3.00s
Doing md4 for 3s on 64 size blocks: 18766727 md4's in 3.00s
Doing md4 for 3s on 256 size blocks: 10853787 md4's in 3.00s
...
```

This means that OpenSSL runs the corresponding algorithm in a loop for 3 seconds with a 16 B resp. 64 B etc. input. E. g. for the hash algorithm `md4` in the case 16 B more than 24 million iterations were run in 3 s. At the end of the console output (it's left out above because it's quite long) a list is shown containing the number of processed bytes, e. g. more than 384 bytes in 3 s for `md4` and 16 B, that is over 128 million bytes per second.

Test all supported ciphers

The test of all functions took about 20 minutes on a modern PC. This can be done by calling `openssl speed`.

```
$ openssl speed
Doing md4 for 3s on 16 size blocks: 24484712 md4's in 3.00s
...
Doing md5 for 3s on 16 size blocks: 33106733 md5's in 3.00s
...
Doing hmac(md5) for 3s on 16 size blocks: 14207977 hmac(md5)'s in 3.00s
...
Doing sha1 for 3s on 16 size blocks: 34526550 sha1's in 3.00s
...
Doing sha256 for 3s on 16 size blocks: 20655948 sha256's in 3.00s
Doing sha256 for 3s on 64 size blocks: 11635481 sha256's in 3.00s
Doing sha256 for 3s on 256 size blocks: 5362737 sha256's in 3.00s
Doing sha256 for 3s on 1024 size blocks: 1660351 sha256's in 3.00s
Doing sha256 for 3s on 8192 size blocks: 225299 sha256's in 3.00s
Doing sha256 for 3s on 16384 size blocks: 113748 sha256's in 3.00s
...
Doing sha512 for 3s on 16 size blocks: 14328104 sha512's in 3.00s
...
Doing whirlpool for 3s on 16 size blocks: 9988996 whirlpool's in 3.00s
...
Doing rmd160 for 3s on 16 size blocks: 11843755 rmd160's in 3.00s
...
Doing rc4 for 3s on 16 size blocks: 170249167 rc4's in 3.00s
...
Doing des cbc for 3s on 16 size blocks: 19706910 des cbc's in 3.00s
...
Doing des ede3 for 3s on 16 size blocks: 7503733 des ede3's in 3.00s
...
Doing aes-128 cbc for 3s on 16 size blocks: 35527030 aes-128 cbc's in 3.00s
...
Doing aes-192 cbc for 3s on 16 size blocks: 30236871 aes-192 cbc's in 3.00s
...
Doing aes-256 cbc for 3s on 16 size blocks: 26267630 aes-256 cbc's in 3.00s
...
Doing aes-128 ige for 3s on 16 size blocks: 35425097 aes-128 ige's in 3.00s
...
Doing aes-192 ige for 3s on 16 size blocks: 30067762 aes-192 ige's in 3.00s
...
Doing aes-256 ige for 3s on 16 size blocks: 26229118 aes-256 ige's in 3.00s
...
Doing ghash for 3s on 16 size blocks: 362653379 ghash's in 3.00s
...
Doing camellia-128 cbc for 3s on 16 size blocks: 29850364 camellia-128 cbc's in 3.00s
...
Doing camellia-192 cbc for 3s on 16 size blocks: 25589359 camellia-192 cbc's in 3.00s
...
Doing camellia-256 cbc for 3s on 16 size blocks: 25915761 camellia-256 cbc's in 3.00s
...
Doing seed cbc for 3s on 16 size blocks: 24155793 seed cbc's in 3.00s
...
Doing rc2 cbc for 3s on 16 size blocks: 13304928 rc2 cbc's in 3.00s
```

[29] The block mode GCM of AES (here e. g. `aes-256-gcm`) is only accessible via the EVP-name (EVP = "Envelope" = the Digital EnVeloPe library, which is a high level interface to OpenSSL cryptographic functions for programmers).

```
...
Doing blowfish cbc for 3s on 16 size blocks: 34147133 blowfish cbc's in 3.00s
...
Doing cast cbc for 3s on 16 size blocks: 30905984 cast cbc's in 3.00s
...
Doing rand for 3s on 16 size blocks: 3144269 rand's in 2.93s
...
Doing 512 bits private rsa's for 10s: 323981 512 bits private RSA's in 10.00s
Doing 512 bits public rsa's for 10s: 5723802 512 bits public RSA's in 10.00s
...
Doing 512 bits sign dsa's for 10s: 219602 512 bits DSA signs in 9.99s
Doing 512 bits verify dsa's for 10s: 417515 512 bits DSA verify in 10.00s
...
Doing 160 bits sign ecdsa's for 10s: 65977 160 bits ECDSA signs in 9.99s
Doing 160 bits verify ecdsa's for 10s: 73865 160 bits ECDSA verify in 10.00s
..
Doing 512 bits sign ecdsa's for 10s: 9819 512 bits ECDSA signs in 10.00s
Doing 512 bits verify ecdsa's for 10s: 13214 512 bits ECDSA verify in 10.00s
...
Doing 160 bits  ecdh's for 10s: 69874 160-bits ECDH ops in 9.99s
..
Doing 448 bits  ecdh's for 10s: 24944 448-bits ECDH ops in 10.00s
...
Doing 253 bits sign Ed25519's for 10s: 325591 253 bits Ed25519 signs in 10.00s
Doing 253 bits verify Ed25519's for 10s: 119356 253 bits Ed25519 verify in 10.00s
Doing 456 bits sign Ed448's for 10s: 28164 456 bits Ed448 signs in 10.00s
Doing 456 bits verify Ed448's for 10s: 22268 456 bits Ed448 verify in 10.00s
...
OpenSSL 1.1.1  11 Sep 2018
built on: Wed May 27 19:15:54 2020 UTC
options:bn(64,64) rc4(16x,int) des(int) aes(partial) blowfish(ptr)
compiler: gcc -fPIC -pthread -m64 -Wa,--noexecstack -Wall -Wa,--noexecstack -g -O2
-fdebug-prefix-map=/build/openssl-dyPhHZ/openssl-1.1.1=. -fstack-protector-strong
-Wformat -Werror=format-security -DOPENSSL_USE_NODELETE -DL_ENDIAN
-DOPENSSL_PIC -DOPENSSL_CPUID_OBJ -DOPENSSL_IA32_SSE2
-DOPENSSL_BN_ASM_MONT -DOPENSSL_BN_ASM_MONT5
-DOPENSSL_BN_ASM_GF2m -DSHA1_ASM -DSHA256_ASM -DSHA512_ASM
-DKECCAK1600_ASM -DRC4_ASM -DMD5_ASM -DAES_ASM -DVPAES_ASM
-DBSAES_ASM -DGHASH_ASM -DECP_NISTZ256_ASM -DX25519_ASM
-DPADLOCK_ASM -DPOLY1305_ASM -DNDEBUG -Wdate-time -D_FORTIFY_SOURCE=2
...
The 'numbers' are in 1000s of bytes per second processed.

type             16 bytes     64 bytes    256 bytes   1024 bytes   8192 bytes  16384 bytes
md2                 0.00         0.00         0.00         0.00         0.00         0.00
mdc2                0.00         0.00         0.00         0.00         0.00         0.00
md4            130585.13k   398696.92k   910739.20k  1345290.92k  1548752.21k  1572099.41k
md5            176569.24k   406777.28k   713249.45k   878841.51k   940930.39k   946765.82k
hmac(md5)       75775.88k   231158.85k   535103.06k   767268.18k   912558.76k   936700.59k
sha1           184141.60k   452444.01k   906871.81k  1217873.92k  1349842.26k  1359686.31k
rmd160          63166.69k   151465.28k   273872.98k   339738.97k   373060.95k   378328.41k
rc4            907995.56k   958468.46k   768643.33k   730315.78k   721253.72k   717357.06k
des cbc        105103.52k   107784.15k   107826.26k   108655.62k   108079.79k   108030.63k
des ede3        40019.91k    40663.00k    40784.30k    40932.69k    40970.92k    40932.69k
idea cbc            0.00         0.00         0.00         0.00         0.00         0.00
seed cbc       128830.90k   129395.80k   129792.34k   129607.68k   129791.32k   129531.90k
rc2 cbc         70959.62k    72832.68k    73396.91k    73166.85k    73430.36k    73362.09k
rc5-32/12 cbc       0.00         0.00         0.00         0.00         0.00         0.00
blowfish cbc   182118.04k   196655.51k   199951.19k   201562.11k   201695.23k   202265.94k
cast cbc       164831.91k   176574.40k   179900.84k   180461.23k   180718.25k   181114.20k
aes-128 cbc    189477.49k   211289.05k   216948.14k   219875.33k   219941.55k   219703.98k
aes-192 cbc    161263.31k   175736.53k   181192.87k   182700.71k   184015.88k   182867.29k
aes-256 cbc    140094.03k   152293.46k   154599.94k   156222.46k   156969.64k   156652.89k
camellia-128 cbc   159201.94k   239704.73k   272104.62k   279830.19k   283238.40k   281204.05k
camellia-192 cbc   136476.58k   188461.03k   205451.26k   211486.38k   211664.90k   212516.86k
camellia-256 cbc   138217.39k   186853.99k   205037.65k   211876.86k   214119.77k   214553.94k
sha256         110165.06k   248223.59k   457620.22k   566733.14k   615216.47k   621215.74k
sha512          76416.55k   303715.82k   518181.63k   774714.71k   918227.63k   931921.92k
whirlpool       53274.65k   114055.74k   190728.70k   230286.68k   245000.87k   246338.90k
aes-128 ige    188933.85k   201223.36k   202676.57k   205343.40k   205739.35k   206067.03k
aes-192 ige    160361.40k   168854.02k   171212.89k   172596.57k   173197.99k   172900.35k
aes-256 ige    139888.63k   146539.63k   148333.31k   149011.11k   148886.87k   148799.49k
ghash         1934151.35k  7725503.27k 11779075.84k 12840266.07k 13200414.04k 13222084.61k
rand            17170.07k    65357.57k   215719.23k   503646.39k   850989.89k   888807.34k

                  sign    verify    sign/s verify/s
rsa  512 bits 0.000031s 0.000002s  32398.1 572380.2
rsa 1024 bits 0.000067s 0.000004s  14992.9 240760.0
rsa 2048 bits 0.000426s 0.000013s   2344.9  78532.9
rsa 3072 bits 0.001293s 0.000026s    773.5  38368.9
rsa 4096 bits 0.002889s 0.000045s    346.1  22114.9
rsa 7680 bits 0.026368s 0.000149s     37.9   6704.6
rsa 15360 bits 0.134533s 0.000579s      7.4   1728.4

                  sign    verify    sign/s verify/s
dsa  512 bits 0.000045s 0.000024s  21982.2  41751.5
dsa 1024 bits 0.000075s 0.000053s  13330.1  18825.3
dsa 2048 bits 0.000185s 0.000158s   5410.0   6345.1

                             sign    verify    sign/s verify/s
 160 bits ecdsa (secp160r1)   0.0002s   0.0001s   6604.3   7386.5
 192 bits ecdsa (nistp192)   0.0002s   0.0002s   5374.1   5928.7
 224 bits ecdsa (nistp224)   0.0000s   0.0001s  23458.6  10054.1
 256 bits ecdsa (nistp256)   0.0000s   0.0000s  62271.3  20989.1
 384 bits ecdsa (nistp384)   0.0007s   0.0005s   1407.2   1923.6
 521 bits ecdsa (nistp521)   0.0002s   0.0004s   4625.7   2388.0
 163 bits ecdsa (nistk163)   0.0002s   0.0003s   6041.7   3051.9
 233 bits ecdsa (nistk233)   0.0002s   0.0005s   4388.5   2215.0
 283 bits ecdsa (nistk283)   0.0004s   0.0008s   2567.7   1304.6
 409 bits ecdsa (nistk409)   0.0007s   0.0013s   1528.8    778.3
 571 bits ecdsa (nistk571)   0.0015s   0.0029s    676.6    348.6
 163 bits ecdsa (nistb163)   0.0002s   0.0003s   5757.6   2910.1
 233 bits ecdsa (nistb233)   0.0002s   0.0005s   4222.7   2148.0
```

```
283 bits ecdsa (nistb283)   0.0004s   0.0008s   2452.6   1248.4
409 bits ecdsa (nistb409)   0.0007s   0.0013s   1456.3    746.6
571 bits ecdsa (nistb571)   0.0016s   0.0031s    631.0    324.5
256 bits ecdsa (brainpoolP256r1)   0.0003s   0.0003s   3332.4   3747.2
256 bits ecdsa (brainpoolP256t1)   0.0003s   0.0003s   3339.5   3876.2
384 bits ecdsa (brainpoolP384r1)   0.0007s   0.0006s   1388.1   1775.2
384 bits ecdsa (brainpoolP384t1)   0.0007s   0.0005s   1406.3   1908.6
512 bits ecdsa (brainpoolP512r1)   0.0010s   0.0008s    965.5   1243.1
512 bits ecdsa (brainpoolP512t1)   0.0010s   0.0008s    981.9   1321.4

                              op      op/s
160 bits ecdh (secp160r1)   0.0001s   6994.4
192 bits ecdh (nistp192)   0.0002s   5659.3
224 bits ecdh (nistp224)   0.0001s  14945.8
256 bits ecdh (nistp256)   0.0000s  27865.6
384 bits ecdh (nistp384)   0.0007s   1449.0
521 bits ecdh (nistp521)   0.0003s   3882.4
163 bits ecdh (nistk163)   0.0002s   6230.6
233 bits ecdh (nistk233)   0.0002s   4584.6
283 bits ecdh (nistk283)   0.0004s   2691.5
409 bits ecdh (nistk409)   0.0006s   1626.3
571 bits ecdh (nistk571)   0.0014s    716.0
163 bits ecdh (nistb163)   0.0002s   5995.4
233 bits ecdh (nistb233)   0.0002s   4445.4
283 bits ecdh (nistb283)   0.0004s   2564.5
409 bits ecdh (nistb409)   0.0007s   1532.7
571 bits ecdh (nistb571)   0.0015s    659.9
256 bits ecdh (brainpoolP256r1)   0.0003s   3513.3
256 bits ecdh (brainpoolP256t1)   0.0003s   3501.7
384 bits ecdh (brainpoolP384r1)   0.0007s   1460.7
384 bits ecdh (brainpoolP384t1)   0.0007s   1478.0
512 bits ecdh (brainpoolP512r1)   0.0010s    998.7
512 bits ecdh (brainpoolP512t1)   0.0010s   1022.6
253 bits ecdh (X25519)   0.0000s  40491.7
448 bits ecdh (X448)   0.0004s   2494.4

                              sign    verify    sign/s verify/s
253 bits EdDSA (Ed25519)   0.0000s   0.0001s  32559.1  11935.6
456 bits EdDSA (Ed448)   0.0004s   0.0004s   2816.4   2226.8
```

1.13 Retrieve and evaluate certificates

Here we list some single commands that allow to get protocol details of connected servers.

Parameters of Google web-server certificates

The following command uses the argument `s_client` to open a secure connection to `www.google.com` and to print details about this connection[30]: The port number 443 is the standard one used by web servers for receiving HTTPS rather than HTTP connections (for HTTP, the standard port is 80). If the attempted connection succeeds, the three digital certificates from Google are displayed together with information about the secure session, the cipher suite in play, and related items. If you rerun the command now, you will get the current Google certificates, which will "slightly" differ from the ones shown here.

```
$ openssl s_client -connect www.google.com:443 -showcerts
CONNECTED(00000005)
depth=2 OU = GlobalSign Root CA - R2, O = GlobalSign, CN = GlobalSign
verify return:1
depth=1 C = US, O = Google Trust Services, CN = GTS CA 1O1
verify return:1
depth=0 C = US, ST = California, L = Mountain View, O = Google LLC, CN = www.google.com
verify return:1
---
Certificate chain
 0 s:C = US, ST = California, L = Mountain View, O = Google LLC, CN = www.google.com
   i:C = US, O = Google Trust Services, CN = GTS CA 1O1
-----BEGIN CERTIFICATE-----
MIIExzCCA6+gAwIBAgIRALJsaMAobZ6SCAAAAABDVSUwDQYJKoZIhvcNAQELBQAw
QjELMAkGA1UEBhMCVVMxHjAcBgNVBAoTFUdvb2dsZSBUcnVzdCBTZXJ2aWNlczET
MBEGA1UEAxMKR1RTIENBIDFPMTAeFw0yMDA1MjYxNTMwMDNaFw0yMDA4MTgxNTMw
MDNaMGgxCzAJBgNVBAYTAlVTMRMwEQYDVQQIEwpDYWxpZm9ybmlhMRYwFAYDVQQH
Ew1Nb3VudGFpbiBWaWV3MRMwEQYDVQQKEwpHb29nbGUgTExDMRcwFQYDVQQDEw53
d3cuZ29vZ2xlLmNvbTBZMBMGByqGSM49AgEGCCqGSM49AwEHA0IABPpZHiiNrjXN
5KoWOrWQjIHzVW5hW0yPa4NH9O6oqjUZPidUIQZmR9Or5+Yx5dLJDzBuEhCoxw6i
9elNnOnwl7ujggJbMIICVzAOBgNVHQ8BAf8EBAMCB4AwEwYDVR0lBAwwCgYIKwYB
BQUHAwEwDAYDVR0TAQH/BAIwADAdBgNVHQ4EFgQUvYFt35OUFFMLkjkidJ8zmSL4
8RUwHwYDVR0jBBgwFoAUmNH4bhDrz5vsYJ8YkBug630J/SswaAYIKwYBBQUHAQEE
XDBaMCsGCCsGAQUFBzABhh9odHRwOi8vb2NzcC5wa2kuZ29vZy9ndHMxbzFjb3Jl
MCsGCCsGAQUFBzAChh9odHRwOi8vcGtpLmdvb2cvZ3NyMi9HVFMxTzEuY3J0MBkG
A1UdEQQSMBCCDnd3dy5nb29nbGUuY29tMCEGA1UdIAQaMBgwCAYGZ4EMAQICMAwG
CisGAQQB1nkCBQMwMwYDVR0fBCwwKjAooCagJIYiaHR0cDovL2NybC5wa2kuZ29v
Zy9HVFMxTzFjb3JlLmNybDCCAQMGCisGAQQB1nkCBAIEgfQEgfEA7wB1AAe3XBvl
fWj/8bDGHSMVx7rmV3xXlLdq7rxhOhpp06IcAAABclHRz04AAAQDAEYwRAIgWz74
kj8TOayaBwRBQs3Ot1xAomxpiqmpq1WV7RqkujECIAax+O9FRc5EMUwOyt/ZCVuj
EqrAq1iwBkmh4+Ht2SmoAHYA5xLysDd+GmL7jskMYYTx6ns3y1YdESZb8+DzS/JB
VG4AAAFyUdHNBQAABAMARzBFAiEA58Y8iumnVpkgGAbJOYOqAwAYmHUn739SBFx0
g8Cwis0CIGyha0ERmcA4Z8exj3GGgqy5c/0OQOmIuFGxtb8TraTZMA0GCSqGSIb3
DQEBCwUAA4IBAQAEZanMYZDcVkdppa6PRuLXMRKSL2J9eGjw4wBNIQqyN0wvxTVx
```

[30] see https://opensource.com/article/19/6/cryptography-basics-openssl-part-1

```
QvQv4M7Horc4zCsinfM6jYUy5DpYZWec770+SbOSK7wkRsW0BiJZ5DyxDMSZV4Wc
eQvA939Ys9pw+0FkoZZ/pvITUXsPWbywcwHWUoiR56Wtnj51I8VaS06Z45GflkAl
R9VIqdXX6EOlT+s6vtLVD65bth5ZYnpLgLfzNixEV4GcRXFvy9G5RHCeQKJMbyJx
kgmPH0eGL0pvB5IWFmUI7Ny4WkAobnsDNpyeuuxhDHQXYbIkue6tnW2Pb0khdsQ+
mKgE9Xo0YKd5jsqtsD421wnxeVnMKYLICw7c
-----END CERTIFICATE-----
 1 s:C = US, O = Google Trust Services, CN = GTS CA 1O1
   i:OU = GlobalSign Root CA - R2, O = GlobalSign, CN = GlobalSign
-----BEGIN CERTIFICATE-----
MIIESjCCAzKgAwIBAgINAeO0mqGNiqmBJWlQuDANBgkqhkiG9w0BAQsFADBMMSAw
HgYDVQQLExdHbG9iYWxTaWduIFJvb3QgQ0EgLSBSMjETMBEGA1UEChMKR2xvYmFs
U2lnbjETMBEGA1UEAxMKR2xvYmFsU2lnbjAeFw0xNzA2MTUwMDAwNDJaFw0yMTEy
MTUwMDAwNDJaMEIxCzAJBgNVBAYTAlVTMR4wHAYDVQQKExVHb29nbGUgVHJ1c3Qg
U2VydmljZXMxEzARBgNVBAMTCkdUUyBDQSAxTzEwggEiMA0GCSqGSIb3DQEBAQUA
A4IBDwAwggEKAoIBAQDQGM9F1IvN05zkQO9+tN1pIRvJzzyOTHW5DzEZhD2ePCnv
UA0Qk28FgICfKqC9EksC4T2fWBYk/jCfC3R3VZMdS/dN4ZKCEPZRrAzDsiKUDzRr
mBBJ5wudgzndIMYcLe/RGGFl5yODIKgjEv/SJH/UL+dEaltN11BmsK+eQmMF++Ac
xGNhr59qM/9il71I2dN8FGfcddwuaej4bXhp0LcQBbjxMcI7JP0aM3T4I+DsaxmK
FsbjzaTNC9uzpFlgOIg7rR25xoynUxv8vNmkq7zdPGHXkxWY7oG9j+JkRyBABk7X
rJfoucBZEqFJJSPk7XA0LKW0Y3z5oz2D0c1tJKwHAgMBAAGjggEzMIIBLzAOBgNV
HQ8BAf8EBAMCAYYwHQYDVR0lBBYwFAYIKwYBBQUHAwEGCCsGAQUFBwMCMBIGA1Ud
EwEB/wQIMAYBAf8CAQAwHQYDVR0OBBYEFJjR+G4Q68+b7GCfGJAboOt9Cf0rMB8G
A1UdIwQYMBaAFJviB1dnHB7AagbeWbSaLd/cGYYuMDUGCCsGAQUFBwEBBCkwJzAl
BggrBgEFBQcwAYYZaHR0cDovL29jc3AucGtpLmdvb2cvZ3NyMjAyBgNVHR8EKzAp
MCegJaAjhiFodHRwOi8vY3JsLnBraS5nb29nL2dzcjIvZ3NyMi5jcmwwPwYDVR0g
BDgwNjA0BgZngQwBAgIwKjAoBggrBgEFBQcCARYcaHR0cHM6Ly9wa2kuZ29vZy9y
ZXBvc2l0b3J5LzANBgkqhkiG9w0BAQsFAAOCAQEAGoA+Nnn78y6pRjd9XlQWNa7H
TgiZ/r3RNGkmUmYHPQq6Scti9PEajvwRT2iWTHQr02fesqOqBY2ETUwgZQ+lltoN
FvhsO9tvBCOIazpswWC9aJ9xju4tWDQH8NVU6YZZ/XteDSGU9YzJqPjY8q3MDxrz
mqepBCf5o8mw/wJ4a2G6xzUr6Fb6T8McDO22PLRL6u3M4Tzs3A2M1j6bykJYi8wW
IRdAvKLWZu/axBVbzYmqmwkm5zLSDW5nIAJbELCQCZwMH56t2Dvqofxs6BBcCFIZ
USpxu6x6td0V7SvJCCosirSmIatj/9dSSVDQibet8q/7UK4v4ZUN80atnZz1yg==
-----END CERTIFICATE-----
---
Server certificate
subject=C = US, ST = California, L = Mountain View, O = Google LLC, CN = www.google.com

issuer=C = US, O = Google Trust Services, CN = GTS CA 1O1

---
No client certificate CA names sent
Peer signing digest: SHA256
Peer signature type: ECDSA
Server Temp Key: X25519, 253 bits
---
SSL handshake has read 2638 bytes and written 396 bytes
Verification: OK
---
New, TLSv1.3, Cipher is TLS_AES_256_GCM_SHA384
Server public key is 256 bit
Secure Renegotiation IS NOT supported
Compression: NONE
Expansion: NONE
No ALPN negotiated
Early data was not sent
Verify return code: 0 (ok)
---
read:errno=0
```

A good description of this output can be found e. g. on the website of Marty Kalin, where we also found the example from above: `https://opensource.com/article/19/6/cryptography-basics-openssl-part-1`

With the following command we get the chain of certificates and the two non-root certificates in base64 format (enter `Enter` at the end of the first line): depth = 2,1,0

```
$ openssl s_client -showcerts -connect www.google.com:443 | sed -ne '/-BEGIN CERTIFICATE-/,/-END CERTIFICATE-/p' ▸
 ▸> google_server_certs.crt
depth=2 OU = GlobalSign Root CA - R2, O = GlobalSign, CN = GlobalSign
verify return:1
depth=1 C = US, O = Google Trust Services, CN = GTS CA 1O1
verify return:1
depth=0 C = US, ST = California, L = Mountain View, O = Google LLC, CN = www.google.com
verify return:1
read:errno=0
```

It is also possible to download the server certificate, cut all not needed information and save the file in PEM format. There seems to be no option to get the algorithm information. With the option `-dates` or `-enddate` you can get the certificate expiry date from the created pem file. You can see that the Google certificate has a validity of about three months.

```
$ openssl s_client -showcerts -connect www.google.com:443 </dev/null 2>/dev/null | openssl x509 -outform PEM >▸
 ▸google_server_certs.pem
$ openssl x509 -in google_server_certs.pem -pubkey -noout
-----BEGIN PUBLIC KEY-----
MFkwEwYHKoZIzj0CAQYIKoZIzj0DAQcDQgAE+lkeKI2uNc3kqhY6tZCMgfNVbmFb
TI9rg0f07qiqNRk+J1QhBmZH06vn5jHl0skPMG4SEKjHDqL16U2c6fCXuw==
```

```
-----END PUBLIC KEY-----

$ openssl x509 -in google_server_certs.pem -noout -dates
notBefore=May 26 15:30:03 2020 GMT
notAfter=Aug 18 15:30:03 2020 GMT

$ openssl x509 -in google_server_certs.pem -noout -enddate
notAfter=Aug 18 15:30:03 2020 GMT
```

A good description of e. g. how you can check the supported cipher suites of a server you can find here: https://www.feistyduck.com/library/openssl-cookbook/online/testing-with-openssl/index.html

Automatically extract the single certificates out of an organization's certificate in order to get the modulus of each single certificate

Using the command below one could show the modulus of a certificate in case RSA has been used for the signature. The certificate of Google from 05/26/2020 (see above) shows "Public Key Algorithm: id-ecPublicKey" instead of "Public Key Algorithm: rsaEncryption". So the option `-modulus` leads to an error:

```
$ openssl x509 -in google_server_certs.crt -text -pubkey -noout -modulus
Modulus=Wrong Algorithm type
```

Now we have a closer look at the certificate of "Let's Encrypt" which is using RSA instead of ECDSA. We fetch the certificate with `openssl s_client`, then we extract the contained single certificates from it (in base64 format) and write every certificate into a separate file using `awk` (`awk` searches lines beginning with the actual value of the `awk` outer variable `RS`. In case it finds such a line it writes this and all following lines until the next occurrence of the search string into a file with the name of the inner variable `FileName`). Finally it tells us the number of occurrences of `BEGIN CERT ... END CERT` constructs that had been found.[31]

```
$ echo | openssl s_client -showcerts -connect www.letsencrypt.org:443 > all.crt

$ cat all.crt | sed -ne '/-BEGIN CERTIFICATE-/,/-END CERTIFICATE-/p'
  > all_base64-only.crt

$ cat all_base64-only.crt | awk -v RS="-----BEGIN CERT" 'NR>1  {++cnt;
  FileName="Cert_"(NR-1)".crt"; print RS $0 > FileName; print "Created "
  FileName} END {print "Number of certs found in base64 format = " cnt}'
Created Cert_1.crt
Created Cert_2.crt
Number of certs found in base64 format = 2

$ ls -l *.crt
-rw-r--r-- 1 bet bet 4395 Jun 25 00:27 all.crt
-rw-r--r-- 1 bet bet 1940 Jun 25 00:28 Cert_1.crt
-rw-r--r-- 1 bet bet 1648 Jun 25 00:28 Cert_2.crt
```

Since these Let's Encrypt certificates use RSA, you can output the module of the certificate owner (subject public key info) for each individual file using `openssl`.

```
openssl x509 -in Cert_1.crt -noout -modulus
Modulus=DEB5AF ... CD6F51
```

Do private and public keys match?

If you want to check whether the public key from a certificate and the public key from the file with the private key match, you can do this with the following commands:

```
$ openssl x509 -in ssl.crt -pubkey -noout > from_crt.pub
$ openssl rsa -in ssl.privkey -pubout > from_privkey.pub
$ diff from_crt.pub from_privkey.pub
```

[31] One also could execute those 3 commands consecutively (per pipe instead of using intermediary files). The first command used `openssl`; the processing of the results with `sed` and `awk` was done in the `bash` shell.

The same goes for ECDSA keys.[32]

Test the availability of the mail server of the university of Siegen and get the used algorithms

```
$ openssl s_client -connect mail.uni-siegen.de:993 -showcerts
bet@tux19:~/D/openssl$ openssl s_client -connect mail.uni-siegen.de:993 -showcerts
CONNECTED(00000005)
depth=3 C = DE, O = T-Systems Enterprise Services GmbH, OU = T-Systems Trust Center, CN = T-TeleSec GlobalRoot Class 2
verify return:1
depth=2 C = DE, O = Verein zur Foerderung eines Deutschen Forschungsnetzes e. V., OU = DFN-PKI, CN = DFN-Verein Certification Authority 2
verify return:1
depth=1 C = DE, O = Verein zur Foerderung eines Deutschen Forschungsnetzes e. V., OU = DFN-PKI, CN = DFN-Verein Global Issuing CA
verify return:1
depth=0 C = DE, ST = Nordrhein-Westfalen, L = Siegen, O = Universitaet Siegen, CN = mail.uni-siegen.de
verify return:1
---
Certificate chain
 0 s:C = DE, ST = Nordrhein-Westfalen, L = Siegen, O = Universitaet Siegen, CN = mail.uni-siegen.de
   i:C = DE, O = Verein zur Foerderung eines Deutschen Forschungsnetzes e. V., OU = DFN-PKI, CN = DFN-Verein Global Issuing CA
-----BEGIN CERTIFICATE-----
MIIIujCCB6KgAwIBAgIMIwfrEZso2bJu9A95MA0GCSqGSIb3DQEBCwUAMIGNMQsw
CQYDVQQGEwJERTFFMEMGA1UECgw8VmVyZWluIHp1ciBGb2VyZGVydW5nIGVpbmVz
IERldXRzY2hlbiBGb3JzY2h1bmdzbmV0emVzIGUuIFYuMRAwDgYDVQQLDAdERk4t
UEtJMSUwIwYDVQQDDBxERk4tVmVyZWluIEdsb2JhbCBJc3N1aW5nIENBMB4XDTIw
MDYxNjA2NTgxMloXDTIyMDkxODA2NTgxMlowdzELMAkGA1UEBhMCREUxHDAaBgNV
BAgME05vcmRyaGVpbi1XZXN0ZmFsZW4xDzANBgNVBAcMBlNpZWdlbjEcMBoGA1UE
CgwTVW5pdmVyc2l0YWV0IFNpZWdlbjEbMBkGA1UEAwwSbWFpbC51bmktc2llZ2Vu
LmRlMIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAowx01PYGm+fXc5/V
1YUc2U7Vnd2W1RM/l6thk+M174Ba8RI0Rd3Eiz4735xta+QaDtQIOG2J3o6bmRw3
Jw2WASuxLJameDF3bxJETuwHkq0uL2NSKOtDcQaf9M2BID38jmMFbfLV3eKf7L4c
ZcZ/MlZ+Ugmh171Qm+UUNliIGZQadO/7skYx1oHqvhW4B6PRjbVLjkA/F5gT+1u4
ORojID8rTY9c+8qdhG+C7BMX44kkWe3buKlkDJPIh67D5byepoKFH5f9x7dUuAKy
mxB9n4c+g6xQEZfLJkIzSOlCM4vRSVsU2ecLer7DsRw4Q5bqmOhtPoD+wcW76BMx
2oZOmQIDAQABo4IFLTCCBSkwVwYDVR0gBFAwTjAIBgZngQwBAgIwDQYLKwYBBAGB
rSGCLB4wDwYNKwYBBAGBrSGCLAEBBDAQBg4rBgEEAYGtIYIsAQEEBzAQBg4rBgEE
AYGtIYIsAgEEBzAJBgNVHRMEAjAAMA4GA1UdDwEB/wQEAwIFoDATBgNVHSUEDDAK
BggrBgEFBQcDATAdBgNVHQ4EFgQU0s2Ru8znB01n/Yhh/lEQOLhH0gIwHwYDVR0j
BBgwFoAUazqYi/nyU4na4K2yMh4JH+iqO3QwgfQGA1UdEQSB7DCB6YIXYXV0aC5t
YWlsLnVuaS1zaWVnZW4uZGWCF2RpbmtlbC5hZC51bmktc2llZ2VuLmRlghdnZXJz
dGUuYWQudW5pLXNpZWdlbi5kZYISaW1hcC51bmktc2llZ2VuLmRlghJtYWlsLnVu
aS1zaWVnZW4uZGWCFW91dGxvb2sudW5pLXNpZWdlbi5kZYIXb3V0bG9va2NzLnVu
aS1zaWVnZW4uZGWCF3JvZ2dlbi5hZC51bmktc2llZ2VuLmRlghJzbXRwLnVuaS1z
aWVnZW4uZGWCF3dlaXplbi5hZC51bmktc2llZ2VuLmRlMIGNBgNVHR8EgYUwgYIw
P6A9oDuGOWh0dHA6Ly9jZHAxLnBjYS5kZm4uZGUvZGZuLWNhLWdsb2JhbC1nMi9w
dWIvY3JsL2NhY3JsLmNybDA/oD2gO4Y5aHR0cDovL2NkcDIucGNhLmRmbi5kZS9k
Zm4tY2EtZ2xvYmFsLWcyL3B1Yi9jcmwvY2FjcmwuY3JsMIHbBggrBgEFBQcBAQSB
zjCByzAzBggrBgEFBQcwAYYnaHR0cDovL29jc3AucGNhLmRmbi5kZS9PQ1NQLVNl
cnZlci9PQ1NQMEkGCCsGAQUFBzAChj1odHRwOi8vY2RwMS5wY2EuZGZuLmRlL2Rm
biljYS1nbG9iYWwtZzIvcHViL2NhY2VydC9jYWNlcnQuY3J0MEkGCCsGAQUFBzAC
hjlodHRwOi8vY2RwMi5wY2EuZGZuLmRlL2Rmbi1jYS1nbG9iYWwtZzIvcHViL2Nh
Y2VydC9jYWNlcnQuY3J0MIIB9wYKKwYBBAHWeQIEAgSCAecEggHjAeEAdwBGpVXr
dfqRIDC1oolp9PN9ESxBdL79SbiFq/L8cP5tRwAAAXK769yOAAAEAwBIMEYCIQDn
6UkxQRveDHHyTxxdvkZNnTkx36+fccP0mdS0zpYhrgIhAOgKdcGNve/UpSPg09/t
Hhv3/Ur5TQCW0EcLIArRbV2qAHUAKXm+8J45OSHwVnOfY6V35b5XfZxgCvj5TV0m
XCVdx4QAAAFyu+vgcQAABAMARjBEAiAtPCEJdS40/4xN5s2z2t94aLGNJAFIb2zq
RCbIrzUaDAIgb5xcvzut8aOLGs9FaS9rUe+DFzoQgUkpc5yh3HVeTgwAdwBvU3as
MfAxGdiZAKRRFf93FRwR2QLBACkGjbIImjfZEwAAAXK76912AAAEAwBIMEYCIQCh
zyqPMhMoLK3k/BiweeRm8Sj+DybAEgN5Zm7tZJko2AIhAPK1DcB6fSarvYLZxEWG
+EvXJa/WfT+0gTdSicaG7VAYAHYAVYHUwhaQNgFK6gubVzxT8MDkOHhwJQgXL6Oq
HQcT0wwAAAFyu+veUAAABAMARzBFAiAWOwjw+5+GOjpuChHVfO6qhXJaDZIsarpB
mbyBBA4iGgIhANXqSeotxdFyeqULQpQDUjOJ9Vhb6C/Tlh2SmBHyhd2xMA0GCSqG
SIb3DQEBCwUAA4IBAQCTzy1S1xiSFjvzpHRCo2lQGdMx2OD161++gEArBCVu/P2f
TcwbI2dv5Xip62Ty7kK3wOBby+JceYET3+VeZ/X2/o89wg2QCyO4rRPiU5WwbEUE
/truWU/YKxkt1wBvMxDljdstxlmfyxYPwG/nmqs0AVqZ2GJMAbtIJSJDu3aMFNGJ
TvWsivpiTg+E8wiGK8t11j/0fJHpqSATXhDbzAF4KUu8NTtuyzFvW1j8Q3iLGx+S
fzSwCeYZNEEKI+WQO1gt6iJ1eCN7FXrQXCkRsYSh9Yyi0pVIj2EhvVeDe77NIN0b
IM6tU3qblo0VcaeO5pQ6lfYQ+dBJJUxJPIA0g2/u
-----END CERTIFICATE-----
 1 s:C = DE, O = Verein zur Foerderung eines Deutschen Forschungsnetzes e. V., OU = DFN-PKI, CN = DFN-Verein Global Issuing CA
   i:C = DE, O = Verein zur Foerderung eines Deutschen Forschungsnetzes e. V., OU = DFN-PKI, CN = DFN-Verein Certification Authority 2
-----BEGIN CERTIFICATE-----
MIIFrDCCBJSgAwIBAgIHG2O60B4sPTANBgkqhkiG9w0BAQsFADCBlTELMAkGA1UE
BhMCREUxRTBDBgNVBAoTPFZlcmVpbiB6dXIgRm9lcmRlcnVuZyBlaW5lcyBEZXV0
c2NoZW4gRm9yc2NodW5nc25ldHplcyBlLiBWLjEQMA4GA1UECxMHREZOLVBLSTEt
MCsGA1UEAxMkREZOLVZlcmVpbiBDZXJ0aWZpY2F0aW9uIEF1dGhvcml0eSAyMB4X
DTE2MDUyNDExMzg0MFoXDTMxMDIyMjIzNTk1OVowgY0xCzAJBgNVBAYTAkRFMUUw
QwYDVQQKDDxWZXJlaW4genVyIEZvZXJkZXJ1bmcgZWluZXMgRGV1dHNjaGVuIEZv
cnNjaHVuZ3NuZXR6ZXMgZS4gVi4xEDAOBgNVBAsMB0RGTi1QS0kxJTAjBgNVBAMM
HERGTi1WZXJlaW4gR2xvYmFsIElzc3VpbmcgQ0EwggEiMA0GCSqGSIb3DQEBAQUA
A4IBDwAwggEKAoIBAQCdO3kcR94fhsvGadcQnjnX2aIw23IcBX8pX0to8a0Z1kzh
axuxC3+hq+B7i4vYLc5uiDoQ7lflHn8EUTbrunBtY6C+li5A4dGDTGY9HGRp5Zuk
rXKuaDlRh3nMF9OuL11jcUs5eutCp5eQaQW/kP+kQHC9A+e/nhiIH5+ZiE0OR41I
X2WZENLZKkntwbktHZ8SyxXTP38eVC86rpNXp354ytVK4hrl7UF9U1/Isyr1ijCs
7RcFJD+2oAsH/U0amgNSoDac3iSHZeTn+seWcyQUzdDoG2ieGFmudn730Qp4PIdL
sDfPU8o6OBDzy0dtjGQ9PFpFSrrKgHy48+enTEzNAgMBAAGjggIFMIICATASBgNV
HRMBAf8ECDAGAQH/AgEBMA4GA1UdDwEB/wQEAwIBBjApBgNVHSAEIjAgMA0GCysG
AQQBga0hgiweMA8GDSsGAQQBga0hgiwBAQQwHQYDVR0OBBYEFGs6mIv58lOJ2uCt
sjIeCR/oqjt0MB8GA1UdIwQYMBaAFJPj2DIm2tXxSqWRSuDqS+KiDM/hMIGPBgNV
HR8EgYcwgYQwQKA+oDyGOmh0dHA6Ly9jZHAxLnBjYS5kZm4uZGUvZ2xvYmFsLXJv
b3QtZzItY2EvcHViL2NybC9jYWNybC5jcmwwQKA+oDyGOmh0dHA6Ly9jZHAyLnBj
YS5kZm4uZGUvZ2xvYmFsLXJvb3QtZzItY2EvcHViL2NybC9jYWNybC5jcmwwgd0G
CCsGAQUFBwEBBIHQMIHNMDMGCCsGAQUFBzABhidodHRwOi8vb2NzcC5wY2EuZGZu
LmRlL09DU1AtU2VydmVyL09DU1AwSgYIKwYBBQUHMAKGPmh0dHA6Ly9jZHAxLnBj
YS5kZm4uZGUvZ2xvYmFsLXJvb3QtZzItY2EvcHViL2NhY2VydC9jYWNlcnQuY3J0
```

[32] https://security.stackexchange.com/questions/73127/how-can-you-check-if-a-private-key-and-certificate-match-in-openssl-with-ecdsa

```
MEoGCCsGAQUFBzAChj5odHRwOi8vY2RwMi5wY2EuZGZuLmRlL2dsb2JhbC1yb290
LWcyLWNhL3B1Yi9jYWNlcnQvY2FjZXJ0LmNydDANBgkqhkiG9w0BAQsFAAOCAQEA
gXhFpE6kfw5V8Amxaj54zGg1qRzzlZ4/8/jfazh3iSyNta0+x/KUzaAGrrrMqLGt
Mwi2JIZiNkx4blDw1W5gjU9SMUOXRnXwYuRuZlHBQjFnUOVJ5zkey5/KhkjeCBT/
FUsrZpugOJ8Azv2n69F/Vy3ITF/cEBGXPpYEAlyEqCk5bJT8EJIGe57u2Ea0G7UD
DDjZ3LCpP3EGC7IDBzPCjUhjJSU8entXbveKBTjvuKCuL/TbB9VbhBjBqbhLzmyQ
GoLkuT36d/HSHzMCv1PndvncJiVBby+mG/qkE5D6fH7ZC2Bd7L/KQaBh+xFJKdio
LXUV2EoY6hbvVTQiGhONBg==
-----END CERTIFICATE-----
 2 s:C = DE, O = Verein zur Foerderung eines Deutschen Forschungsnetzes e. V., OU = DFN-PKI, CN = DFN-Verein Certification Authority 2
   i:C = DE, O = T-Systems Enterprise Services GmbH, OU = T-Systems Trust Center, CN = T-TeleSec GlobalRoot Class 2
-----BEGIN CERTIFICATE-----
MIIFEjCCA/qgAwIBAgIJAOML1fivJdmBMA0GCSqGSIb3DQEBCwUAMIGCMQswCQYD
VQQGEwJERTErMCkGA1UECgwiVC1TeXN0ZW1zIEVudGVycHJpc2UgU2VydmljZXMg
R21iSDEfMB0GA1UECwwWVC1TeXN0ZW1zIFRydXN0IENlbnRlcjElMCMGA1UEAwwc
VC1UZWxlU2VjIEdsb2JhbFJvb3QgQ2xhc3MgMjAeFw0xNjAyMjIxMzM4MjJaFw0z
MTAyMjIyMzU5NTlaMIGVMQswCQYDVQQGEwJERTFFMEMGA1UEChM8VmVyZWluIHp1
ciBGb2VyZGVydW5nIGVpbmVzIERldXRzY2hlbiBGb3JzY2h1bmdzbmV0emVzIGUu
IFYuMRAwDgYDVQQLEwdERk4tUEtJMS0wKwYDVQQDEyRERk4tVmVyZWluIENlcnRp
ZmljYXRpb24gQXV0aG9yaXR5IDIwggEiMA0GCSqGSIb3DQEBAQUAA4IBDwAwggEK
AoIBAQDLYNf/ZqFBzdL6h5eKc6uZTepnOVqhYIBHFU6MlbLlz87TV0uNzvhWbBVV
dgfqRv3IA0VjPnDUq1SAsSOcvjcoqQn/BV0YD8SYmTezIPZmeBeHwp0OzEoy5xad
rg6NKXkHACBU3BVfSpbXeLY008F0tZ3pv8B3Teq9WQfgWi9sPKUA3DW9ZQ2PfzJt
8lpqS2IB7qw4NFlFNkkF2njKam1bwIFrEczSPKiL+HEayjvigN0WtGd6izbqTpEp
PbNRXK2oDL6dNOPRDReDdcQ5HrCUCxLx1WmOJfS4PSu/wI7DHjulv1UQqyquF5de
M87I8/QJB+MChjFGawHFEAwRx1npAgMBAAGjggF0MIIBcDAOBgNVHQ8BAf8EBAMC
AQYwHQYDVR0OBBYEFJPj2DIm2tXxSqWRSuDqS+KiDM/hMB8GA1UdIwQYMBaAFL9Z
IDYAeaCgImuM1fJh0rgsy4JKMBIGA1UdEwEB/wQIMAYBAf8CAQIwMwYDVR0gBCww
KjAPBg0rBgEEAYGtIYIsAQEEMA0GCysGAQQBga0hgiweMAgGBmeBDAECAjBMBgNV
HR8ERTBDMEGgP6A9hjtodHRwOi8vcGtpMDMzNi50ZWxlc2VjLmRlL3JsL1RlbGVT
ZWNfR2xvYmFsUm9vdF9DbGFzc18yLmNybDCBhgYIKwYBBQUHAQEEejB4MCwGCCsG
AQUFBzABhiBodHRwOi8vb2NzcDAzMzYudGVsZXNlYy5kZS9vY3NwcjBIBggrBgEF
BQcwAoY8aHR0cDovL3BraTAzMzYudGVsZXNlYy5kZS9jcnQvVGVsZVNlY19HbG9i
YWxSb290X0NsYXNzXzIuY2VyMA0GCSqGSIb3DQEBCwUAA4IBAQCHC/8+AptlyFYt
1juamItxT9q6Kaoh+UYu9bKkD64ROHk4sw50unZdnugYgpZi20wz6N35at8yvSxM
R2BVf+d0a7Qsg9h5a7a3TVALZge17bOXrerufzDmmf0i4nJNPoRb7vnPmep/11I5
LqyYAER+aTu/de7QCzsazeX3DyJsR4T2pUeg/dAaNH2t0j13s+70103/w+jlkk9Z
PpBHEEqwhVjAb3/4ru0IQp4e1N8ULk2PvJ6Uw+ft9hj4PEnnJqinNtgs3iLNi4LY
2XjiVRKjO4dEthELlQxSr2mMDwbf0KJTi1eYe8/9ByT0/L3D/UqSApcb8re2z2WK
GqK1chk5
-----END CERTIFICATE-----
---
Server certificate
subject=C = DE, ST = Nordrhein-Westfalen, L = Siegen, O = Universitaet Siegen, CN = mail.uni-siegen.de

issuer=C = DE, O = Verein zur Foerderung eines Deutschen Forschungsnetzes e. V., OU = DFN-PKI, CN = DFN-Verein Global Issuing CA
---
No client certificate CA names sent
Peer signing digest: SHA256
Peer signature type: RSA
Server Temp Key: ECDH, P-256, 256 bits
---
SSL handshake has read 5490 bytes and written 446 bytes
Verification: OK
---
New, TLSv1.2, Cipher is ECDHE-RSA-AES128-GCM-SHA256
Server public key is 2048 bit
Secure Renegotiation IS supported
Compression: NONE
Expansion: NONE
No ALPN negotiated
SSL-Session:
    Protocol  : TLSv1.2
    Cipher    : ECDHE-RSA-AES128-GCM-SHA256
    Session-ID: 541F00003A7E8C0C2752B7A9311FFF3F3FFEF9906C73CD66A8E32090AC203137
    Session-ID-ctx:
    Master-Key: F95B5BEEACC98FB50B014F90A64D3E733304DF995105C25EFB4E6983ECBF9D170E8CF51845C5CF47F18BE2B530782D7C
    PSK identity: None
    PSK identity hint: None
    SRP username: None
    Start Time: 1592595632
    Timeout   : 7200 (sec)
    Verify return code: 0 (ok)
    Extended master secret: yes
---
* OK University Siegen IMAP4 service is ready.
```

1.14 OpenSSL 3 in CrypTool-Online (CTO)

OpenSSL 3.0.0 was officially released beginning of September 2021 after 3 years of development.[33] From version 3, OpenSSL is under the Apache 2.0 license, which simplifies its use in other open-source projects.

As part of a bachelor thesis a plugin for CrypTool-Online was built that made OpenSSL available in the browser: the OpenSSL app. This was done by first porting OpenSSL 1.1.1 to WebAssembly (wasm); and with this experience it was easy to bring the then released OpenSSL 3.0.0 to WebAssembly and make it available still in September 2021.

This CTO plugin (web app) then was enhanced with a GUI[34] and it can be found at:

`https://www.cryptool.org/en/cto/openssl`

Here you can perform (almost) all OpenSSL command line commands as long as they use no special feature of the Bash shell apart from the pipe and besides `echo` no further Linux commands.[35] As a result, the OpenSSL app can ad hoc execute almost all commands in this chapter without having to install OpenSSL. This should be of particular interest on the go or for Windows and Mac users who need an OpenSSL command quickly.

To demonstrate and support teaching, many OpenSSL functions are not only available in the terminal of the CTO OpenSSL plugin, but also directly via the GUI in 6 tabs – also see Fig. 1:

1. Encrypt & Decrypt
2. Generate Keys
3. Sign & Verify
4. Certificates
5. Hashes
6. Files

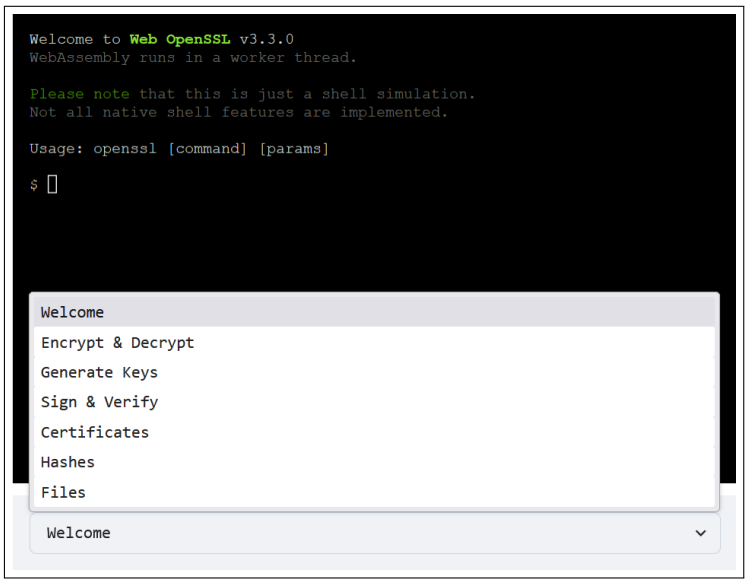

Fig. 1: In CTO: OpenSSL 3 in a browser: The six tab of the GUI

[33] https://www.openssl.org/blog/blog/2021/09/07/OpenSSL3.Final/
[34] The Web terminal in it runs in its own thread and the plugin can use a browser fullscreen view.
[35] This is the status as of June 2024. These restrictions on the web terminal are to be gradually removed.

If you perform an action in the **GUI**, the corresponding OpenSSL command is displayed synchronously on the **command line**. This is helpful to learn the OpenSSL commands.

The next three screenshots from CTO (Figs. 2, 4 and 5) show OpenSSL within the browser:

1. Figure 2 shows how a text was encrypted using the symmetric cipher AES and a password. The password is used as input for the key derivation function PBKDF2 which generates the salt, the initialization vector (IV) and the symmetric key. Alternatively, the salt and initialization vector can be entered manually. With the additional option `-p`, the hex values for salt, IV, and key generated by the key derivation function are also output. The option `-a` causes the output to be base64 encoded. The screenshot shows the selection of the corresponding fields in the GUI and what the corresponding OpenSSL command looks like on the command line. For instance like this:

   ```
   $ echo Hello World! | openssl enc -aes-256-cbc -e -k abc -pbkdf2 -a
   ```

   The encrypted message can only be decrypted by someone who knows the appropriate key or password.

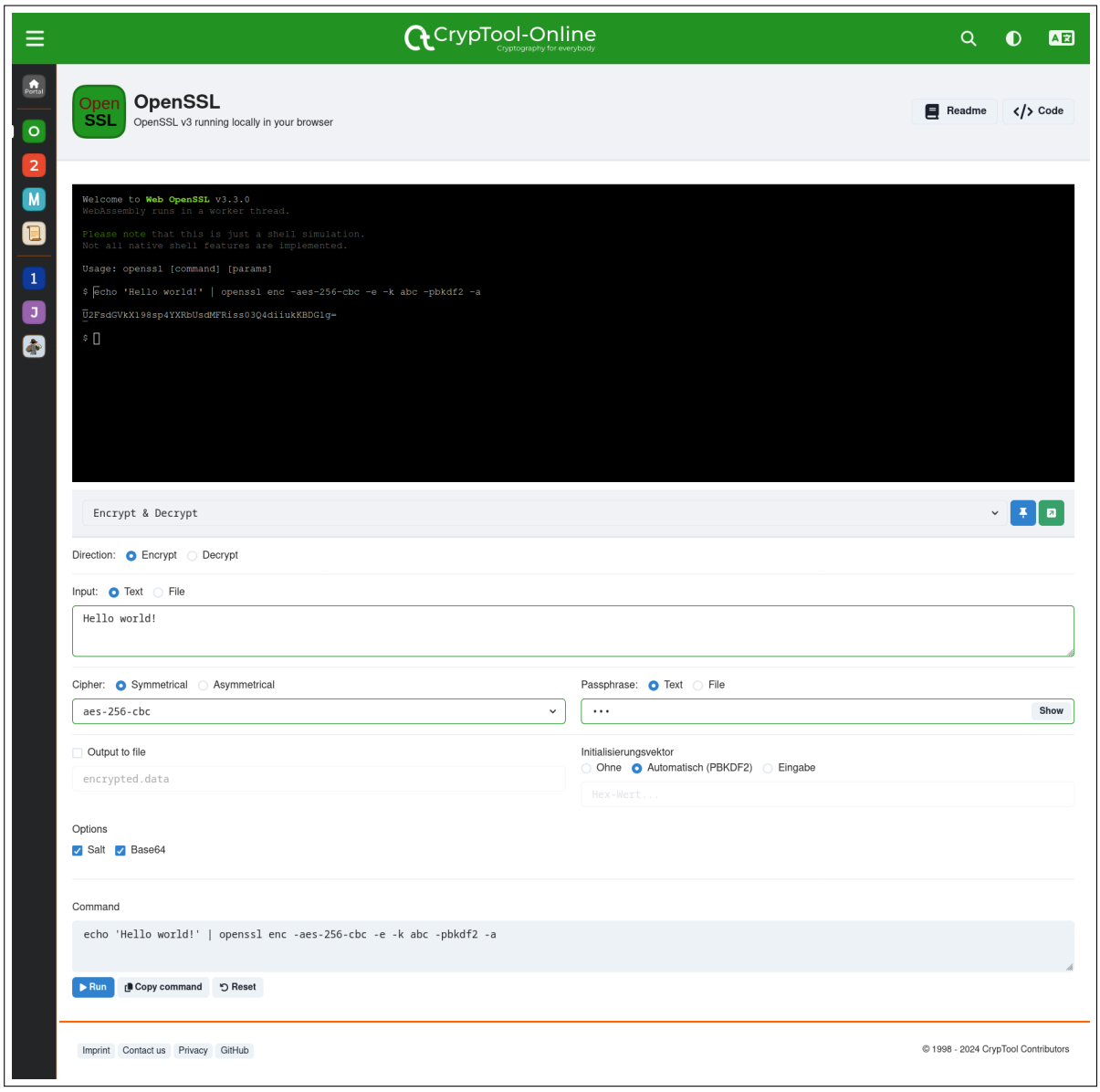

Fig. 2: In CTO: OpenSSL 3 in a browser: AES encryption

2. Figure 4 on page 31 shows the "Certificates" tab. The sender of a message (e. g. Alice) needs a certificate in order to distribute it to possible recipients of messages so that the recipients can verify the authenticity of a message signed by her.

   Before we look at the process in the CTO app (Fig. 4), Fig. 3 shows the process between the parties involved in a diagram. Here, the certificate is signed by a third party (certification authority, CA). However, a certificate can also be self-signed.

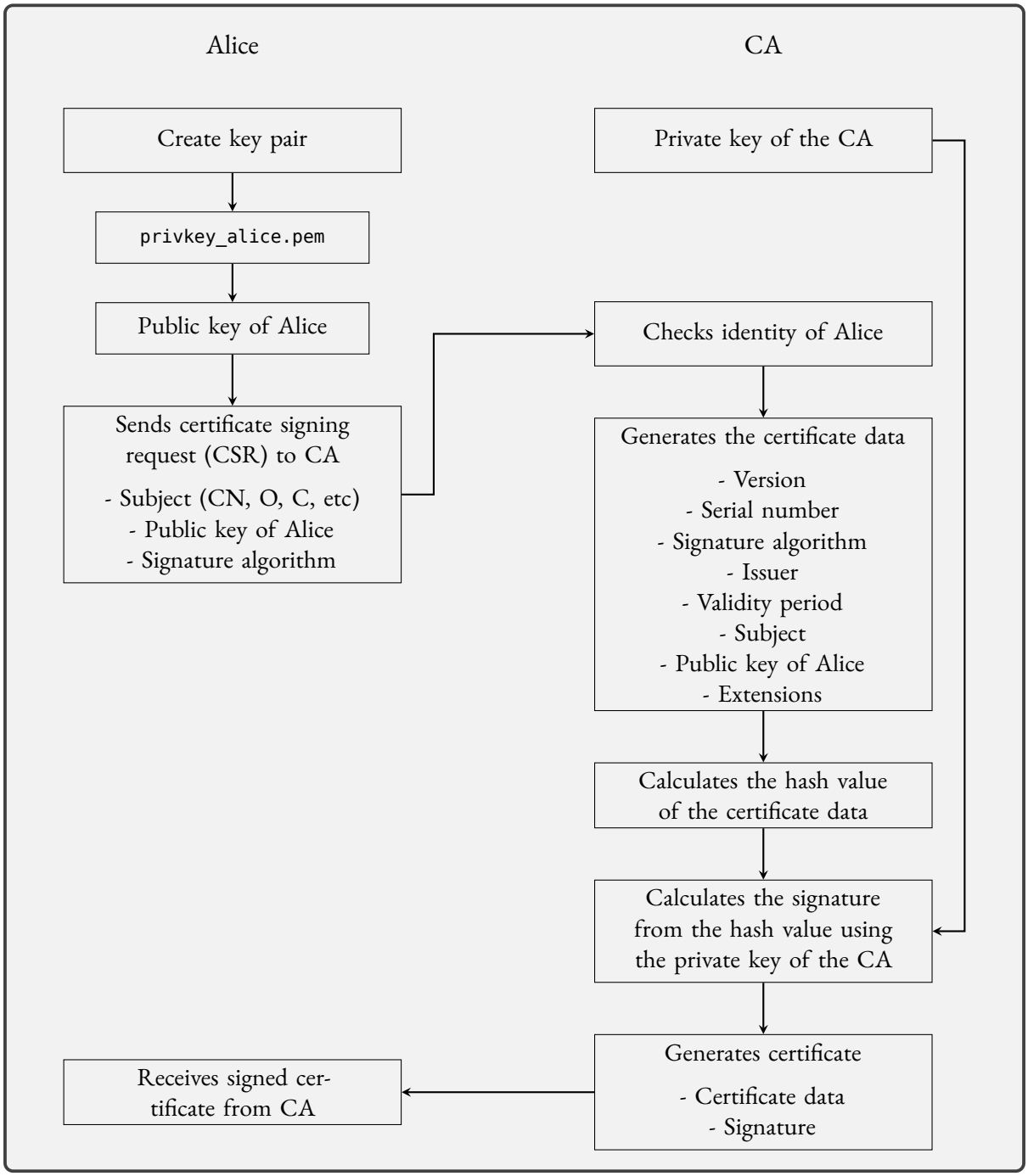

Fig. 3: Creation of a certificate signed by a CA for Alice

Two things are needed to create a certificate: You need a private key to sign the certificate (create signature) and you need the data about the certificate holder (in this case Alice) including their public key. A finished certificate therefore consists of the data part and the signature part.

In the GUI, a self-signed certificate has been selected as a variant so that both keys can come from the

certificate holder. You can use an existing key pair or generate a new key pair. With OpenSSL, the file for the "private key" always contains the corresponding public key. In the screenshot, a new RSA key pair is generated with a modulus length of 2048 bits. The data part is hashed for the signature, by default with SHA-256. You can also select a different hash function, e. g. Keccak with the option `-sha3-256`. The validity period of the certificate is entered in the certificate. The password for encryption is only used when a new key pair is generated.

OpenSSL generates the certificate in one step (a single command). The command then generates two things: a private key and the certificate. Depending on the option selected, these are written to a file or displayed in the black terminal on the command line.

If you do not create a self-signed certificate, you send the necessary data to the CA in the form of a certificate signing request (CSR). This can be simulated with the other two variants. Since you can build an entire PKI with OpenSSL, the possibilities are far more extensive than what you can click together in the GUI.

The screenshot shows the selection of the corresponding fields in the GUI and what the corresponding OpenSSL command looks like on the command line. For instance like this:

```
openssl req -x509 -out cert.pem -days 30 -noenc -newkey rsa\:2048 -keyout private.key -subj '/C=US/L=San
Francisco/ST=California/O=ExampleCorp/OU=IT Department/CN=www.example.com/emailAddress=info@examle.com'
 -addext subjectAltName\=email\:my\@example.com -batch
```

The default values for the subject of a certificate are C=AU, ST=Some-State, O=Internet Widgits Pty Ltd. The following command can be used to display the subject data of a certificate:

```
openssl x509 -in cert.pem -noout -subject
```

3. Figure 5 on page 32 shows the "Files" tab, in which a file can be seen in the virtual data system that contains the private key that was generated beforehand via OpenSSL. Technically this is achieved via WebAssembly which has a virtual memory in the browser in which the generated files are stored. Input files can also be uploaded there.

Each of the 6 tabs (see Fig. 1 on page 27) can also be pinned on the right-hand side in a navigation bar (see Fig. 6 on page 33). This is useful if you want to display two tabs at the same time in the OpenSSL app. So you can see and use the functions of two tabs at the same time. For example, when working with files, pinning the files tab can be advantageous: You can see directly whether a file has been created, or you can upload and manage required files that you want to encrypt, decrypt, or sign. Figure 6 shows how a file is encrypted on the left. The result is a file, which is shown at once on the right.

Note: From Q4/2024, the following YouTube playlist from Nils Kopals offers a "Comprehensive OpenSSL Course with CrypTool-Online for Beginners and Advanced":

https://www.youtube.com/playlist?list=PLMuvAbyIl0PQxywnjQ_Axzt8eicwXb_dp

Thus, with this CTO plugin even as a beginner, you can get to know the OpenSSL commands and even use OpenSSL on your smartphone.

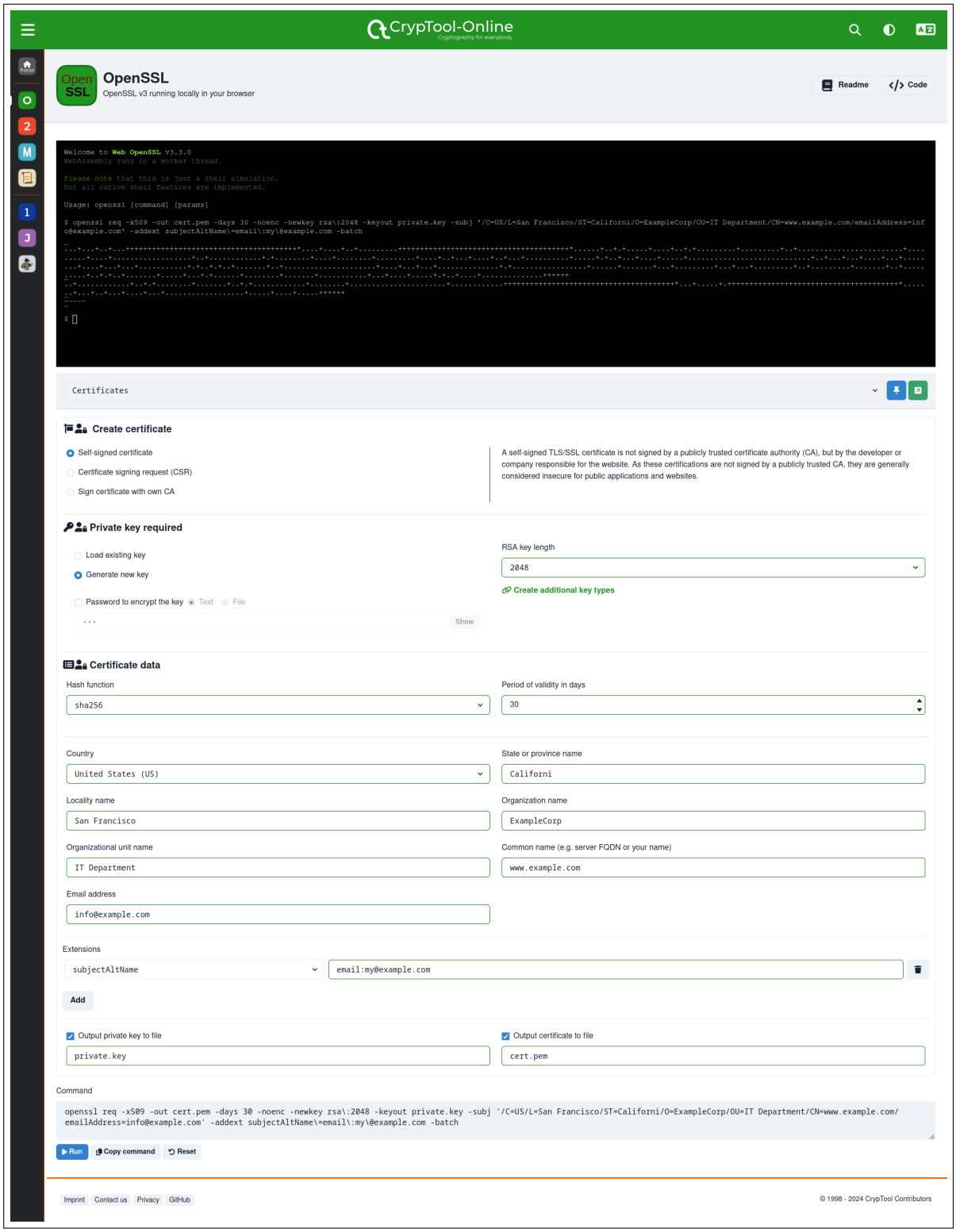

Fig. 4: In CTO: OpenSSL 3 in a browser: Creating a digital certificate

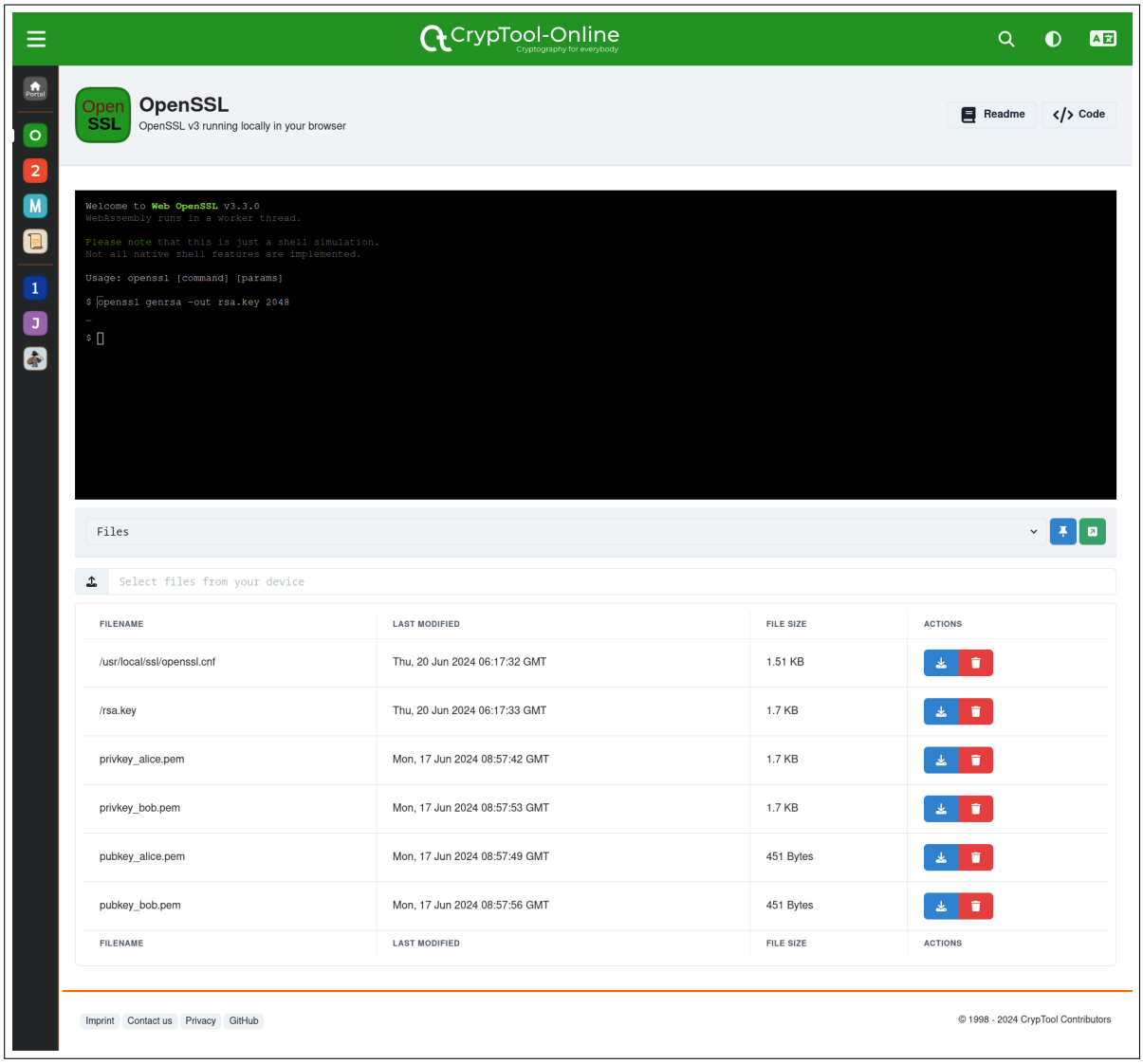

Fig. 5: In CTO: OpenSSL 3 in a browser: Files in the virtual file system of the browser

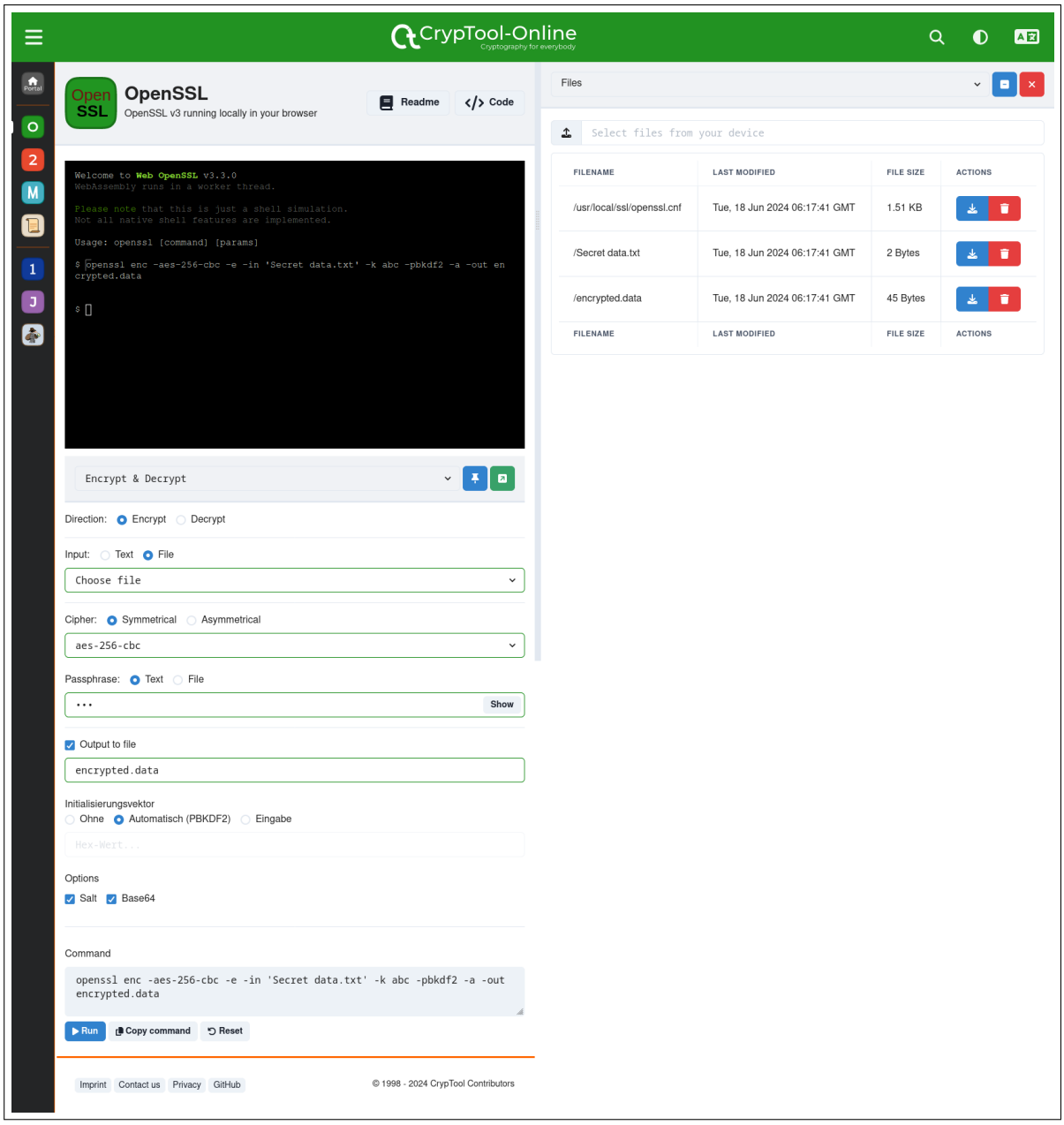

Fig. 6: In CTO: OpenSSL 3 in a browser: The files tab is pinned on the right

1.15 Web links of this Appendix 1

1. *OpenSSL for Web (via WebAssembly in the browser)*
   https://www.cryptool.org/en/cto/openssl
   https://wiki.openssl.org/index.php/Binaries
2. *OpenSSL – Cryptography and SSL/TLS Toolkit*, 2020, primary source
   https://www.openssl.org/
   https://wiki.openssl.org/index.php/Main_Page
   https://wiki.openssl.org/index.php/Standard_commands
   https://wiki.openssl.org/index.php/Command_Line_Utilities#ciphers
   https://www.openssl.org/docs/manmaster/man1/enc.html
3. *Getting started with OpenSSL: Cryptography basics*, 2019, good tutorial by Marty Kalin
   https://opensource.com/article/19/6/cryptography-basics-openssl-part-1
4. *How to use OpenSSL: Hashes, digital signatures, and more*, 2019, Marty Kalin
   https://opensource.com/article/19/6/cryptography-basics-openssl-part-2
5. *A 6 Part Introductory OpenSSL Tutorial*, 2019, good tutorial by Cody Arsenault
   https://www.keycdn.com/blog/openssl-tutorial
6. *An Introduction to the OpenSSL command line tool*, 2004, tutorial by Philippe Camacho
   https://users.dcc.uchile.cl/~pcamacho/tutorial/crypto/openssl/openssl_intro.html
7. *OpenSSL Command-Line HOWTO*, 2016, cookbook-style tutorial by Paul Heinlein
   https://www.madboa.com/geek/openssl/
8. Wikipedia: *RSA (cryptosystem)*
   https://en.wikipedia.org/wiki/RSA_(cryptosystem)#Example
9. *Hybrid encryption with OpenSSL* using openssl genrsa and openssl rsa, 2015
   https://gist.github.com/markus-k/fc37c36e13b32c8a6dd1
10. *OpenSSL Cookbook – A Short Guide to the Most Frequently Used OpenSSL Features and Commands*, 3rd ed., 2022, Ivan Ristić
    https://www.feistyduck.com/library/openssl-cookbook/online/

All links have been confirmed at June 5, 2024.

The `openssl` examples from this book plus their according Shell and Python scripts can be found on the CrypTool website. Further details can be found in the list on page 35.

2 Lists of Figures, Tables, Code Examples, etc.

2.1 List of Figures

2.2 List of OpenSSL Examples

The introduction of the CLI **OpenSSL** in Section 1 on page 5 uses the following two Bash shell scripts and two Python scripts:

- `hybrid-openssl-enc-dec.sh` with commands from A.1.2–A.1.6, A.1.8, and A.1.9; calling
  - `convert-pem-dec.py`
- `zahl-direkt-openssl-enc-dec.sh` with commands from A.1.7; calling
  - `int2bin.py`

You can find these four scripts with the `openssl` examples from this book on the CrypTool website:
`https://www.cryptool.org/en/ctbook/openssl`

All examples have been tested with OpenSSL versions 1.1.1l and 3.3.0.

3 Index