Anhang zum
CrypTool-Buch: Kryptografie lernen
und anwenden mit CrypTool und SageMath

Kurze Einführung in das CLI von OpenSSL

Bernhard Esslinger

Anhang zum “CrypTool-Buch”:
Kryptografie lernen und anwenden mit CrypTool und SageMath

Kurzeinführung in das CLI OpenSSL

Bernhard Esslinger

https://www.cryptool.org

29. Juni 2024 (18:00:00)

Dies ist ein freies Dokument, d. h. Inhalte des Dokuments können kopiert und verbreitet werden, auch zu kommerziellen Zwecken. Im Gegenzug sind Autor, Titel und die CrypTool-Webseite (https://www.cryptool.org) zu nennen. Selbstverständlich darf aus dem CrypTool-Buch, genau wie aus jedem anderen Werk auch, zitiert werden. Darüber hinaus unterliegt dieses Dokument der GNU-Lizenz für freie Dokumentation. Dies betrifft auch den Code für die SageMath- und OpenSSL-Beispiele in diesem Dokument.

Copyright © 1998–2024 Bernhard Esslinger and the CrypTool Team.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation (FSF). A copy of the license is included in the section entitled "GNU Free Documentation License".

Zum Referenzieren per Bib(La)TEX:

```
@article{Esslinger:ctb_2024_app_openssl_de,
  editor    = {Bernhard Esslinger},
  title     = {Anhang zum CrypTool-Buch: Kryptografie lernen
               und anwenden mit CrypTool und SageMath:
               Kurzeinführung in das CLI OpenSSL},
  publisher = {CrypTool-Projekt},
  year      = {2024}
}
```

Schriftsatz-Software: LaTeX
Versionsverwaltungs-Software: Git
Unterstützung bei der Übersetzung: DeepL-Übersetzer von DeepL.com und ChatGPT 4 von openai.com
Unterstützung beim Generieren von Tikz-Grafik: Claude 3.5 Sonnet von claude.ai

Inhaltsverzeichnis

1 **Kurzeinführung in das CLI `openssl`** . . . 5
1.1 Ausgabe aller `openssl`-Kommandos . . . 6
1.2 Symmetrische Verschlüsselung mit `openssl` . . . 7
1.2.1 Zufälligen Sessionkey für AES erzeugen . . . 7
1.2.2 Eine Nachricht mit AES und einem zufälligen Sessionkey verschlüsseln . . . 7
1.2.3 Eine Nachricht mit AES und einem Passwort verschlüsseln (Variante 2) . . . 8
1.2.4 Entschlüsseln einer Datei mit AES-256 . . . 8
1.3 Asymmetrische Verschlüsselung mit `openssl`: Schlüsselerzeugung . . . 8
1.3.1 Einen privaten RSA-Schlüssel der Länge 2048 bit erzeugen . . . 8
1.3.2 OpenSSL-Datei „Privater Schlüssel“ . . . 9
1.3.3 OpenSSL-Datei „Öffentlicher Schlüssel“ . . . 10
1.4 Asymmetrische Ver- und Entschlüsselung mit `openssl`: noch keine hybride Verschlüsselung . . . 11
1.4.1 Verschlüsselung mit RSA (kein Textbook-RSA) . . . 11
1.4.2 Entschlüsselung mit RSA (kein Textbook-RSA) . . . 11
1.5 Hybride Ver- und Entschlüsselung mit `openssl` . . . 11
1.5.1 Vorbereitung beim Empfänger Bob: RSA-Schlüsselgenerierung . . . 12
1.5.2 Verschlüsselung: Drei Aufgaben auf der Seite der Senderin Alice . . . 12
1.5.3 Entschlüsselung: Zwei Aufgaben auf der Seite des Empfängers Bob . . . 12
1.6 Alle Schlüssel einer privaten PEM-Datei als Dezimalzahl ausgeben – eigenes Python-Skript . . . 13
1.7 Schlüssel einer PEM-Datei als Dezimalzahl ausgeben – per RsaCtfTool) . . . 14
1.8 Überblick über alle vorhergehenden OpenSSL-Befehle (als Liste und als Shell-Skript) . . . 14
1.9 Textbook-RSA mit `openssl` und eigenem Python-Skript . . . 17
1.10 Zufallswerte erzeugen . . . 19
1.11 Primzahlen mit OpenSSL erzeugen . . . 19
1.12 Geschwindigkeitsvergleiche mit `openssl` . . . 20
1.13 Zertifikate abrufen und evaluieren . . . 22
1.14 OpenSSL 3 in CrypTool-Online (CTO) . . . 28
1.15 Web-Links zu diesem Anhang 1 . . . 35

2 Verzeichnisse der Abbildungen, Tabellen, Code-Beispiele, etc. 36
2.1 Abbildungsverzeichnis . . . 36
2.2 Verzeichnis der OpenSSL-Programmbeispiele . . . 36

3 Index 37

1 Kurzeinführung in das CLI `openssl`

(Bernhard Esslinger. Letzter Update: Juni 2024)

OpenSSL[1] ist eine weit verbreitete, freie Open-Source-Kryptobibliothek, die bspw. produktiv benutzt wird, um auf Webservern TLS-/SSL-Funktionalität zur Verfügung zu stellen. OpenSSL ist das erste nach FIPS 140-2 zertifizierte Open-Source-Programm.

Aber man kann mit dem Kommandozeilenprogramm nicht nur das TLS-Protokoll abbilden und Zertifikate verwalten, sondern auch viele kryptografische Einzelfunktionen ausführen (immer aber nur für produktive Parametersets, nicht für kleine Werte, wie sie in didaktischen Lernprogrammen verwendet werden). Unter Unix ist OpenSSL normalerweise gleich mitinstalliert.[2]

Zu OpenSSL gehört auch das englisch-sprachige Kommandozeilentool[3] `openssl`, mit dem man die Funktionalität auf vielen Betriebssystemen direkt ausprobieren kann.[4]

Im Folgenden werden viele Beispiele aus der Kryptografie mit dem CLI `openssl` implementiert und bei Bedarf mit einem Python-Skript erweitert. Der Code ist lauffähig und getestet mit den OpenSSL-Versionen 1.1.1 (unter Ubuntu 20.04) und 3.0 (unter Ubuntu 22.04). Zu OpenSSL im Browser siehe Abschnitt 1.14 auf Seite 28.

Abfrage der OpenSSL-Version

Um die Versionsnummer der bei Ihnen installierten Version von OpenSSL abzufragen, können Sie einen der folgenden drei Befehle eingeben:

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

Das Ergebnis ist immer dasselbe, beispielsweise:

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

Eine komplette Versionsausgabe inkl. der beim Bauen von OpenSSL verwendeten Compiler-Flags erhält man mit der Option `-a`. Diese Versionsausgabe sieht auf verschiedenen Betriebssystemen unterschiedlich aus. Am ausführlichsten ist sie unter Linux. Hier die Ausgabe von einer WebAssembly-Version von 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] Siehe auch https://www.openssl.org/ und https://de.wikipedia.org/wiki/OpenSSL. Weitere kommentierte Links zur Vertiefung finden sich am Ende dieses Anhangs auf Seite 35.

[2] Updaten kann man OpenSSL per `sudo apt-get install openssl`, wobei die vorgepackte Version deutlich von der neuesten stabilen Version, die man sich aus den Sourcen selbst erstellen muss, abweichen kann (beim Schreiben der ersten Version dieses Anhangs gab es unter Ubuntu 18.04 per `apt-get` die OpenSSL-Major-Version 1.1.1 vom 11.9.2019, während die letzte Stable die Minor-Version 1.1.1g vom 21.4.2020 war. Im April 2024 gab es unter Ubuntu 22.04 standardmäßig die Version OpenSSL 3.0.2 vom 15.3.2022, obwohl OpenSSL 3.3.0 schon verfügbar war).

[3] Kommandozeilenprogramm = command line interface = CLI

[4] Im Gegensatz zum ebenfalls sehr verbreiteten Kommandozeilentool `gpg` von GNU Privacy Guard (https://de.wikipedia.org/wiki/GNU_Privacy_Guard) erlaubt `openssl` auch Aufrufe, die sehr ins Detail gehen.
Bei `gpg` liegt der Schwerpunkt nur auf den im praktischen Einsatz befindlichen Cipher-Suites. Einfach genau einen Block ohne Padding zu verschlüsseln wie im OpenSSL-Beispiel 1.11.1 des Buchs geht unseres Wissens mit `gpg` nicht. Andererseits kann man auch bei OpenSSL bspw. RSA und TLS nur mit bestimmten Mindestgrößen aufrufen (bspw. 512 bit für die Länge des RSA-Moduls).

1.1 Ausgabe aller openssl-Kommandos

Was an Funktionen in openssl verfügbar ist, erhält man per:

```
$ openssl help
```

Die Ausgabe, die diesem Befehl folgt, ist in drei Gruppen unterteilt: die sogenannten Standard-Befehle, die Befehle mit Hash-Funktionen („Message-Digest“) und die Befehle mit Verschlüsselungs- und Codierungsfunktionen („Cipher“)[5].

```
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
```

Alle Kommandos haben wieder Optionen, die man sich bspw. folgendermaßen anzeigen lassen kann:

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

[5] Nicht dabei in der Cipher-Liste ist bspw. das Verschlüsselungsverfahren ChaCha20, das man in openssl nur in Cipher-Suites und in Kombination mit dem MAC Poly1305 aufrufen kann.

1.2 Symmetrische Verschlüsselung mit openssl

Das symmetrische Verschlüsselungsverfahren AES braucht als Eingabe eine Nachricht m, einen Schlüssel (Sessionkey) k und einen Initialisierungs-Vektor (IV).[6]

Es gibt verschiedene Arten, wie man AES mit OpenSSL aufrufen kann. Hier werden zwei Varianten vorgestellt: Einmal werden die Parameter explizit übergeben, einmal werden sie aus dem übergebenen Passwort intern hergeleitet (dazu benutzt OpenSSL eine eingebaute Schlüsselherleitungsfunktion wie PBKDF2). Die Nutzung eines Passworts und einer Schlüsselherleitungsfunktion ist der übliche Weg.

Der Sessionkey kann sowohl direkt auf der Kommandozeile stehen als auch aus einer Datei gelesen werden. Beim Passwort gibt es sogar drei Möglichkeiten: Es kann direkt auf der Kommandozeile stehen oder aus einer Datei gelesen werden oder man wird nach einem Passwort gefragt.

1.2.1 Zufälligen Sessionkey für AES erzeugen

Zuerst erzeugen wir einen 256-bit langen Schlüssel und speichern ihn in `symkey.bin`. Das Argument 32 bedeutet, dass man 32 byte, also 256 bit Länge haben möchte.

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

Dann erzeugen wir einen zufälligen IV von 16 byte = 128 bit = 32 Hexzeichen Länge. Beispiel-Output könnte sein:

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

Die im vorhinein erzeugten zufälligen Werte für den Sessionkey und den IV braucht man nur in der ersten Variante, wo die Parameter explizit an OpenSSL übergeben werden. In der ersten Variante müssen sowohl Sessionkey als auch IV explizit angegeben werden.

1.2.2 Eine Nachricht mit AES und einem zufälligen Sessionkey (Variante 1) verschlüsseln

Wir verschlüsseln die Datei `message.txt` mit AES-256 im CFB8-Blockmodus und benutzen den oben erzeugten IV als weiteren Input. Als Output erhalten wir eine Datei namens `message.txt.enc`, die den Geheimtext enthält. Mit der Option `-p` kann man sich anzeigen lassen, welche Parameter dieses OpenSSL-Kommando nutzte. Der Sessionkey wird nach `-K` eingegeben – einmal direkt als Hexwert und einmal, nachdem er aus der Datei `symkey.bin` ausgelesen wurde.

```
$ 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
```

[6] Mit `-iter` kann man den Iterationszähler spezifizieren, den die Schlüsselherleitungsfunktion (KDF) nutzt. Weiterhin nutzt die KDF per Default einen Saltwert, den man mit `-nosalt` abschalten kann.

1.2.3 Eine Nachricht mit AES und einem Passwort verschlüsseln (Variante 2)

Wir verschlüsseln die Datei `message.txt` wie zuvor, benutzen aber statt der expliziten Parameter ein Passwort. Dies ist der empfohlene Weg. Zum Passwort kommt der Befehl auf eine von drei verschiedenen Arten: Der Befehl fragt danach, oder das Passwort wird nach der Option `-pass pass:` auf der Kommandozeile eingegeben, oder es wird nach der Option `-pass file:` aus der Datei `symkey.bin` ausgelesen.[7] Diese Datei wurde zuvor (wie in 1.2.1) als zufällige Passwort-Datei erzeugt. Eine Option `-iv` ist optional in allen Fällen möglich.

```
$ 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
```

Nachlesen kann man das auch in der OpenSSL-Online-Doku: `https://wiki.openssl.org/index.php/Enc`

1.2.4 Entschlüsseln einer Datei mit AES-256

Wir entschlüsseln die Datei `message.txt.enc`. Hierzu braucht man die zusätzliche Option `-d`. Als Output erhalten wir eine Datei namens `message.txt.enc.dec`, die wieder den Klartext enthält.

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

Die Dateien `message.txt` und `message.txt.enc.dec` müssen dann gleich sein.[8,9]

1.3 Asymmetrische Verschlüsselung mit `openssl`: Schlüsselerzeugung

Für das asymmetrische Verschlüsselungsverfahren RSA benötigt jeder Teilnehmer zuerst ein Schlüsselpaar, das aus einem öffentlichen und einem privaten Schlüssel besteht. Dann müssen beide Seiten ihren öffentlichen Schlüssel austauschen. So entsteht die nötige „Public-Key-Infrastruktur“.

Der Sender verschlüsselt dann mit dem öffentlichen Schlüssel des Empfängers, der Empfänger entschlüsselt mit seinem privaten Schlüssel.

1.3.1 Einen privaten RSA-Schlüssel der Länge 2048 bit erzeugen

Als Länge eines RSA-Schlüssels wird die Länge des Moduls[10] genommen.

Mit dem folgenden Befehl wird der private Schlüssel erzeugt und im PKCS#8-Format[11] in der Datei `privatekey▸▸.pem` gespeichert (base64-codiert). Bei diesem Format wird in der Datei neben dem Schlüssel auch die OID[12], die den Schlüsseltyp identifiziert, abgelegt.[13,14]

[7] Veraltete Optionen zur Passwortübergabe sind `-k` und `-kfile`. Nach `https://wiki.openssl.org/index.php/Enc` sind diese Optionen überholt und sollten ersetzt werden durch `-pass pass:` and `-pass file:`.

[8] Es ist egal, ob man `enc -aes` oder `aes` als Option für `openssl` verwendet, also `aes` *ohne* das vorgestellte „-“, vgl. Codebeispiel auf S. 12.

[9] Für das Tagesgeschäft eines Endusers empfehlen sich für die symmetrische Verschlüsselung statt openssl eher die Programme GnuPG (ist in (Mail-)Anwendungen eingebunden) oder VeraCrypt (beherrscht Festplattenverschlüsselung unter verschiedenen Betriebssystemen).

[10] Im Singular verwenden wir im Deutschen „Modul“ und im Plural „Moduli“. So macht es bspw. auch Wikipedia für „RSA-Modul“ und „RSA-Moduli“ in `https://de.wikipedia.org/wiki/RSA-Kryptosystem`.
Im Englischen wird als Einzahl „modulus“ und im Plural ebenfalls „moduli“ verwendet. Siehe `https://en.wikipedia.org/wiki/RSA_(cryptosystem)`.

[11] PKCS steht für „public-key cryptography standards“, siehe RFC 5208 (`https://datatracker.ietf.org/doc/html/rfc5208`).

[12] Object Identifier, siehe `https://www.ietf.org/rfc/rfc3447.txt`

[13] Kleine Schlüssellängen wie in Übungsaufgaben sind bei `openssl` nicht erlaubt. OpenSSL 1.1.1 erfordert wenigstens eine RSA-Schlüssellänge von 512 bit.

[14] `genrsa` hat auch eine Option, um als öffentlichen Exponenten die Zahl 3 statt 65537 zu nutzen. Und mit der Option `-primes` kann man einen sogenannten „multi-prime“ RSA-Schlüssel erstellen. Siehe die Originaldoku `https://www.openssl.org/docs/manmaster/man1/genrsa.html`.

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

Alternativ[15]:

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

Die beiden obigen Aufrufe sind äquivalent: `openssl genpkey -algorithm RSA` und `openssl genrsa` erzeugen beide eine Datei im Base64-codierten, plattformunabhängigen PEM-Format[16].

Man kann beim Erzeugen den privaten Schlüssel auch gleich AES-verschlüsselt ablegen. Die Passphrase (4 bis 1023 Zeichen) kann man entweder nach einem Prompt eingeben oder direkt auf der Kommandozeile:

a) Hier fragt `openssl` nach der Passphrase, die man zweimal eingeben muss:

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

b) Hier wird die Passphrase „test“ direkt auf der Kommandozeile angegeben (kein Prompt):

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

1.3.2 OpenSSL-Datei „Privater Schlüssel“

In der Mathematik wird es so dargestellt, dass der private RSA-Schlüssel genau aus dem Modul `n` und dem Entschlüsselungs-Exponenten `d` besteht. Das ist auch richtig so.

Aus Implementierungsgründen enthält die „privater Schlüssel“ genannte Datei in `openssl` aber alle relevanten Informationen, die bei der Schlüsselgenerierung anfielen: also nicht nur `n` und `d`, sondern auch die Werte des öffentlichen Schlüssels (`n` und den Verschlüsselungs-Exponenten `e`), die beiden Primzahlen `p` und `q`, die Länge des Moduls und drei weitere Werte, die man benutzt, um die Entschlüsselung zu beschleunigen.[17]

Die tatsächlichen Inhalte des privaten Schlüssels kann man sich mit dem folgenden Befehl ansehen:[18,19]

```
$ 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:
```

[15] Auf der Webseite von OpenSSL wird allerdings dazu geraten, diese Alternative **nicht** zu verwenden, siehe `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.”

[16] PEM steht für *privacy enhanced mail*, vgl. `https://datatracker.ietf.org/doc/html/rfc1421`.

[17] Weitere Informationen zu den zusätzlichen Exponenten finden sich im Absatz zum Chinesischen Restsatz im RSA-Artikel in der Wikipedia:

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

[18] Ergänzt man die Option `-noout`, erhält man dieselbe Ausgabe, aber ohne den base64-Vorspann zwischen `BEGIN PRIVATE KEY` und `END PRIVATE KEY`.

[19] Wie man die Werte als ganzzahlige Dezimalzahlen erhält, wird in Abschnitt 1.6 auf Seite 13 erläutert.

```
    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:
    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-Datei „Öffentlicher Schlüssel“

Bei der Schlüsselerzeugung erstellt `openssl` nur den privaten Schlüssel, den wir unter dem Dateinamen `privatekey.pem` gespeichert hatten. Dieser sogenannte private Schlüssel enthält bei `openssl` auch alle Informationen des öffentlichen Schlüssels. Deshalb kann man bei `openssl` (aber nicht generell bei RSA) den öffentlichen Schlüssel aus der Datei mit dem privaten Schlüssel „extrahieren“.[20]

Um das Schlüsselpaar zu vervollständigen, erzeugen wir nun aus dem privaten Schlüssel den öffentlichen Schlüssel (nur dieser wird dann verteilt). Für den öffentlichen Schlüssel ist ein eigener Name anzugeben.[21]

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

Mit dem folgenden Befehl kann man sich die Details des öffentlichen Schlüssels ansehen (er enthält nur die Länge des Moduls, dessen Wert in Hex-Darstellung und den Wert des öffentlichen Exponenten):[22]

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

Will man nur den Modul, kann man sich diesen so ausgeben lassen:

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

Mit derselben Option kann man sich den Modul auch aus dem privaten Schlüssel ausgeben lassen:

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

---

[20] Beachte: Natürlich kann man aus dem öffentlichen Schlüssel auch bei `openssl` nicht den privaten Schlüssel herleiten.

[21] Es ist immer besser die interne Option `-out` zu nutzen als ein Redirect auf eine Datei zu machen:
`openssl pkey -in privatekey.pem -pubout > publickey.pem`
Alternativ geht der veraltete Befehl: `$ openssl rsa -in privatekey.pem -out publickey.pem -pubout`

[22] Wie man alle Werte eines Schlüssels als ganzzahlige Dezimalzahl erhält, wird in Abschnitt 1.6 auf Seite 13 erläutert. Im Einzelfall kann man Hexzahlen folgendermaßen in Dezimalzahlen umwandeln:

- in Python:
  `s="6a48f82d8e828ce82b82"; i = int(s, 16); s=str(i); i; s`
- auf der Kommandozeile mit dem BC-Tool:
  ```
  $ echo "ibase=16; 6A48F82D8E828CE82B82"|bc
  501916895863876199394178
  ```

1.4 Asymmetrische Ver- und Entschlüsselung mit openssl: noch keine hybride Verschlüsselung

Hier wird nun das in Abschnitt 1.3 erzeugte Schlüsselpaar genutzt, um eine Datei zu verschlüsseln.

1.4.1 Verschlüsselung mit RSA (kein Textbook-RSA)

Wir wollen mit `openssl` eine Datei `message.txt` verschlüsseln und benutzen dazu den öffentlichen Schlüssel `publickey.pem`. Die verschlüsselte Datei nennen wir `message.txt.rsaenc`.[23,24]

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

1.4.2 Entschlüsselung mit RSA (kein Textbook-RSA)

Wir wollen mit `openssl` die Datei `message.txt.rsaenc` entschlüsseln. Dazu brauchen wir den zugehörigen privaten Schlüssel aus `privatekey.pem`. Die entschlüsselte Datei nennen wir `message.decrypted.txt`.

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

1.5 Hybride Ver- und Entschlüsselung mit openssl

Unter hybrider Verschlüsselung versteht man eine Kombination aus asymmetrischer und symmetrischer Verschlüsselung. Dabei wählt der Sender einen zufälligen symmetrischen Schlüssel (genannt Sessionkey oder Sitzungsschlüssel). Mit diesem Sessionkey werden die zu schützenden (evtl. sehr langen) Daten symmetrisch (z. B. mit AES) verschlüsselt. Anschließend wird der Sessionkey asymmetrisch (z. B. mit RSA) mit dem öffentlichen Schlüssel des Empfängers verschlüsselt. Dieses Vorgehen löst das Schlüsselverteilungsproblem und erhält dabei den Geschwindigkeitsvorteil der symmetrischen Verschlüsselung. Siehe auch https://de.wikipedia.org/wiki/Hybride_Verschl%C3%BCsselung.

Varianten der hybriden Verschlüsselung werden bspw. bei sicherer Email und in HTTPS benutzt.

Vorbemerkung zur hybriden Ver- und Entschlüsselung

[23] Generell muss beim Verschlüsseln mit RSA die Länge von `message.txt` kleiner sein als die Länge des RSA-Moduls – sofern man die Message-Datei nicht in kleinere Teile aufteilt. `openssl` nutzt als Default jedoch RSA-OAEP statt Textbook-RSA, so dass der Input noch kleiner sein muss. OAEP steht für „Optimal Asymmetric Encryption Padding“ und soll laut RFC 8017 (https://datatracker.ietf.org/doc/html/rfc8017) verwendet werden, um den deterministischen RSA-Algorithmus um eine Zufallskomponente zu erweitern. Dadurch entsteht ein kleiner Overhead, der die maximal verschlüsselbare Nachrichtengröße, die bei Textbook-RSA möglich ist, um einige Bytes verringert. Genaueres findet man hierzu etwa auf `crypto.stackexchange.com` (https://crypto.stackexchange.com/questions/42097/what-is-the-maximum-size-of-the-plaintext-message-for-rsa-oaep. Dort steht u. a. folgender Post aus 2017 (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).“*
Beispiel: Hat der Modul des öffentlichen Schlüssels eine Länge von `2048bit`, kann man mit `openssl` eine Datei der Länge 245 B $=$ $(256-11)$B $=$ 1960 bit noch verschlüsseln, aber eine Datei der Länge $(256-10)$B $=$ 246 B $=$ 1968 bit schon nicht mehr. Im folgenden Codebeispiel wird einmal eine Zufallsnachricht von 245 B (genannt `x245`, per Default im Binärformat) und einmal eine von 246 B (genannt `y246`, auch binär) mit dem Befehl `rand` erzeugt und anschließend verschlüsselt bzw. es zumindest versucht:

```
$ 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:
```

[24] Wie man auch mit `openssl` das unsichere Textbook-RSA für didaktische Zwecke durchführen kann, wird in Abschnitt 1.9 beschrieben.

Alle Kommandos dieses Unterkapitels 1.5 sind in dem Shell-Skript `hybrid-openssl-enc-dec.sh` zum direkten Ausführen enthalten. Sein Listing (1.2) finden Sie am Ende dieses Abschnitts oder auf der Webseite: `https://www.cryptool.org/de/ctbook/openssl`

In unserem Szenario kommunizieren Alice und Bob.

1.5.1 Vorbereitung beim Empfänger Bob: RSA-Schlüsselgenerierung

Bob erzeugt sein Schlüsselpaar und speichert seinen privaten Schlüssel in einer Datei. Danach erzeugt er aus seinem privaten Schlüssel seinen öffentlichen Schlüssel:

```
$ 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 sendet seinen öffentlichen Schlüssel an Alice.

1.5.2 Verschlüsselung: Drei Aufgaben auf der Seite der Senderin Alice

1. Alice erzeugt einen zufälligen Sesssionkey und speichert ihn in der Datei `symkey.bin`.

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

2. Alice verschlüsselt den Sessionkey mit Hilfe von RSA unter Verwendung von Bobs öffentlichem Schlüssel.

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

3. Alice verschlüsselt die eigentliche Nachricht mit AES und `symkey.bin` als Sessionkey.

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

Alice sendet dann zwei Dinge an Bob:

- die mit dem Sessionkey symmetrisch verschlüsselte Nachricht als Datei „message.txt.enc“,
- den Sessionkey, asymmetrisch verschlüsselt mit Bobs öffentlichem Schlüssel, als Datei `symkey.bin.rsa-encrypted-for-bob`

Anschließend sollte Alice den Sessionkey löschen.

1.5.3 Entschlüsselung: Zwei Aufgaben auf der Seite des Empfängers Bob

1. Bob entschlüsselt den Sessionkey mit seinem privaten RSA-Schlüssel.

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

2. Bob entschlüsselt die eigentlichen Daten (Nachricht) mit 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 Alle Schlüssel einer privaten PEM-Datei als Dezimalzahl ausgeben – eigenes Python-Skript

Das mitgelieferte Skript `convert-pem-dec.py` konvertiert die gefundenen Hexzahlen in einer privaten oder öffentlichen PEM-Datei in Dezimalzahlen. Dazu durchsucht es die Textform der jeweiligen Schlüsseldatei nach den Suchstrings „RSA Private-Key“ und „RSA Public-Key“.

Bemerkung: Die von OpenSSL erzeugte private PEM-Datei wird bei OpenSSL oft auch „privater Schlüssel“ genannt. Siehe Unterkapitel 1.3.2.

Aufruf für die private PEM-Datei:

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

Analog geht es für die öffentliche PEM-Datei mit dem folgenden Aufruf:

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

Hier ein beispielhafter Output des öffentlichen Schlüssels (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 =25741008743014329602130075944926942393815840463882852482920558426679866327952824599118760618278259658850320492882235375131870650095604556532228132849189488301155048125268908387629473309159744005639267678283241886855049709291508947644006901365636919580674215699486371723448625643678604815254035894944521623776040635114105544615730756280024048599159857254792990232718018797025000784068035797549400275841333556147429868886438288336921556046961725006605043014256276199801354741104082121534386715731344850439112419539584983157934660693325340884464951058404325817167575799817003708324528796039188502047526399151614669321827
Exponent:
    e = 65537
```

1.7 Schlüssel einer PEM-Datei als Dezimalzahl ausgeben – per RsaCtfTool)

Das sogenannte „RSA-Angriffswerkzeug“ RsaCtfTool[25] wird hauptsächlich für CTFs[26] verwendet, um einen privaten Schlüssel aus einem schwachen öffentlichen Schlüssel zu gewinnen oder einen mit Textbook-RSA verschlüsselten Geheimtext zu entschlüsseln. Es kann auch verwendet werden, um die Parameter (Zahlen) eines Schlüssels im PEM- oder DER-Format auszulesen.

RsaCtfTool kennt auch das von OpenSSL benutzte PEM-Format zum Abspeichern von asymmetrischen Schlüsseln.

Die zwei Befehle von oben

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

entsprechen dem folgenden Befehl:

```
$ ./RsaCtfTool/RsaCtfTool.py --dumpkey --key privatekey.pem --private
```

1.8 Überblick über alle vorhergehenden OpenSSL-Befehle (als Liste und als Shell-Skript)

In den bisherigen Beispielen wurden ein zufälliger Sessionkey, ein zufälliger IV und RSA-Schlüsselpaare für Alice und Bob erzeugt. Dann wurde damit eine verschlüsselte Datei von Alice für Bob erzeugt (einmal wurde nur RSA verwendet, einmal die Hybridverschlüsselung). Zum Schluss wurden die Werte der Parameter in den PEM-Dateien in hex und dezimal ausgegeben. Dies ist zusammengefasst in den folgenden beiden Beispielen: Das Skript im OpenSSL-Beispiel 1.2 macht noch etwas mehr als die Befehlsliste im OpenSSL-Beispiel 1.1.

OpenSSL-Beispiel 1.1: Die bisherigen Tasks als Aufruf-Folge

```
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-Beispiel 1.2: Shell-Skript `hybrid-openssl-enc-dec.sh`

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

echo
openssl version
```

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

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

Fortsetzung OpenSSL-Beispiel 1.2

```
# 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
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)"
```

Fortsetzung OpenSSL-Beispiel 1.2

```
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 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
```

Fortsetzung OpenSSL-Beispiel 1.2

```
-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 mit `openssl` und eigenem Python-Skript

Vorbereitung

Im Weiteren nutzen wir das Python-Skript `int2bin.py`. Dieses wandelt eine natürliche Zahl in das binäre Format um, das openssl für eine bestimmte Länge benutzt. Die Länge muss dieselbe sein wie die des RSA-Moduls im benutzten Public Key. Der folgende Befehl zeigt die Bitlänge (key size) des Moduls:

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

Schritt 1: Ganze Zahl in `openssl`-Format umwandeln

Der folgende Aufruf speichert die Zahl `m=25` in derselben Länge wie der zu benutzende RSA-Modul (4096 bit) und zwar in der Datei „message.bin“ (deren Länge in Byte (512) sind genau 4096 bit):

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

Schritt 2: Die ganze Zahl mit `openssl` verschlüsseln

Damit das Verschlüsselungs-Ergebnis die pure Zahl ohne Padding ist, braucht man die Option `-raw`. Das Ergebnis wird in der Datei „message.bin.enc“ abgespeichert.

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

Schritt 3: Die verschlüsselte Zahl in eine ganze Zahl verwandeln

Das Python-Skript `int2bin.py` konvertiert das Ergebnis von `openssl` zurück in eine ganze Zahl `c`:

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

Schritt 4: Die verschlüsselte Zahl mit `openssl` wieder entschlüsseln

Das Ergebnis von `openssl` ist die entschlüsselte Zahl. Sie wird in der Datei „message.bin.enc.dec“ abgespeichert.

```
$ 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
```

Schritt 5: Die entschlüsselte Zahl in eine ganze Zahl verwandeln

Das Python-Skript `int2bin.py` konvertiert das Ergebnis von `openssl` zurück in eine ganze Zahl `m`:

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

Schritt 6: Nachvollziehen des Ganzen mit der Python-Funktion `pow()`

Mit Hilfe von `openssl` und dem Python-Skript `convert-pem-dec.py` holen wir uns die RSA-Parameter n, e und d aus dem privaten Schlüssel.

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

Man könnte dann in Python die Potenzierung wie gewohnt vornehmen:

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

Stattdessen sollte man jedoch unbedingt die `pow()`-Funktion nutzen, sonst dauert zumindest die Entschlüsselungs-Operation bei großen Zahlen viel zu lange oder bricht ab.[27]

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

Bemerkung

Alle Kommandos dieses Unterkapitels 1.9 finden sich in dem Shell-Skript OpenSSL-Beispiel 1.3. Dieses Skript ist verfügbar unter `https://www.cryptool.org/de/ctbook/openssl`.

OpenSSL-Beispiel 1.3: Textbook-RSA mit `openssl` und Python-Skript (ausführlich kommentiert 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
```

[27] a) SageMath erwartet ^ statt ** als Potenzierungsoperator.
b) SageMath meldet beim Entschlüsseln ohne `pow()`:
`exponent must be at most 9223372036854775807`
c) Schnell geht die Entschlüsselung nur, wenn man von vornherein sagt, man rechnet in einem endlichen Integerring, was auch per `pow()` geht: Dann steht das Ergebnis sowohl in Python als auch in Sage sofort da: 25 (in Python gibt er `25L` aus)
d) `pow()` ist so schnell, dass bei einem 4096-bit n auf der Kommandozeile noch kein Bedarf besteht, das CRT zu verwenden. Siehe `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).“

1.10 Zufallswerte erzeugen

Hier einige Befehle, um Zufallsdaten in verschiedener Länge und Form zu erzeugen:

1) 32 B = 256 bit erzeugen und in eine Datei namens `testkey.bin` schreiben:

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

2) 100 zufällige Bytes in Hex-Form erzeugen und in eine Datei namens `file.hex` abspeichern:

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

3) 100 zufällige Bytes in Base64-Form erzeugen und in eine Datei namens `file.b64` abspeichern:

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

Die Dateien sind in Byte unterschiedlich lang:

```
$ 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 Primzahlen mit OpenSSL erzeugen

OpenSSL wird produktiv benutzt, um große Primzahlen zu erzeugen. Für eine 4096-bit lange Primzahl benötigt ein moderner PC unter einer Sekunde.[28]

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

Auch kleinere Primzahlen lassen sich generieren, aber es werden nicht alle Primzahlen der jeweiligen Bitlänge ausgegeben (bei 2, 3 und 4 bit wird immer nur ein und dieselbe Primzahl erzeugt).[29]

```
$ 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, ...
```

OpenSSL kann nicht nur pseudozufällige Primzahlen generieren, sondern auch überprüfen, ob eine gegebene Zahl prim ist. Die Ausgabe zeigt die eingegebene Zahl zuerst in hex, dann dezimal:

---

[28] Will man die Ausgabe nicht dezimal, sondern hexadezimal, hängt man die Option `-hex` an:
`openssl prime -generate -hex -bits 200`
Mit der Option `-safe` kann man sich auch sogenannte sichere (englisch „safe“) oder starke Primzahlen erzeugen lassen. Diese braucht man nicht unbedingt für RSA, aber für den Diffie-Hellman-Schlüsselaustausch.
Die Liste der Optionen findet man in der Originaldoku: https://www.openssl.org/docs/manmaster/man1/openssl-prime.html

[29] Warum unter OpenSSL 1.1.1 die Primzahlen 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, ... nie angeboten wurden, ist mir noch nicht klar.

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

OpenSSL-Beispiel 1.4: Zufall und Primzahlen erzeugen (siehe auch 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
```

1.12 Geschwindigkeitsvergleiche mit `openssl`

Mit `OpenSSL` kann man die Geschwindigkeit der Verschlüsselungsverfahren auf dem PC testen und vergleichen.

Testen einer einzelnen Chiffre[30]

```
$ 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
...
```

Das bedeutet: Das jeweilige Verfahren wurde drei Sekunden lang in einem Loop auf je einen 16 B bzw. 64 B etc. langen Input angewandt. Beim Hashverfahren `md4` für den Fall 16 B beispielsweise wurden in 3 s über 24 Millionen Iterationen durchgeführt. Am Ende der Konsolenausgabe (hier oben weggelassen, denn sie ist recht lang) wird noch eine Liste ausgegeben, in der steht, wie viele Bytes hierbei verarbeitet wurden, beim Hashverfahren `md4` für den Fall 16 B waren das über 384 Millionen Bytes in 3 s, pro Sekunde also über 128 Millionen Bytes.

Testen aller unterstützten Verfahren

Der Gesamttest aller Funktionen dauerte auf einem modernen PC rund 20 Minuten. Dies leistet der Aufruf `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
```

[30] Der Blockmodus GCM von AES ist nur über den EVP-Namen (hier bspw. „aes-256-gcm“) zugänglich (EVP = „Envelope“ = „the Digital EnVeloPe library“, ein high-level Interface zu den Krypto-Funktionen von OpenSSL für Programmierer).

```
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
...
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 Zertifikate abrufen und evaluieren

Hier noch ein paar Einzelbefehle, um Protokolldetails verbundener Server abzurufen.

Parameter von Webserver-Zertifikaten von Google

Der folgende `openssl`-Befehl nutzt das Argument `s_client`, um eine sichere Verbindung zu `https://www.google.com` aufzubauen und Details zur Verbindung auszugeben[31]: Port 443 ist der Standard bei Webservern für HTTPS-Verbindungen (der Standard-Port für HTTP ist 80). Wenn die Verbindung erfolgreich ist, werden mehrere Zertifikate von Google angezeigt zusammen mit Infos zur sicheren Verbindung und zur benutzten

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

Cipher-Suite. Wenn Sie den Befehl jetzt neu aufrufen, erhalten Sie die aktuellen Google-Zertifikate, die sich von den hier angezeigten „leicht“ unterscheiden werden.

```
$ 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
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
```

Eine gut verständliche Beschreibung, was der Output bedeutet, findet sich bspw. auf der Webseite von Marty Kalin, woher auch das obige Beispiel stammt: https://opensource.com/article/19/6/cryptography-basics-openssl-part-1

Mit dem folgenden Kommando sieht man die Zertifikatskette und erhält die beiden Nicht-Root-Zertifikate im Base64-Format (den Befehl mit `Enter` abschließen): 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
```

Man kann sich ein Server-Zertifikat auch herunterladen, alles Überflüssige abschneiden und als Datei im PEM-Format abspeichern. Eine Option zur Ausgabe nur der Algorithmen gibt es offenbar nicht. Mit der Option `-dates` oder `-enddate` kann man das Expiration-Date des Zertifikats aus der erzeugten PEM-Datei bekommen. Sie können sehen, dass das Google-Zertifikat einen Gültigkeits-Zeitraum von rund 3 Monaten hat.

```
$ 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
```

Eine gut verständliche Beschreibung, wie man bspw. die unterstützten Cipher-Suites eines Servers checken kann, finden sich in dem Buch-Kapitel: `https://www.feistyduck.com/library/openssl-cookbook/online/testing-with-openssl/index.html`

Automatisches Extrahieren der Einzel-Zertifikate aus einem Organisations-Zertifikat, um dann aus jedem Einzel-Zertifikat den Modul auszugeben

Auf folgende Weise könnte man sich den Modul eines Zertifikats ansehen, wenn RSA zum Signieren verwendet wird. Beim Zertifikat von Google vom 26.5.2020 (siehe oben) steht aber „Public Key Algorithm: id-ecPublicKey“ statt „Public Key Algorithm: rsaEncryption“, so dass die Option `-modulus` zu einem Fehler führt:

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

Im folgenden betrachten wir das Zertifikat von Let's Encrypt, das RSA statt ECDSA verwendet. Wir holen uns das Zertifikat mit `openssl s_client`, extrahieren daraus per `sed` nur die darin enthaltenen Einzel-Zertifikate (im Base64-Format) und schreiben m. H. von `awk` jedes Einzel-Zertifikat in eine eigene Datei (awk sucht die Zeilen, die mit dem Inhalt der `awk`-Außenvariable `RS` anfangen. Hat awk eine solche Zeile gefunden, schreibt es diese und alle folgenden Zeilen bis zum nächsten Vorkommen des Suchstrings in eine Datei mit dem Namen der Innenvariablen `FileName`). Am Ende gibt es noch aus, wieviele solcher `BEGIN CERT ... END CERT`-Konstrukte es fand.[32]

```
$ 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
```

[32] Man könnte die 3 Befehle auch direkt hintereinander (per Pipe statt mit Zwischendateien) ausführen. Der erste Befehl nutzte `openssl`; die Auswertung des Ergebnisses mit sed und awk wurde in der Bash-Shell vorgenommen.

```
$ 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
```

Da diese Let's Encrypt-Zertifikate RSA nutzen, kann man für jede Einzeldatei mittels `openssl` den Modul des Zertifikats-Owners (Subject Public Key Info) ausgeben.

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

Passen privater und öffentlicher Schlüssel zusammen

Will man prüfen, ob der öffentliche Schlüssel aus einem Zertifikat und der öffentliche Schlüssel aus der Datei mit dem privaten Schlüssel zusammen passen, kann man das mit den folgenden Befehlen machen:

```
$ 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
```

Das geht auch analog für ECDSA-Schlüssel.[33]

Testen der Verfügbarkeit und Abruf der benutzten Algorithmen des Mail-Servers der Uni Siegen

```
$ 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+UUNliIGZQad0/7skYx1oHqvhW4B6PRjbVLjkA/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
zjCBzAzBggrBgEFBQcwAYYnaHR0cDovL29jc3AucGNhLmRmbi5kZS9PQ1NQLVNl
cnZlci9PQ1NQMEkGCCsGAQUFBzAChj1odHRwOi8vY2RwMS5wY2EuZGZuLmRlL2Rm
```

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

```
bi1jYS1nbG9iYWwtZzIvcHViL2NhY2VydC9jYWNlcnQuY3J0MEkGCCsGAQUFBzAC
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/0fJHpqSATXhDbzAF4KUu8NTtuyzFvWlj8Q3iLGx+S
fzSwCeYZNEEKI+WQO1gt6iJ1eCN7FXrQXCkRsYSh9Yyi0pVIj2EhvVeDe77NIN0b
IM6tU3qblo0VcaeO5pQ61fYQ+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
MEoGCCsGAQUFBzAChj5odHRwOi8vY2RwMi5wY2EuZGZuLmRlL2dsb2JhbC1yb290
LWcyLWNhL3B1Yi9jYWNlcnQvY2FjZXJ0LmNydDANBgkqhkiG9w0BAQsFAAOCAQEA
gXhFpE6kfw5V8Amxaj54zGglqRzzlZ4/8/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
2XjiVRKjO4dEthEL1QxSr2mMDwbf0KJTi1eYe8/9ByT0/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 wurde nach 3 Jahren Entwicklungszeit Anfang September 2021 offiziell freigegeben.[34] Ab Version 3 steht OpenSSL unter der Apache-2.0-Lizenz, was die Verwendung in anderen Open-Source-Projekten vereinfacht.

Im Rahmen einer Bachelorarbeit wurde ein Plugin für CrypTool-Online gebaut, das OpenSSL im Browser verfügbar machte: die OpenSSL-App. Dazu wurde zuerst OpenSSL 1.1.1 nach WebAssembly (wasm) portiert und mit dieser Erfahrung war es dann leicht, das dann freigegebene OpenSSL 3.0.0 nach WebAssembly zu bringen und noch im September 2021 zur Verfügung zu stellen.

Das ursprüngliche CTO-Plugin (Web-App) wurde dann um eine GUI erweitert[35] und ist erreichbar unter:

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

Hier können (fast) alle OpenSSL-Kommandozeilen-Befehle ausgeführt werden, sofern sie außer der Pipe (Datenstrom zwischen zwei Prozessen) keine Besonderheit der Bash-Shell und außer `echo` keine weitere Linux-Kommandos nutzen.[36] Somit sind mit der OpenSSL-App fast alle Befehle dieses Kapitels ad hoc ausführbar, ohne dass man sich OpenSSL installieren muss. Das dürfte besonders unterwegs oder für Windows- und MAC-Nutzer interessant sein, bei denen OpenSSL nicht vorinstalliert ist.

Zur Demonstration und zur Unterstützung der Lehre sind viele OpenSSL-Funktionen nicht nur im Terminal des CTO-OpenSSL-Plugins verfügbar, sondern auch direkt über die GUI in 6 Reitern – siehe auch Abb. 1:

1. Verschlüsseln
2. Schlüssel erzeugen
3. Signieren & Verifizieren
4. Zertifikate
5. Prüfsummen
6. Dateien

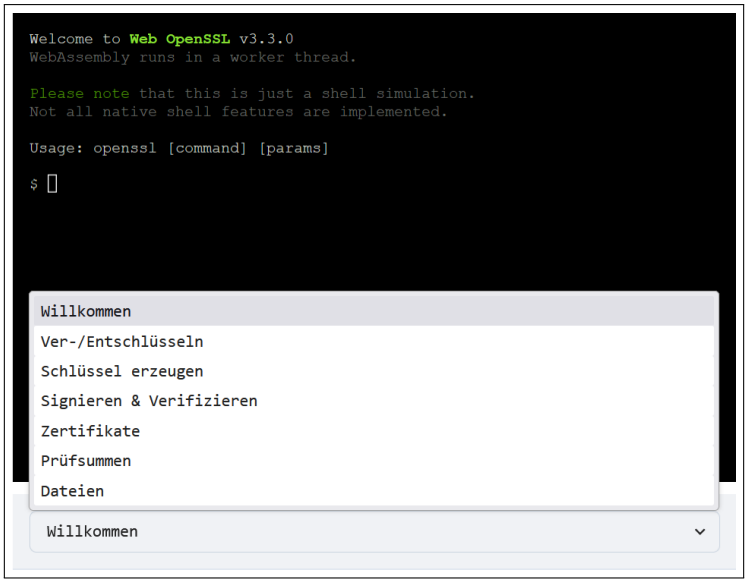

Abb. 1: In CTO: OpenSSL 3 im Browser: Die 6 Reiter in der GUI

[34] https://www.openssl.org/blog/blog/2021/09/07/OpenSSL3.Final/
[35] Das Webterminal darin läuft in einem eigenen Thread und das Plugin kann die Browser-Vollbild-Ansicht nutzen.
[36] Das ist der Stand Juni 2024. Diese Einschränkungen des Web-Terminals sollen nach und nach beseitigt werden.

Wenn man in der **GUI** eine Aktion ausführt, wird auf der **Kommandozeile** das entsprechende OpenSSL-Kommando synchron angezeigt, was zum Erlernen der OpenSSL-Befehle hilfreich ist.

Die folgenden drei Screenshots aus CTO (Abb. 2, 4 und 5) zeigen OpenSSL im Browser:

1. Abbildung 2 zeigt, wie ein Text mit dem symmetrischen Verfahren AES und Passwort verschlüsselt wird. Aus dem Passwort wird über die Schlüsselableitungsfunktion PBKDF2 der Salt, der Initialisierungsvektor (IV) und der symmetrische Schlüssel erzeugt. Alternativ könnten Salt und Initialisierungsvektor auch manuell eingegeben werden. Mit der zusätzlichen Option `-p` werden die von der Schlüsselableitungsfunktion erzeugten Hex-Werte für Salt, IV und Schlüssel auch ausgegeben. Die Option `-a` bewirkt, dass die Ausgabe Base64-codiert ist. Der Screenshot zeigt die Auswahl der entsprechenden Felder in der GUI, und wie der zugehörige OpenSSL-Befehl auf der Kommandozeile aussieht. Beispielsweise so:

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

   Die verschlüsselte Nachricht kann nur von jemand entschlüsselt werden, der den passenden Schlüssel oder das passende Passwort kennt.

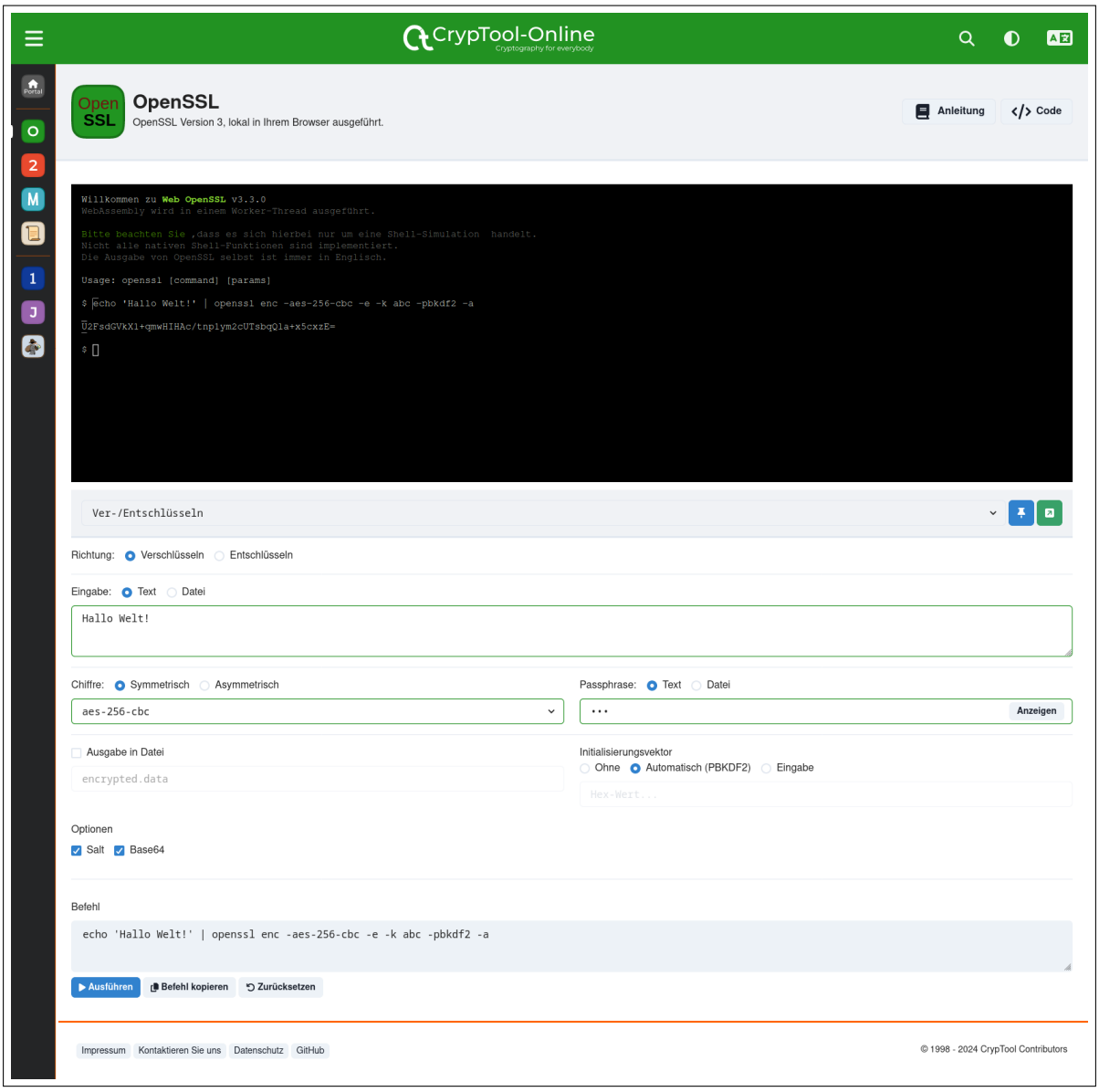

Abb. 2: In CTO: OpenSSL 3 im Browser: AES-Verschlüsselung

2. Abbildung 4 auf Seite 32 zeigt den Reiter „Zertifikate“. Die Senderin einer Nachricht (z. B. Alice) braucht ein Zertifikat, um es an mögliche Empfänger von Nachrichten zu verteilen, damit die Empfänger die Authentizität einer von ihr signierten Nachricht überprüfen können.

   Bevor wir uns den Ablauf in der CTO-App (Abb. 4) ansehen, zeigt Abb. 3 den Ablauf zwischen den Beteiligten in einem Diagramm. Hier wird das Zertifikat von einer dritten Instanz (Zertifizierungsstelle, engl. CA) signiert. Ein Zertifikat kann aber auch selbst-signiert erstellt werden.

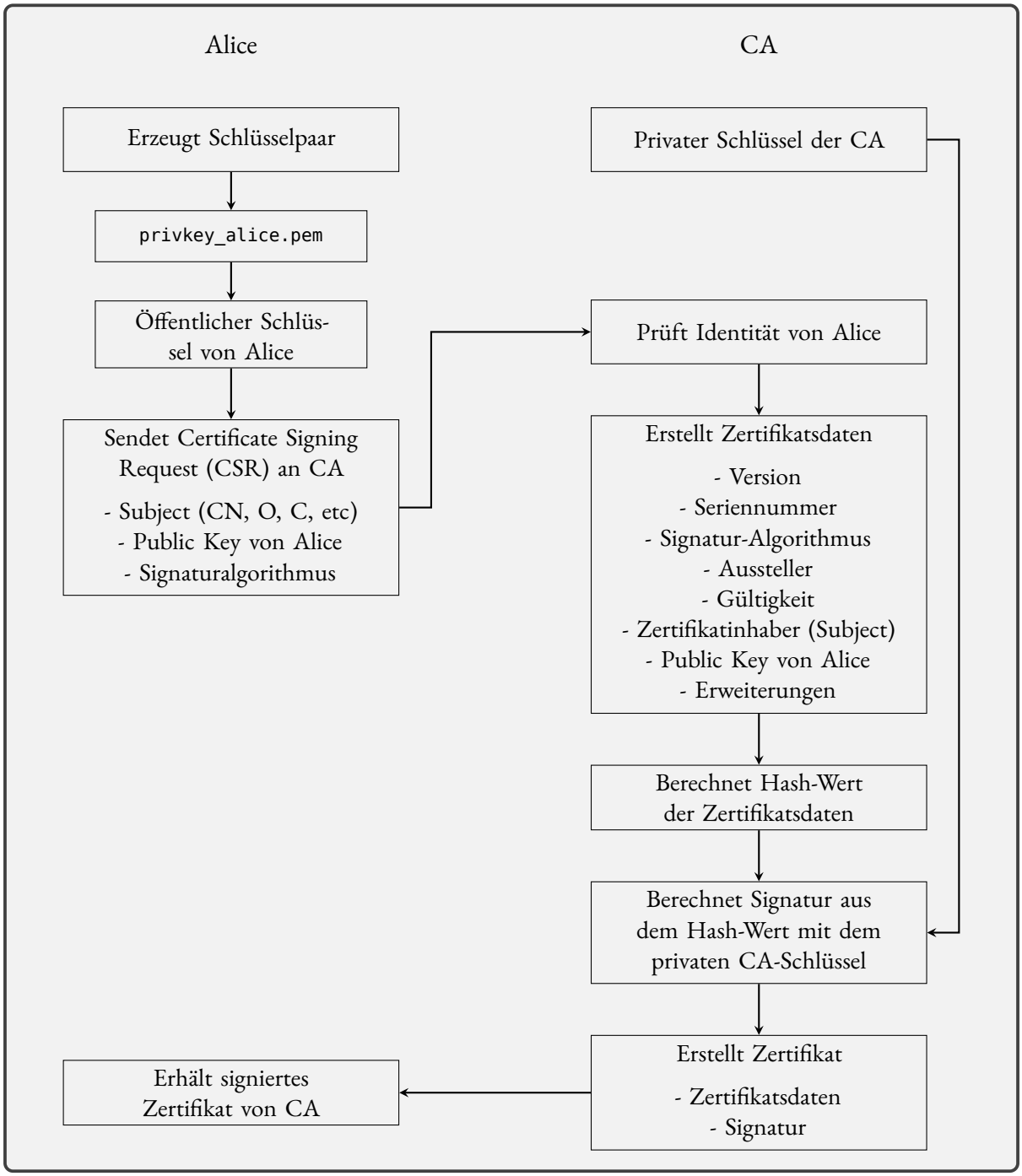

Abb. 3: Erstellung eines von einer CA signierten Zertifikats für Alice

Um ein Zertifikat zu erstellen sind zwei Dinge nötig: Man braucht einen privaten Schlüssel, um das Zertifikat zu unterschreiben (Signatur erstellen) und man braucht die Daten über den Zertifikatsinhaber (hier Alice) inclusive dessen öffentlichem Schlüssel. Ein fertiges Zertifikat besteht also aus dem Daten-Teil und dem Signatur-Teil.

In der GUI wurde als Variante ein selbst-signiertes Zertifikat ausgewählt, so dass beide Schlüssel vom Zertifikatsinhaber kommen können. Man kann ein bereits vorhandenes Schlüsselpaar nutzen oder ein neues Schlüsselpaar erzeugen. Bei OpenSSL ist in der Datei für den „privaten Schlüssel“ immer auch der zugehörige öffentliche Schlüssel enthalten. Im Screenshot wird ein RSA-Schlüsselpaar neu erzeugt, dessen Modulus-Länge 2048 bit beträgt. Für die Signatur wird der Datenteil gehasht, standardmäßig mit SHA-256. Man kann auch eine andere Hash-Funktion wählen, z. B. Keccak mit der Option `-sha3-256`. Die Gültigkeitsdauer des Zertifikats wird in das Zertifikat eingetragen. Das Passwort zum Verschlüsseln wird nur benutzt, wenn man ein Schlüsselpaar neu erzeugt.

OpenSSL führt die Erzeugung des Zertifikats in einem Schritt durch (ein einziger Befehl). Der Befehl erzeugt dann zwei Dinge: einen privaten Schlüssel und das Zertifikat. Je nach gewählter Option werden diese in eine Datei geschrieben oder im schwarzen Terminal auf der Kommandozeile ausgegeben.

Erzeugt man kein selbst-signiertes Zertifikat, schickt man die nötigen Daten in Form eines Certificate Signing Requests (CSR) an die CA. Das lässt sich mit den beiden anderen Varianten simulieren. Da man mit OpenSSL eine ganze PKI bauen kann, sind die Möglichkeiten bei weitem umfangreicher als das, was man in der GUI zusammenklicken kann.

Der Screenshot zeigt die Auswahl der entsprechenden Felder in der GUI, und wie der zugehörige OpenSSL-Befehl auf der Kommandozeile aussieht. Beispielsweise so:

```
openssl req -x509 -out cert.pem -days 30 -noenc -newkey rsa\:2048 -keyout private.key -subj '/C=DE/L=
 München/ST=Bayern/O=Beispiel GmbH/OU=IT-Abteilung/CN=www.beispiel.de/emailAddress=info@beispiel.de' -
 addext subjectAltName\=email\:info\@example.com -batch
```

Die Default-Werte für das Subject eines Zertifikats sind C=AU, ST=Some-State, O=Internet Widgits Pty Ltd. Mit dem folgenden Befehl kann man sich die Subject-Daten eines Zertifikats anzeigen zu lassen:

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

3. Abbildung 5 auf Seite 33 zeigt den Reiter „Dateien“, in dem eine Datei im virtuellen Dateinsystem zu sehen ist, die den privaten Schlüssel enthält, der zuvor per OpenSSL erzeugt wurde. Technisch wird dies mittels WebAssembly gemacht, das einen virtuellen Speicher im Browser anlegen kann, in dem die erzeugten Dateien abgelegt werden. Hierhin kann man auch Eingabe-Dateien hochladen.

Jedes der 6 Tabs (siehe Abb. 1 auf Seite 28) kann auch auf der rechten Seite in einer Navigationsleiste angepinnt werden (siehe Abb. 6 auf Seite 34). Sinnvoll ist das, wenn man sich in der OpenSSL-App zwei Tabs gleichzeitig anzeigen lassen möchte. So kann man den Funktionsumfang von zwei Tabs gleichzeitig sehen und nutzen. Beispielsweise beim Arbeiten mit Dateien kann das Anpinnen des Dateien-Tabs Vorteile bringen: Man sieht direkt, ob eine Datei erstellt wurde, oder kann benötigte Dateien, die man verschlüsseln, entschlüsseln oder signieren will, hochladen und verwalten. Abb. 6 zeigt, wie links eine Datei verschlüsselt wird. Als Ergebnis erhält man eine Datei, die rechts auch gleich zu sehen ist.

Hinweis: Die folgende YouTube-Playlist von Nils Kopals bietet ab Q4/2024 einen umfassenden OpenSSL-Kurs mit CrypTool-Online für Einsteiger und Fortgeschrittene:

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

Somit kann man mit diesem CTO-Plugin auch als Anfänger die OpenSSL-Befehle kennen lernen und OpenSSL sogar auf dem Smartphone nutzen.

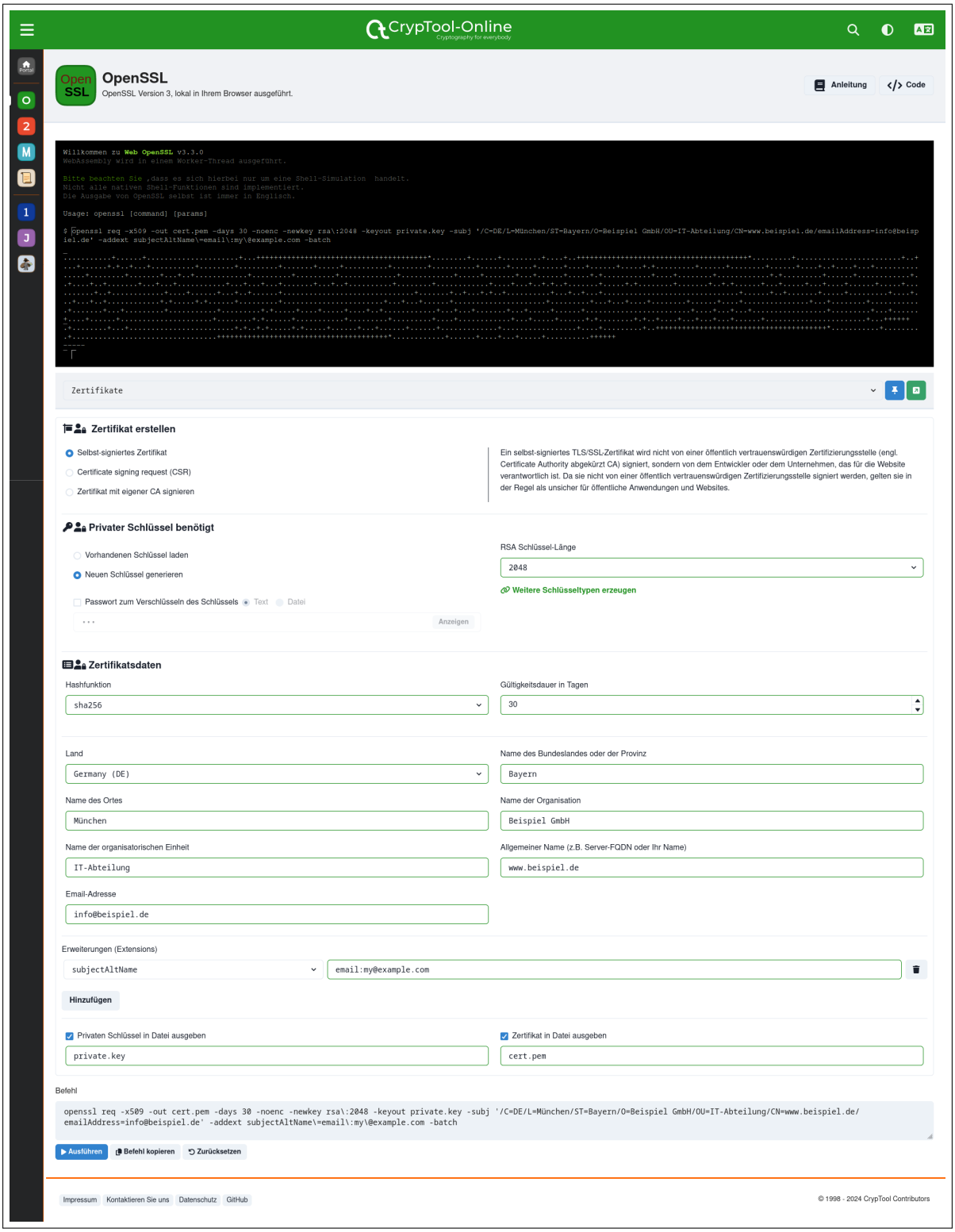

Abb. 4: In CTO: OpenSSL 3 im Browser: Erzeugung eines digitalen Zertifikats

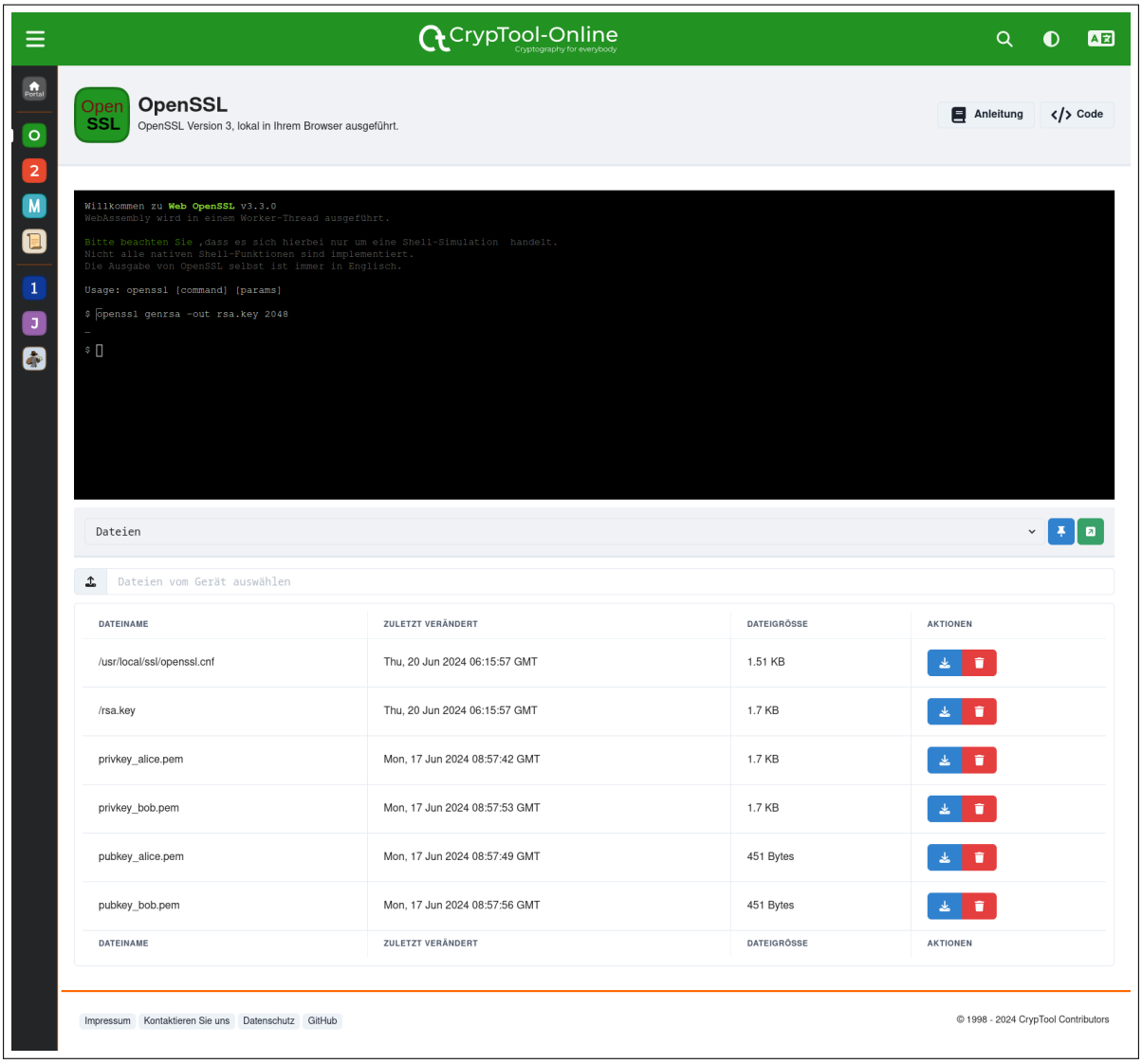

Abb. 5: In CTO: OpenSSL 3 im Browser: Dateien im virtuellen Browser-Dateisystem

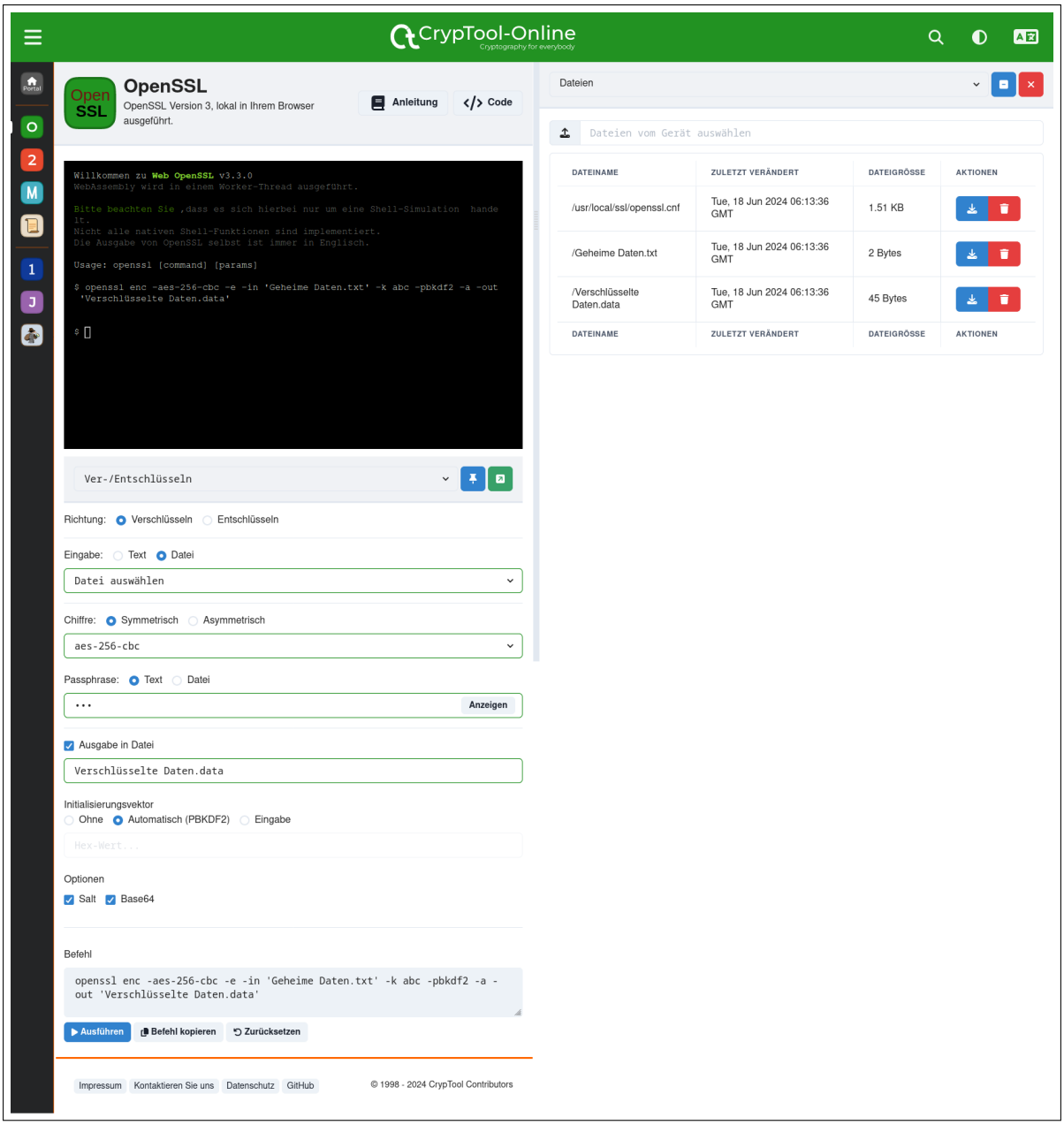

Abb. 6: In CTO: OpenSSL 3 im Browser: Das Dateien-Tab ist rechts angepinnt

1.15 Web-Links zu diesem Anhang 1

1. *OpenSSL for Web (via WebAssembly im Browser)*
   https://www.cryptool.org/de/cto/openssl
   https://wiki.openssl.org/index.php/Binaries
2. *OpenSSL – Cryptography and SSL/TLS Toolkit*, 2020, Primäre Quelle
   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, Gutes Tutorial von 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, Gutes Tutorial von Cody Arsenault
   https://www.keycdn.com/blog/openssl-tutorial
6. *An Introduction to the OpenSSL command line tool*, 2004, Tutorial von Philippe Camacho
   https://users.dcc.uchile.cl/~pcamacho/tutorial/crypto/openssl/openssl_intro.html
7. *OpenSSL Command-Line HOWTO*, 2016, rezeptartiges Tutorial von Paul Heinlein
   https://www.madboa.com/geek/openssl/
8. Wikipedia: *RSA-Kryptosystem*
   https://de.wikipedia.org/wiki/RSA-Kryptosystem#RSA_mit_dem_Chinesischen_Restsatz
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*, 3. Aufl., 2022, Ivan Ristić
    https://www.feistyduck.com/library/openssl-cookbook/online/

Alle Links wurden am 5.6.2024 überprüft.

Die `openssl`-Beispiele aus diesem Buch samt den zugehörigen Shell- und Python-Skripten finden sich auch auf der CrypTool-Webseite. Weitere Details in der Übersicht auf Seite 36.

2 Verzeichnisse der Abbildungen, Tabellen, Code-Beispiele, etc.

2.1 Abbildungsverzeichnis

1 In CTO: OpenSSL 3 im Browser: Die 6 Reiter in der GUI . . . 28
2 In CTO: OpenSSL 3 im Browser: AES-Verschlüsselung . . . 29
3 Erstellung eines von einer CA signierten Zertifikats für Alice . . . 30
4 In CTO: OpenSSL 3 im Browser: Erzeugung eines digitalen Zertifikats . . . 32
5 In CTO: OpenSSL 3 im Browser: Dateien im virtuellen Browser-Dateisystem . . . 33
6 In CTO: OpenSSL 3 im Browser: Das Dateien-Tab ist rechts angepinnt . . . 34

2.2 Verzeichnis der OpenSSL-Programmbeispiele

1.1 Tasks als Aufruf-Folge . . . 14
1.2 Shell-Skript `hybrid-openssl-enc-dec.sh` . . . 14
1.3 Textbook-RSA . . . 18
1.4 Zufall und Primzahlen erzeugen . . . 20

Die Einführung in das CLI `openssl` im Abschnitt 1 auf Seite 5 verwendet die folgenden zwei Bash-Shell-Skripte und die folgenden zwei Python-Skripte:

- `hybrid-openssl-enc-dec.sh` mit Befehlen aus A.1.2–A.1.6, A.1.8 und A.1.9; ruft
  - `convert-pem-dec.py`
- `zahl-direkt-openssl-enc-dec.sh` mit Befehlen aus A.1.7; ruft
  - `int2bin.py`

Diese 4 Skripte mit den `openssl`-Beispielen aus diesem Buch finden Sie auf der CrypTool-Webseite:
`https://www.cryptool.org/de/ctbook/openssl`

Alle Beispiele wurden mit den OpenSSL-Versionen 1.1.1l und 3.3.0 getestet.

3 Index

C
CTF Capture-the-flag, 14
CTO, 29

G
GnuPG, 5

K
KDF, 7
Keccak, 31

M
Modul, 8

O
OAEP, 11
OpenSSL, 5
    Beispiele, 36
    Programmbeispiele, 5

P
PKCS, 8, 11

R
RSA, 11, 31
    multi-prime, 8
    OAEP, 11
RsaCtfTool, 14

S
Sessionkey, 7
SHA3, 31

T
Textbook-RSA, 11, 17, 18

Z
Zertifikat
    selbst-signiert, 31