Institut für Softwaretechnologie

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Erstellung von CryptoExamples in Python

Manuel Kloppenburg

Studiengang:	Softwaretechnik
Prüfer/in:	Prof. Dr. Stefan Wagner
Betreuer/in:	Kai Mindermann, M.Sc.
Beginn am:	20. Juni 2018
Beendet am:	21. September 2018

Kurzfassung

Kontext: Python ist eine sehr verbreitete und leicht zu erlernende Programmiersprache. Die Verwendung von kryptographischen Programmbibliotheken ist aber selbst in Python oft nicht einfach. Außerdem gibt es viele Beispiele im Internet, die veraltet und nicht sicher sind. Gerade für Entwickler, die keine Erfahrung in der Kryptographie haben, stellt es eine Schwierigkeit dar, kryptographisch sichere Software zu programmieren. *Ziel*: Es sollen Code-Beispiele für die Open Source Plattform CryptoExamples in der Programmiersprache Python erstellt werden. Diese Code-Beispiele haben die Anforderung sicher, minimal, vollständig, kopierbar, ausführbar und getestet zu sein, und helfen dadurch Entwicklern, sicheren Code zu schreiben. *Methode*: Es wurden generelle Richtlinien für das Erstellen und die Wartung von Code-Beispielen für CryptoExamples definiert. In den Richtlinien wird auf das Bundesamt für Sicherheit in der Informationstechnik (BSI) und das National Institute of Standards & Technology (NIST) der Vereinigten Staaten verwiesen, um sichere Algorithmen und Parameter auszuwählen. Auf Basis dieser Richtlinien wurde ein Umsatzkonzept für die Programmiersprache Python geschaffen und davon ausgehend Code-Beispiele entwickelt, die den Anforderungen gerecht werden. Für die Code-Beispiele wurde die kryptographische Programmbibliothek `cryptography.io` verwendet. *Ergebnisse*: Code-Beispiele für symmetrische Verschlüsselung, asymmetrische Verschlüsselung, digitale Signatur, Speichern von Schlüsselpaaren und Hashing sind entstanden. Die Code-Beispiele wurden auf der Open Source Plattform GitHub veröffentlicht und stehen somit der Öffentlichkeit zur Verfügung. Durch Statische Code-Analyse und automatische Tests ist die Code Qualität und Korrektheit gewährleistet. Außerdem werden die Code-Beispiele durch automatische Prüfung auf Konformität mit den Richtlinien überprüft. *Fazit*: Die erstellten Code-Beispiele helfen, sichereren Code zu schreiben. Sie müssen aber noch von Experten begutachtet werden, und es besteht die Möglichkeit, noch weitere Beispiele zu verwirklichen. Außerdem muss die Plattform CryptoExamples eine größere Reichweite erlangen, sodass jeder Entwickler diese Beispiele und keine anderen sieht.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Listings

Abkürzungsverzeichnis

AES Advanced Encryption Standard. 18

BSI Bundesamt für Sicherheit in der Informationstechnik. 3, 27

CBC Cipher-Block Chaining. 31

CMAC Cipher-based Message Authentication Code. 32

CTR Counter Mode. 31

DCIES Discrete Logarithm Integrated Encryption Scheme. 32

DER Distinguished Encoding Rules. 43

DSA Digital Signature Algorithm. 32

ECDSA Elliptic Curve Digital Signature Algorithm. 32

ECIES Elliptic Curve Integrated Encryption Scheme. 32

GCM Galois/Counter Mode. 31

GMAC Galois Message Authentication Code. 32

HMAC Keyed-Hash Message Authentication Code. 32

KDF Key Derivation Function. 24

MAC Message Authentication Code. 22

MGF Mask Generation Function. 32

NIST National Institute of Standards & Technology. 3, 24

OAEP Optimal Asymmetric Encryption Padding. 32

PBKDF2 Password-Based Key Derivation Function 2. 32

PEM Privacy Enhanced Mail. 43

PKCS Public-Key Cryptography Standards. 24

PSS Probabilistic Signature Scheme. 33

RSA Rivest Shamir Adleman. 23

SHA Secure Hash Algorithm. 33

TDEA Triple Data Encryption Algorithm. 31

1 Einleitung

Kryptographie ist überall. Sobald wir eine Website aufrufen, die mit „https://“ beginnt, oder eine Nachricht per WhatsApp verschicken, lösen wir kryptographische Prozesse aus, die unseren Internet-Verkehr oder unsere Nachrichten verschlüsseln. Die Tatsache, dass immer mehr Leute auf ihre Privatsphäre Wert legen und ihre Daten in Gefahr sehen, unterstützt den Trend, alles zu verschlüsseln und kryptographische Prozesse zu nutzen. Doch wer setzt diese Prozesse und Algorithmen um, um unsere Daten zu sichern? Es sind Entwickler, die sich oft nicht mit Kryptographie auskennen, die die Software der Apps, Programme und Webseiten schreiben.

1.1 Motivation

An sich werden kryptographische Algorithmen und ihre mathematischen Grundlagen von Kryptologen, Leuten die schon sehr viel Erfahrung in dem Gebiet haben, entwickelt und von anderen Kryptologen hinterfragt und auf ihre Sicherheit überprüft. Das heißt aber nicht, dass alle kryptographischen Algorithmen perfekt und ohne Fehler sind. Und wie später darauf hinweisen wird, ist es wichtig, ein aktuelles und sicheres Verfahren zu wählen und notfalls auch zu ändern, sollten Sicherheitslücken auftreten, oder sicherere und schnellere Verfahren entwickelt werden. Jedoch soll es in dieser Arbeit nicht primär darum gehen, welches Verfahren das Beste ist, sondern es geht um Entwickler, die diese Verfahren nutzen, um Software zu programmieren.

Nehmen wir einen Entwickler, der einen Dienst zum Verschicken von Nachrichten programmieren will. Vermutlich ist er Experte in der Programmiersprache, die er benutzt und kennt sich vielleicht sogar mit Netzwerkkommunikation aus. Wenn er jedoch dazu kommt, die Verschlüsselung der Nachrichten zu implementieren, stößt er an seine Grenzen und zieht das Internet zu Rat. Er ist kein Experte in der Kryptographie und wählt sich eine kryptographische Programmierbibliothek aus. Diese sind kompliziert, haben unsichere vorgegebene Werte oder sind schlecht dokumentiert [1]. Da der Entwickler nicht weiß, wie er fortfahren soll, führt das meistens zu einer Suche auf Google und damit zu einem Blogpost von vor sechs Jahren oder einer Frage auf Stackoverflow[1] oder ähnlichen Plattformen. Dort werden Beispiele von Nutzern eingestellt. Acar et al. [2] fanden heraus, dass nur 17% solcher Code-Beispiele sicher sind. Der Entwickler verwendet nun in den meisten Fällen unwissentlich ein unsicheres Beispiel. Das ist ihm aber auch egal, solange es funktioniert. So eine Vorgehensweise führt zu unsicherer Software, kann aber nur schwer verhindert werden.

Das ist ein Problem und soll mit CryptoExamples [22] und damit auch mit dieser Arbeit zum Thema „Erstellung von CryptoExamples in Python“ angegangen werden. Es finden sich im Web meist nur veraltete, unsichere und nicht direkt ausführbare Beispiele und selbst in den Dokumentationen der Kryptographie Bibliotheken finden sich oft keine vollständigen Beispiele. „With CryptoExamples

[1] `https://stackoverflow.com/` (abgerufen 2018-09-19)

the gap between hard to change API documentation and the need for complete and secure code examples can be closed“ [22] (deutsch: Mit CryptoExamples kann die Lücke zwischen schwer veränderbaren Dokumentationen von Programmierschnittstellen und der Notwendigkeit für komplette und sichere Code-Beispiele geschlossen werden). Die Plattform CryptoExamples[2] hat das Ziel, solche Beispiele zur Verfügung zu stellen. Dabei ist wichtig, dass die Beispiele sicher, minimal, vollständig, kopierbar, ausführbar und getestet sind. Warum die Beispiele sicher sein sollten ist offensichtlich. Es geht darum Code-Beispiele zur Verfügung zu stellen, die Entwickler nutzen können, um sichere kryptographische Prozesse zu implementieren. Minimal sollen die Beispiele sein, damit der Benutzer nicht riesige Mengen an Code übernehmen muss. Dieser würde den eigenen Code des Entwicklers unübersichtlich machen und in vielen Fällen braucht der Entwickler gar nicht alle Funktionen, die es gibt. Vollständig sollten die Beispiele aber trotzdem sein. Dem Benutzer sollte der komplette kryptographische Prozess zur Verfügung stehen. Es hilft ihm nicht, wenn er seine Nachrichten nur verschlüsseln, aber nicht wieder entschlüsseln kann. Das Ziel ist es, dass der Nutzer den Beispiel-Code kopieren und genau so für sein eigenes Projekt übernehmen kann. Deshalb ist das Stichwort kopierbar wichtig. Die Beispiele sollten auch ausführbar sein, damit der Nutzer sie direkt verwenden kann und nicht erst Änderungen daran vornehmen muss. Zuletzt sollten die Beispiele getestet sein. Wenn sie nicht funktionieren, sind die Beispiele keine Hilfe sondern sabotieren den Code des Nutzers.

Nun ist klar, warum die Plattform CryptoExamples benötigt wird. Warum spielt jedoch Python eine entscheidende Rolle und warum sollten solche Beispiele auch in Python existieren? Python ist eine Programmiersprache, die sehr verbreitet ist und in den verschiedensten Bereichen verwendet wird. Außerdem gibt es für Python eine große Zahl an Bibliotheken. Python ist auch eine sehr einfache Sprache. Sie ist leicht zu lesen und auch zu schreiben. Sie wird von Einsteigern gerne beim Erlernen vom Programmieren benutzt, aber auch unter erfahrenen Entwicklern ist sie weit verbreitet [1]. Außerdem findet sie in vielen oft genutzten Programmen Verwendung und wird auch von großen Firmen benutzt [25]. 2014 war Python, nach aktiven Repositories gemessen, die dritt verbreitetste Sprache auf GitHub [18]. Und in dem Jahresrückblick von GitHub selbst [35] wird sie als zweit populärste Sprache genannt, gemessen an geöffneten *Pull Requests*. Die Daten von GitHub repräsentieren natürlich nur den Open Source Teil und keinen kommerziellen Code. Stackoverflow führt jedes Jahr eine Umfrage unter Entwicklern durch. In dieser Umfrage findet sich Python unter den populärsten Sprachen auf Rang sieben wieder. Bei der Frage nach der am meisten geliebten Sprache kann Python sich sogar mit dem ersten Platz rühmen [12]. Außerdem ist laut Stackoverflow Python die am schnellsten wachsende Programmiersprache, wenn man auf die bedeutenden Programmiersprachen schaut [33]. Auch auf dem *TIOBE Index* [36] ist Python mit Platz drei sehr hoch platziert. Außerdem ist Python auch bei diesem Index als eine wachsende Programmiersprache zu erkennen. Der *TIOBE Index* misst die Popularität von Programmiersprachen anhand von Suchmaschinen. Ein weiterer Index zum Messen von Popularität ist der *PYPL PopularitY of Programming Language Index* [6]. Dieser misst, wie oft Anleitungen zu den Programmiersprachen auf Suchmaschinen gesucht werden. Das gibt zum Einen Aufschluss darüber wie populär die Sprache ist, wie schon der Name des Index verrät. Zum Anderen werden Anleitungen meist angeschaut, wenn Personen eine Programmiersprache lernen. Der Index gibt also auch darüber Aufschluss, dass Python eine wachsende Sprache ist oder zumindest mehr Personen diese erlernen wollen. Diese

[2] `https://www.cryptoexamples.com/` (abgerufen 2018-09-19)

Punkte motivieren die Erstellung von CryptoExamples in Python. Vor allem da es eine wachsende Sprache ist und CryptoExamples für Entwickler sind, die neue Software programmieren und Hilfe mit Kryptographie brauchen, ist Python wichtig.

Python ist aber auch speziell für die Kryptographie eine wichtige Sprache. Laut einer Liste, welche auf Statistiken der „submission challenge“ von Matasano Security basiert, ist Python die favorisierte Sprache wenn es um Kryptographie geht. Python ist mit 43% die am meisten genutzte Sprache. Golang folgt auf dem zweiten Platz mit 10,72% [26]. Python ist also mit Abstand die wichtigste Sprache für Kryptographie.

1.2 Verwandte Arbeiten

Es gibt wenige Arbeiten, die das gleiche Ziel verfolgen und Beispiele für die korrekte und sichere Nutzung von kryptographischen Prozessen zur Verfügung stellen. Das et al. [11] beschreiben das fälschliche Gebrauchen von kryptographischen Bibliotheken. Dafür untersuchen sie verschiedene Sprachen und Bibliotheken und vergleichen diese auf Auffälligkeiten, die zum Missbrauch führen können. Die Motivation hinter dieser Arbeit ist ähnlich zu unserer. Während Das et al. jedoch Empfehlungen für Entwickler von kryptographischen Bibliotheken aussprechen, ist unser Ziel, die Entwickler zu erreichen, die diese nutzen.

Acar et al. [2] haben eine Studie durchgeführt, in der sie Entwickler sichere Software für Android programmieren ließen. Dabei hatten die Entwickler beschränkten Zugriff auf unterschiedliche Quellen, um Informationen zu erlangen. Zum Beispiel durfte eine Gruppe nur auf Stackoverflow zugreifen, währen eine andere Gruppe die offizielle Dokumentation von Android verwendete. „Those participants who were allowed to use only Stack Overflow produced significantly less secure code than those using, the official Android documentation or books [..]“ [2] (deutsch: Die Teilnehmer, die nur Stackoverflow verwenden durften haben signifikant unsichereren Code produziert, als die, die die offizielle Android Dokumentation oder Bücher verwendeten). Das Ergebnis der Arbeit ist, dass zum einen kryptographische Programmbibliotheken, wenn auch schwer zu verwenden, sicher sind. Auf der anderen Seite jedoch nicht offizielle Informationen, wie die auf Stackoverflow, besser zu erreichen sind, aber zu unsicherem Code führen. Acar et al. sprechen sich dafür aus Dokumentationen zu erstellen, die sicher, aber auch einfach zu nutzen sind.

Acar et al. [1] haben eine weitere Studie durchgeführt, welche die Nutzbarkeit von kryptographischen Programmbibliotheken untersucht. Sie ließen Entwickler kryptographische Aufgaben mit unterschiedlichen Programmbibliotheken programmieren. Die entstandenen Ergebnisse wurden unter anderem auf ihre Sicherheit überprüft und die Entwickler sollten die Programmbibliotheken einschätzen. „[..] while new cryptographic libraries [..] should offer a simple, convenient interface, this is not enough: they should also, and perhaps more importantly, ensure support for a broad range of common tasks and provide accessible documentation with secure, easy-to-use code examples“ [1] (deutsch: Während neue kryptographische Bibliotheken eine einfache, praktische Schnittstelle anbieten sollten, ist das nicht genug: Sie sollten auch, und eventuell wichtiger, eine breite Auswahl an geläufigen Aufgaben unterstützen und eine zugängliche Dokumentation mit einfach verwendbaren Code-Beispielen bieten).

Braga und Dahab [5] haben Online Foren durchsucht, um Kryptographie zu finden, die falsch verwendet wurde. „We found that, with surprisingly high probabilities (90% for Java and 71% for Android), several types of cryptography misuse can be found in the posts“ [5] (deutsch: Wir haben herausgefunden, dass mit einer erstaunlich hohen Wahrscheinlichkeit (90% für Java und 71% für Android) verschiedene Typen von Kryptographie, die falsch verwendet wurde, in solchen Posts gefunden werden können). Auch sie machen schwer zu nutzende Programmbibliotheken für dieses Problem verantwortlich.

Nun wird auf die Dokumentation der Bibliothek `cryptography` [9], im Folgenden der Klarheit wegen immer `cryptography.io` genannt, hingewiesen. Diese Bibliothek wird auch für die Beispiele, die im Laufe dieser Arbeit entstanden sind, verwendet. Warum diese Bibliothek verwendet wird, wird im Abschnitt 2.2 begründet. Die Dokumentation von `cryptograhpy.io` ist sehr gut und ausführlich. Es gibt zu den meisten Klassen und Funktionen ein kleines Beispiel. Der Nachteil, und warum CryptoExamples nötig ist, ist, dass die Dokumentation nicht darauf ausgelegt ist, vollständige, aktuelle und sichere Beispiele zu liefern. Wenn man ein Beispiel zu einer bestimmten Verschlüsselung haben möchte, muss man sich erst durch die Dokumentation arbeiten, um den richtigen Abschnitt zu finden. Außerdem gibt es zu jeder Klasse ein Beispiel, das heißt auch, für nicht mehr aktuelle kryptographische Verfahren, die große Sicherheitslücken aufweisen. `Cryptography.io` schlägt auch vor „Fernet“ zu benutzen. Laut ihrer Spezifikation[3] benutzt Fernet den „Advanced Encryption Standard (AES) 128 in CBC mode“. Wir diskutieren in Abschnitt 3.2.2, warum wir das nicht verwenden werden.

Des weiteren gibt es viele online Tutorials (deutsch: Anleitungen), Blog-Einträge oder Ähnliches. Pyprogramming.net[4], Matt Borgerson[5] und Jay Sridhar[6] beschreiben jeweils nur ein Beispiel zur Ver- und Entschlüsselung mit Hilfe des AES, welcher ein symmetrisches Verfahren ist. Der Vorteil dieser Beispiele ist, dass sie gut dokumentiert sind und meist ein Text vorangeht, in dem die Anwendungsfälle erklärt werden. Jedoch ist auch hier das Problem, dass die Beispiele nicht aktuell gehalten werden und nicht, oder zumindest weiß man es nicht genau, von Kryptologen gemacht oder verifiziert wurden. Seiten wie example-code.com[7], devarea.com[8], Laurent Luce's Blog[9], tutorialspoint.com[10], blog.pythonlibrary.org[11] bieten mehrere Beispiele an. Example-code.com hat sehr viele Beispiele zu verschiedenen, auch aktuellen, kryptographischen Verfahren. Der Nachteil allerdings ist, dass es eine so große Auswahl gibt. Das heißt, wenn symmetrische Verschlüsselung gefordert ist, bieten sich mehrere Beispiele mit verschiedenen Techniken an. Nutzer wissen dann wieder nicht, welche die aktuell Beste ist. Die anderen Seiten bieten eine umfangreiche Dokumentation an und erklären, was genau in ihren Beispielen passiert. Das ist für Nutzer natürlich sehr gut, da sie dann wissen, was sie programmieren. Jedoch geht dadurch die von uns verfolgte Eigenschaft der Kopierbarkeit verloren.

[3] `https://github.com/fernet/spec/blob/master/Spec.md` (abgerufen 2018-09-19)

[4] `https://pythonprogramming.net/encryption-and-decryption-in-python-code-example-with-explanation/` (abgerufen 2018-09-19)

[5] `https://mborgerson.com/cryptography-in-python-with-pycrypto/` (abgerufen 2018-09-19)

[6] `https://www.novixys.com/blog/using-aes-encryption-decryption-python-pycrypto/` (abgerufen 2018-09-19)

[7] `https://www.example-code.com/python/encryption.asp` (abgerufen 2018-09-19)

[8] `http://devarea.com/python-cryptographic-api/` (abgerufen 2018-09-19)

[9] `https://www.laurentluce.com/posts/python-and-cryptography-with-pycrypto/` (abgerufen 2018-09-19)

[10] `https://www.tutorialspoint.com/cryptography_with_python/` (abgerufen 2018-09-19)

[11] `https://www.blog.pythonlibrary.org/2016/05/18/python-3-an-intro-to-encryption/` (abgerufen 2018-09-19)

Als weitere verwandte Arbeiten kann man hier die Arbeiten an anderen Programmiersprachen auf CryptoExamples anführen. Auf dem GitHub Repository [8] gibt es schon programmierte Beispiele, jedoch noch keine geschriebenen Arbeiten. Es gibt Arbeiten zu den Programmiersprachen Java, C#, JavaScript, Rust und diese Arbeit zu Python. Verwandt sind sie in der Hinsicht, dass sie das gleiche Ziel verfolgen. Es sollen sichere, minimale, vollständige, kopierbare, ausführbare und getestete Beispiele zur Verfügung gestellt werden. Die Arbeiten unterscheiden von dieser, dass sie nicht für Python sind.

1.3 Ziel und Aufbau der Arbeit

1.3.1 Ziel der Arbeit

Ziel dieser Arbeit ist es unter anderem, die im Abschnitt 1.1 motivierten, sicheren Beispiele zu erstellen. Dabei sollen Beispiele zu symmetrischer und asymmetrischer Verschlüsselung sowie zu digitalen Signaturen, Hashing und Speicherung von Schlüsseln erstellt werden. An diese Code-Beispiele werden die Anforderungen gestellt, dass sie sicher, minimal, vollständig, kopierbar, ausführbar und getestet sind. Es ist jedoch nicht das einzige Ziel, dass die Code-Beispiele erstellt werden, sondern sie sollen auch in einem Code Projekt in der Organisation von CryptoExamples auf GitHub [8] veröffentlicht werden. Dabei soll das Projekt auch mit automatischen Tests versehen werden. Außerdem soll das Code Projekt auch in das Hauptprojekt CryptoExamples integriert werden. Im Zuge der Arbeit sollen auch generelle Richtlinien und ein damit verbundenes Umsetzungskonzept erstellt werden. Der Aufbau der Arbeit orientiert sich an diesen Zielen und wird im folgenden Abschnitt erläutert.

1.3.2 Aufbau der Arbeit

Die Arbeit gliedert sich in vier Kapitel. Während Kapitel 1 die Einleitung in diese Arbeit beinhaltet und Kapitel 2 die Grundlagen beschreibt, die für die Arbeit wichtig sind, steht der Hauptteil in Kapitel 3. Geschlossen wird die Arbeit von Kapitel 4.

Das Kapitel 1 ist in drei Teile unterteilt. Im Abschnitt 1.1 wird diese Arbeit motiviert. Dabei ist wichtig warum die Plattform CryptoExamples überhaupt benötigt wird und weshalb die Sprache Python dabei eine Rolle spielt. Anschließend werden in Abschnitt 1.2 verwandte Arbeiten behandelt. Zuletzt wird in diesem Abschnitt zuerst das Ziel der Arbeit definiert und dann genau an dieser Stelle ein Überblick über den Aufbau der Arbeit gegeben.

Das Kapitel 2 gibt einen Einblick in die Grundlagen, welche für die Arbeit benötigt werden. Zuerst werden in Abschnitt 2.1 die Grundlagen von kryptographischen Verfahren erläutert. Dabei werden die Verfahren beschrieben, welche für das Ziel, Code-Beispiele zu erstellen, benötigt werden. Weiter wird in Abschnitt 2.2 eine kryptographische Programmbibliothek gesucht, die verwendet werden kann, um die geforderten Verfahren zu implementieren. Die Standardbibliothek reicht nicht aus, weshalb eine externe Programmbibliothek benötigt wird. Zuletzt wird im Abschnitt 2.3 der Open Source Aspekt dieser Arbeit erläutert, und eine Lizenz genannt, die für die Code-Beispiele gilt.

Das Kapitel 3 ist der Hauptteil der Arbeit und beschreibt, wie die Code-Beispiele entstehen. Dabei werden zuerst in Abschnitt 3.1 generelle Richtlinien definiert, die beim Erstellen von Code-Beispielen, egal welcher Programmiersprache, gelten sollen. Das in Abschnitt 3.2 beschriebene Umsetzungskonzept soll diese Richtlinien für Python konkretisieren und legt die Wahl der verwendeten Algorithmen fest. In Abschnitt 3.3 werden die Code-Beispiele beschrieben, welche im Laufe der Arbeit entstanden sind. Dabei wird detailliert auf jedes Code-Beispiel eingegangen und es wird die Wahl der Schlüssellängen und Parameter begründet. Zuletzt wird noch in Abschnitt 3.4 auf die, auch zu dem Projekt zugehörigen, Tests und die Statische Code-Analyse eingegangen. Die eingesetzten Werkzeuge werden erklärt und die Ergebnisse präsentiert.

Das letzte Kapitel, Kapitel 4, fasst die Arbeit nochmals zusammen, und gibt einen Ausblick darauf, was im Zuge der Arbeit nicht passiert ist, und in zukünftigen Arbeiten enthalten sein sollte.

2 Grundlagen

2.1 Kryptographie

In diesem Abschnitt werden unter anderem die kryptographischen Verfahren erklärt, die für die Erstellung der Code-Beispiele benötigt werden. Außerdem werden auch Begriffe erklärt, welche für diese Verfahren von Bedeutung sind. Die Auswahl der Algorithmen für die einzelnen Verfahren erfolgt in Abschnitt 3.2.2 und wird dort auch begründet. Ferguson et al. [16] erklärt kryptographische Verfahren und auch die Algorithmen im Detail. Die folgenden Beschreibungen werden aus [16, S. 23 ff.] übernommen, bieten aber nur einen Überblick über die Verfahren. Weitere Details könne in [16, S. 23 ff.] nachgelesen werden. Es werden folgende Verfahren vorgestellt:

- Symmetrische Verschlüsselung
- Authentifizierung
- Asymmetrische Verschlüsselung
- Digitale Signatur
- Hashing
- Schlüsselableitung

Es wird das in der Kryptographie übliche Modell verwendet, welches in Abbildung 2.1 dargestellt ist. Dabei versucht Alice mit Bob zu kommunizieren, aber der Kommunikationskanal ist nicht sicher und Nachrichten können von Eve mitgehört werden.

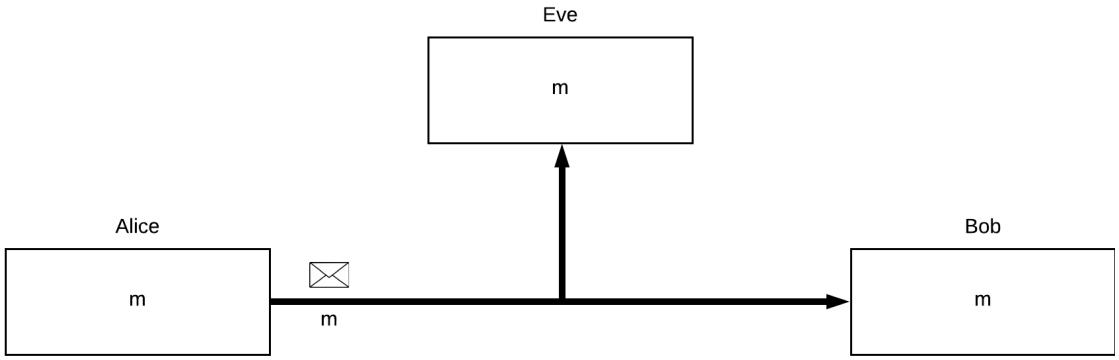

Abbildung 2.1: Kommunikation von Alice und Bob nach [16, S. 23]

2.1.1 Symmetrische Verschlüsselung

Bei der Verschlüsselung allgemein, aber auch besonders bei der symmetrischen Verschlüsselung ist es das Ziel, dass Alice und Bob kommunizieren können, ohne dass Eve etwas über den Inhalt der Nachrichten erfährt. Dafür werden die Nachrichten mit einem geheimen Schlüssel K_e verschlüsselt. Dieser Schlüssel ist Alice und Bob bekannt und muss vorher ausgetauscht werden, ohne dass auch Eve an diesen Schlüssel kommt. Wenn Alice nun eine Nachricht an Bob senden möchte, verschlüsselt sie diese mit der Verschlüsselungsfunktion $E(K_e, m)$. Statt der originalen Nachricht m wird die verschlüsselte Nachricht c über den von Eve überwachten Kanal gesendet. Bob kann die Nachricht nun mit der Entschlüsselungsfunktion $D(K_e, c)$ entschlüsseln und erhält somit die originale Nachricht m. Da Eve den vorher ausgetauschten Schlüssel K_e nicht kennt, kann sie die verschlüsselte Nachricht c nicht entschlüsseln. Ein gutes Verschlüsselungsverfahren sollte es unmöglich machen, die originale Nachricht aus der verschlüsselten ohne den Schlüssel wieder herzustellen. Es sollte auch nicht möglich sein, andere Informationen über die Nachricht zu erlangen, außer dessen Länge und Informationen über die Zeit, wann sie gesendet wurde.

„[..] Kerckhoffs' principle: the security of the encryption scheme must depend only on the secrecy of the key K_e and not on the secrecy of the algorithm“ [16, S. 24] (deutsch: Kerckhoffs' Prinzip: Die Sicherheit von einem Verschlüsselungsverfahren muss nur von der Sicherheit des Schlüssels K_e und nicht von der Geheimhaltung des Algorithmus abhängen). Eve kennt also die Ver- und Entschlüsselungsfunktionen $E(K_e, m)$ und (K_e, c), aber kann ohne den Schlüssel K_e die Nachricht nicht entschlüsseln. Es gibt gute Gründe für dieses Prinzip, unter anderem, dass Verschlüsselungsverfahren oft lange in Benutzung sind und es schon schwierig genug ist, einen einfachen Schlüssel zu schützen. Außerdem lassen sich Fehler in Verschlüsselungsverfahren viel einfacher finden, wenn diese veröffentlicht sind und jeder die Fehler erkennen kann.

2.1.2 Authentifizierung

An dieser Stelle soll noch die Authentifizierung vorgestellt werden, welche im Beispiel im Abschnitt 3.3.2 in Verbindung mit der symmetrischen Verschlüsselung verwendet wird. Eve kann die Nachrichten nicht nur lesen, sondern diese auch verändern, löschen, neue Nachrichten generieren, oder die Reihenfolge der bestehenden verändern. Es stellt sich also die Frage, wie sich Bob sicher sein kann, dass eine bestimmte Nachricht von Alice verfasst und abgeschickt wurde. Wie bei der symmetrischen Verschlüsselung wird auch für die Authentifizierung einer Nachricht ein geheimer Schlüssel, diesmal K_a, benötigt. Wenn Alice eine Nachricht senden will, erstellt sie einen Message Authentication Code (MAC) (deutsch: Nachrichtenauthentifizierungscode) a mit der MAC Funktion $h(K_a, m)$. Dann sendet sie beides, den MAC a und die Nachricht m an Bob. Dieser kann nun aus der empfangenen Nachricht mit der selben Funktion und dem selben geheimen Schlüssel K_a auch einen MAC erstellen und mit dem von Alice mitgelieferten vergleichen. So kann Bob erkennen, ob die Nachricht original ist, oder geändert wurde. Dies funktioniert, da eine gute MAC Funktion nicht das gleiche Ergebnis für zwei unterschiedliche Nachrichten produziert. Authentifizierung ist keine vollständige Lösung, da Eve immer noch die Nachrichten löschen, alte Nachrichten nochmals senden, oder die Reihenfolge der Nachrichten manipulieren kann. Es hilft, wenn Alice die Nachrichten nummeriert und Bob diese nur in der richtigen Reihenfolge oder einer Teilfolge davon akzeptiert. Authentifizierung wird oft in Verbindung mit Verschlüsselung verwendet, so auch in den Code-Beispielen der symmetrischen Verschlüsselung.

2.1.3 Asymmetrische Verschlüsselung

Es ist schwierig einen geheimen Schlüssel sicher auszutauschen. Wenn man jetzt auch noch eine größere Anzahl an Personen hat, müssen sehr viele Schlüssel ausgetauscht werden. Die asymmetrische Verschlüsselung oder auch *public-key* (deutsch: öffentlicher Schlüssel) Verschlüsselung bietet eine Lösung für dieses Problem. Der große Unterschied bei der asymmetrischen Verschlüsselung ist, dass Alice und Bob nicht mehr den selben Schlüssel verwenden. Um die Kommunikation zu starten muss Bob zunächst ein Schlüsselpaar $(S_{\text{Bob}}, P_{\text{Bob}})$ generieren. Der Schlüssel S_{Bob} ist der private, oder auch sichere Schlüssel, welcher nicht geteilt wird. Bob stellt aber den öffentlichen Schlüssel P_{Bob} für jeden zugreifbar zur Verfügung. Es können also beide, Alice und Eve, diesen Schlüssel abrufen. Wenn Alice nun eine Nachricht an Bob senden möchte, muss sie den öffentlichen Schlüssel von ihm abrufen und damit die Nachricht m verschlüsseln. Die so verschlüsselte Nachricht c kann nun an Bob gesendet werden, der diese mit dem Entschlüsselungsalgorithmus und seinem privaten Schlüssel S_{Bob} entschlüsseln kann und die originale Nachricht m erhält. Um dieses Verfahren zu ermöglichen, muss für alle möglichen Nachrichten gelten: $D(S_{\text{Bob}}, E(P_{\text{Bob}}, m)) = m$. Die Algorithmen zum Ver- und Entschlüsseln unterscheiden sich voneinander. Die Sicherheit dieser Algorithmen basiert auf sogenannten Einwegfunktionen (englisch: one-way functions). Die zugrundeliegenden mathematischen Funktionen funktionieren in die eine Richtung sehr schnell, aber sind in der anderen Richtung fast nicht lösbar. So beruht zum Beispiel die Sicherheit von dem Rivest Shamir Adleman (RSA) Verfahren darauf, dass die Multiplikation von Primzahlen einfach, aber die Primfaktorzerlegung schwer ist. Da die mathematischen Details an dieser Stelle keine Rolle spielen, können weiterführende Informationen in [16] nachgelesen werden.

Durch die Bereitstellung der öffentlichen Schlüssel wird das Problem, dass geheime Schlüssel geteilt werden müssen, sehr viel einfacher. Wenn jede Person ihren öffentlichen Schlüssel teilt, kann eine andere Person mit dieser sicher kommunizieren. Jedoch muss auch dieses Teilen verwaltet werden und es muss garantiert werden, dass auch wirklich der korrekte öffentliche Schlüssel einer Person erhalten wird. Wie das geregelt ist kann in [16, S. 29 f.] nachgelsen werden. Das symmetrische Verschlüsselungsverfahren wesentlich effektiver als das asymmetrische. In den meisten Systemen kann man deshalb eine Verbindung von beiden Verfahren finden. Das asymmetrische Verfahren wird verwendet, um den geheimen Schlüssel für das symmetrische Verfahren auszutauschen.

2.1.4 Digitale Signatur

„Digital Signatures are the public-key equivalent of message authentication codes“ [16, S. 29] (deutsch: Digitale Signaturen sind das *public-key* Equivalent zu Nachrichtenauthentifizierungscodes). Es geht wieder darum, wie Bob sich sicher sein kann, dass eine bestimmte Nachricht wirklich von Alice stammt. Dabei erstellt auch Alice ein Schlüsselpaar $(S_{\text{Alice}}, P_{\text{Alice}})$ und stellt ihren öffentlichen Schlüssel zur Verfügung. Mittels der Funktion zum Signieren $\sigma(S_{\text{Alice}}, m)$ erstellt Alice eine Signatur s aus der Nachricht m, die sie beide zu Bob sendet. Bob kann dann die Funktion $\nu(P_{\text{Alice}}, m, s)$ verwenden, um die Nachricht zu verifizieren, dafür verwendet er den öffentlichen Schlüssel P_{Alice} von Alice. Im Prinzip ist es das gleiche Verfahren wie das zum Authentifizieren mit einem geteilten, geheimen Schlüssel, außer dass der sichere, private Schlüssel zum Signieren und der öffentliche Schlüssel zum Verifizieren ist. Da der öffentliche Schlüssel, wie der Name schon sagt öffentlich ist, kann nicht nur Bob, sondern jeder die Nachricht verifizieren. Deshalb heißt es auch Signatur, da Alice eine Nachricht signiert und für alle sichtbar ist, dass diese Nachricht von ihr stammt. Es

gibt auch ein paar Probleme mit der digitalen Signatur. Eines davon ist, dass Alice die Nachricht nicht persönlich signiert, sondern ihr Computer diese Arbeit für sie übernimmt. Es kann also sein, wenn zum Beispiel ein Virus auf ihrem Computer ist, dass von ihr signierte Nachrichten verschickt werden, die sie nicht selbst erstellt hat.

2.1.5 Hashing

Als Hash-Funktion (to hash, deutsch: zerhacken) bezeichnet man eine Funktion, die eine beliebig große Zeichenkette auf eine Zeichenkette mit fester Länge, den Hashwert, abbildet. Auf den mathematischen Hintergrund soll hier nicht eingegangen werden, dieser wird aber von Cormen et al. [7] ausführlich beschrieben. Eine Hash-Funktion muss gewisse Sicherheitseigenschaften erfüllen, um für den kryptographischen Zweck nutzbar zu sein. Es ist wichtig, dass die Hash-Funktion nicht umgekehrt werden kann. Wenn man einen Hashwert und die zugehörige Funktion hat, soll es nicht möglich sein, die ursprüngliche Eingabe wiederherzustellen. Außerdem wird erwartet, dass die Hash-Funktion gegen Kollisionen, also dass zwei Eingabewerte auf den gleichen Ausgabewert abgebildet werden, resistent ist. Dabei ist es unmöglich, dass keine Kollisionen entstehen, da eine unendliche Menge auf eine endliche abgebildet wird. Kollisionsresistenz sagt nur, dass diese Kollisionen nicht gefunden werden können. Sichere Hash-Funktionen werden vom National Institute of Standards & Technology (NIST) der Vereinigten Staaten von Amerika ausgesucht und definiert.

2.1.6 Schlüsselableitung

Eine Key Derivation Function (KDF) (deutsch: Schlüsselableitungsfunktion) ist eine Funktion, die aus einem Passwort oder anderem sicheren Text einen oder mehrere sichere Schlüssel generiert [24, S. 4 ff.]. Da Passwörter nicht direkt für kryptographische Verfahren anwendbar sind, werden bestimmte Prozesse angewendet, um einen für das Verfahren verwendbaren Schlüssel aus diesem abzuleiten. Dafür werden Hash-Funktionen verwendet. Es ist aber sehr leicht, diese mit einem Wörterbuchangriff[1] zu knacken. Bei einem Wörterbuchangriff werden bekannte Passwörter schon vorher gehasht, welche dann mit den Hashwerten aus zum Beispiel einer Datenbank verglichen werden können. Treten gleiche Werte auf, kann man so das Passwort herausfinden. Es gibt unterschiedliche Ansätze dies zu erschweren, welche von den Public-Key Cryptography Standards (PKCS) #5 [19, S. 4 ff.] definiert werden. Zum einen kann ein *salt* (deutsch: Salz) mit dem Passwort kombiniert werden. Durch die Zugabe eines *salt* zu einem Passwort bevor es gehasht wird, wird es für einen Angreifer sehr schwer alle möglichen Schlüssel, die zu einem Passwort gehören, zu generieren. Außerdem ist es sehr unwahrscheinlich, dass bei einem zufälligen, großen *salt* gleiche Schlüssel aus dem selben Passwort generiert werden. Ein weiterer Ansatz ist, dass der Prozess zum Ableiten eines Schlüssels mehrfach iteriert wird. Dabei wird beim Erstellen des Schlüssels nur geringfügig mehr Zeit benötigt, während es für Angreifer erheblich schwerer wird, das Passwort herauszufinden. Beide Ansätze können kombiniert werden.

[1] `https://www.searchsecurity.de/definition/Woerterbuchangriff-Dictionary-Attack` (abgerufen 2018-09-19)

2.2 Kryptographische Programmbibliotheken

Warum Python einen Platz in CryptoExamples verdient und deshalb diese Arbeit gerechtfertigt ist, wurde schon in der Motivation in Abschnitt 1.1 ausgeführt. Da CryptoExamples für Python unter anderem die zu diesem Zeitpunkt aktuellste stabile Version 3.7 unterstützt, wird im Folgenden davon ausgegangen, dass diese Version verwendet wird. Die Standardbibliothek[2] von Python verfügt über eine sehr limitierte Anzahl an kryptographischen Modulen. Es gibt das *hashlib*, *hmac* und *secrets* Modul. Das *hashlib* Modul stellt Hash-Funktionen bereit und kann Schlüssel ableiten. Das *hmac* stellt eine Variante des in Abschnitt 2.1.2 beschriebenen MAC Verfahren zur Verfügung und das *secrets* Modul ist ein sicherer Zufallszahlengenerator. Mindermann und Wagner [22] legen fest, dass bei CryptoExamples versucht wird, nur Funktionalitäten der Standardbibliothek der jeweiligen Sprache zu verwenden. Es ist jedoch nicht möglich Code-Beispiele für die geforderten kryptographischen Verfahren zu erstellen, da die angebotenen Module der Standardbibliothek unzureichend sind. Aus diesem Grund muss eine externe kryptographische Programmbibliothek ausgewählt werden.

Bei der Suche nach einer Programmbibliothek, die diese Zwecke unterstützt, wurden einige gefunden. Es wurde eine einfache Suche auf Google[3] gestartet, welche die im folgenden aufgelisteten Programmbibliotheken hervorbrachte. Die Liste hat keinen Anspruch auf Vollständigkeit, da zum Beispiel Acar et al. [1] mehr Werkzeuge gefunden haben. Jedoch genügt diese Auswahl, da wie wir weiter unten sehen, die populärsten dabei sind.

cryptography.io `Cryptography.io` ist eine Programmbibliothek, welche geläufige kryptographische Algorithmen anbietet. Sie setzt sich als Ziel, die kryptographische Standardbibliothek für den Nutzer zu werden und bietet zum einen eine Schnittstelle mit hoher Nutzerfreundlichkeit an, bei der man keine Parameter mehr eintragen muss. Sie bietet aber auch eine Schnittstelle an, die es ermöglicht, die Algorithmen auf unterster Ebene zu nutzen [9].

PyCrypto `PyCrypto` ist eine weitere Programmbibliothek, die eine Fülle an Algorithmen und Funktionen anbietet. Acar et al. [1] schreibt, dass sie die populärste sei. Jedoch wurde die Programmbibliothek zuletzt 2014 aktualisiert und es gibt viele Sicherheitswarnungen, sie zu verwenden [30].

PyCryptodome `PyCryptodome` ist eine Abspaltung von `PyCrypto` und wird aktiv entwickelt. Die Entwickler betonen, dass sie kein Wrapper (englisch für Umschlag) für OpenSSL oder eine andere C Bibliothek sind, sondern Python direkt verwenden [27].

PyNaCl `PyNaCl` ist eine Bindung für Python an die libsodium[4] Programmbibliothek [28] und ist von der gleichen Organisation wie `cryptography.io`. Auch diese Programmbibliothek stellt alle nötigen Algorithmen zur Verfügung.

[2] `https://docs.python.org/3/library/` (abgerufen 2018-09-19)
[3] `https://www.google.com/` (abgerufen 2018-09-19)
[4] `https://download.libsodium.org/doc/` (abgerufen 2018-09-19)

PyOpenSSL `PyOpenSSL` ist ebenfalls von der gleichen Organisation wie `cryptography.io` und ist ein Wrapper für die OpenSSL Bibliothek [29]. Die Autoren empfehlen allerdings dringend, `cryptography.io` statt dieser Programmbibliothek zu verwenden.

M2Crypto `M2Crypto` ist auch ein Wrapper für die OpenSSL Bibliothek. Acar et al. [1] bezeichnet sie allerdings als komplettere Alternative, als zum Beispiel `PyOpenSSL`. Sie wird jedoch nur noch sehr sporadisch weiterentwickelt.

Keyzar Zuletzt gibt es noch die `Keyzar` Programmbibliothek, welche sich als Werkzeug beschreibt, die es einfacher und sicherer macht, Kryptographie in Anwendungen zu benutzen [20]. Es gibt `Keyzar` für verschiedene Sprachen, unter anderem Python, das Ziel sei aber nicht, existierende Programmbibliotheken wie OpenSSL oder `PyCrypto` zu ersetzen.

In die engere Auswahl kommen die Programmbibliotheken, welche alle in Abschnitt 1.3.1 definierten Verfahren unterstützen. Das sind `cryptography.io`, `PyCrypto`, `M2Crypto` und `PyCryptodome` [1]. Von diesen Programmbibliotheken haben `cryptography.io` und `PyCrypto` die meisten Downloads. Die Downloadzahlen dienen als Indikator für die Popularität. Da `PyCrypto` seit 2014 nicht mehr aktiv entwickelt wird und deshalb mit einer hoher Wahrscheinlichkeit Sicherheitslücken beinhaltet, entfällt diese Wahl. `cryptography.io` ist die favorisierte Programmbibliothek und wird verwendet, um die Code-Beispiele zu erstellen.

2.3 Open Source

Open Source bedeutet erst einmal, dass der Quellcode von Software frei zugänglich ist. Lizenzbedingungen legen fest, was ein Nutzer mit der Software machen darf. Open Source Software darf, von der Lizenz abhängig, geändert und für den eigenen Zweck angepasst werden. Sowohl die unveränderte, als auch die veränderte Software darf unter Umständen geteilt werden. Die Open Source Initiative stellt zehn Kriterien vor, die für Lizenzen von Open Source Software gelten müssen [34]. Für alle Code-Beispiele von CryptoExamples und damit auch von diesem Projekt gilt die *The Unlicense* [38]. Der Name der Lizenz suggeriert, dass der Code dadurch nicht lizenziert ist. Da nicht lizenzierter Code aber legaler weise gar nicht verwendet werden darf, ist der Name etwas ungeschickt gewählt. Es geht darum, dass der Code ohne einschränkende Bedingungen verwendet werden darf. Die Unlicense erlaubt es jedem, die Code-Beispiele zu kopieren, verändern, veröffentlichen, verwenden, kopieren, verkaufen und verbreiten. Diese Lizenz wird verwendet, um zu erreichen, dass die Code-Beispiele von jedem für den eigenen Code verwendet werden können und die Nutzer sich keine Gedanken über Urheberrechte machen müssen. Außerdem müssen die Nutzer nicht vermerken, dass sie diese Code-Beispiele verwendet haben. Die Unlicense bedeutet aber auch, dass die Autoren der Beispiele keine Garantie übernehmen, und nicht haftbar gemacht werden können.

3 Richtlinien und Umsetzung

3.1 Generelle Richtlinien

Es gibt schon bestehende Richtlinien für CryptoExamples[1]. Trotzdem sollen hier nochmals Richtlinien definiert werden. Die Richtlinien, die im Folgenden dokumentiert werden, gelten nicht spezifisch für Python, sondern allgemein für CryptoExamples. Es wird unterteilt in das Auswählen von Algorithmen und Parametern, die Erstellung von neuen und die Weiterentwicklung und Wartung der bestehenden Beispiele.

Da schon Richtlinien bestehen, und auf diesen basiert auch schon Code-Beispiele einstanden sind, macht es Sinn manche Richtlinien, wie zum Beispiel im Abschnitt 3.1.2 Richtlinie 1, zu übernehmen. Dabei ist darauf zu achten, dass manche Richtlinien sehr stark und andere nur leicht geändert wurden. Andere Richtlinien sind zwar sowohl in dem bestehenden als auch in dem hier definierten Katalog vorhanden, wurden aber eigenständig erarbeitet. Übernommen wurden im Abschnitt 3.1.1 die Richtlinie 1 und in Abschnitt 3.1.2 die Richtlinien 1, zum Teil 2, zum Teil 4 und 6. Ähnlich oder gleich sind die Richtlinien 3, 5, 8, 11, 12, 13, 14 und 15 im Abschnitt 3.1.2.

Die Richtlinien wurden mit dem Hintergrund erstellt, die Anforderungen sicher, minimal, vollständig, kopierbar, ausführbar und getestet in konkrete Punkte zu fassen. Sie werden im Abschnitt 3.2 in Python umgesetzt und dann auf die Code-Beispielen im Abschnitt 3.3 angewandt.

3.1.1 Auswahl von Algorithmen und Parametern

1. Es werden offizielle bzw. vertrauenswürdige Quellen eingesehen, um einen Algorithmus, eine Schlüssellänge und eventuelle andere Parameter auszusuchen. Gute Quellen sind unter anderem das Bundesamt für Sicherheit in der Informationstechnik (BSI)[2] und das NIST[3]. Auf keylength.com[4] können vertrauenswürdige Quellen verglichen werden. Es werden mindestens zwei Quellen zu Rate gezogen, um zu vermeiden, dass fehlerhafte oder unsichere Informationen bezogen werden.
2. Es wird die Dokumentation der kryptographischen Programmbibliothek oder der Standardbibliothek eingesehen, um die dort in Beispielen verwendeten oder empfohlenen Algorithmen, Schlüssellängen und andere Parameter zu evaluieren. Es wird auch überprüft, welche Techniken überhaupt vorhanden sind.

[1] `https://github.com/cryptoexamples/CryptoExamples-Guidelines` (abgerufen 2018-09-19)
[2] `https://www.bsi.bund.de/` (abgerufen 2018-09-19)
[3] `https://www.nist.gov/` (abgerufen 2018-09-19)
[4] `https://www.keylength.com/` (abgerufen 2018-09-19)

3. Es wird der beste Algorithmus, die beste Schlüssellänge und geeignete Parameter gewählt. Dabei gibt es verschiedene Faktoren, die darauf einwirken, was das Beste ist. Wenn es mehrere Algorithmen oder andere Parameter zur Auswahl gibt, muss eine Abwägung zwischen der Stärke des Algorithmus und dessen Performanz gemacht werden. Dabei bedeutet Stärke die Zeit, die gebraucht wird, um den Algorithmus zu brechen, wobei auf Schwachstellen geachtet werden muss, die diese Zeit reduzieren. Deshalb ist es wichtig, auf die in Richtlinien 1 und 2 dieses Abschnitts gefundenen Quellen zu achten. Dabei werden offizielle bzw. vertrauenswürdige Quellen bevorzugt verwendet. Wenn jedoch überzeugend argumentiert wird, kann die Lösung aus der Dokumentation verwendet werden.

3.1.2 Erstellung von Beispielen

1. Für jedes Beispiel wird eine neue Datei mit dem Präfix *example_* erstellt. Der andere Teil des Dateinamens spiegelt wieder, was in dem Beispiel erreicht werden soll.
2. Die Importe aus anderen Programmbibliotheken stehen immer oben. Sie werden einzeln aufgelistet und sind explizit, um keine unnötigen Module zu importieren, und damit der Nutzer sieht, was importiert und auch verwendet wird. Die Importe der kryptographischen Programmbibliothek werden aus Gründen der Übersichtlichkeit gruppiert und durch eine Leerzeile von den anderen Importen getrennt.
3. Es wird ein *logger* implementiert, welcher im Beispiel verwendet werden kann. Andere Mechanismen schreiben meist alle Ausgaben in die Standardausgabe. Da die Beispiele von anderen Entwicklern verwendet und in ihre Projekte übernommen werden sollen, muss das vermieden werden. Es gilt als schlechtes Verhalten Ausgaben in die Standardausgabe zu schreiben, da die Ausgaben so nicht richtig verwaltet werden können. Mit einem *logger* kann das *logging* ausgeschaltet oder die Ausgaben an die richtigen Stellen geleitet werden.
4. Es wird eine Funktion implementiert, welche den Präfix *demonstrate_* trägt und dessen Name das Ziel des Beispiels ist. Diese Vorführ-Funktion repräsentiert den kompletten Ablauf des Beispiels. Das kann zum Beispiel Vorbereiten, Verschlüsseln und Entschlüsseln sein. Andere Funktionen sind erlaubt, werden aber nach Möglichkeit vermieden, um für eine bessere Übersicht zu sorgen.
5. Es existiert eine *main* Methode oder, in Programmiersprachen die keine *main* Methode haben, ein entsprechendes Konstrukt. Durch diese Methode lässt sich das Code-Beispiel direkt ausführen.
6. In einem Methodenkommentar wird beschrieben, welche Techniken und Verfahren in dem Beispiel angewendet werden.
7. Der Code ist in verschiedene Blöcke strukturiert. Diese Blöcke repräsentieren einzelne Schritte des Beispiels und werden durch einen beschreibenden Kommentar eingeleitet. Blöcke könnten zum Beispiel die Schlüsselgenerierung, Verschlüsselung oder Entschlüsselung darstellen.

8. *Exceptions* (deutsch: Ausnahmen) werden im Code abgefangen und behandelt. Dabei ist vor allem wichtig, dass die *Exceptions* der kryptographischen Programmbibliothek abgefangen werden. Konstrukte wie *try: except:* umfassen möglichst viel Code, damit dieser von dem Abfangen der *Exceptions* nicht unterbrochen wird. Damit wird die Übersichtlichkeit gewährleistet.
9. Wenn namentliche Parameter zur Verfügung stehen, werden diese benutzt. Dies ist vor allem für die kryptographischen Programmbibliothek wichtig, damit dessen Methodenaufrufe verstanden werden.
10. Der Code ist so programmiert, dass Parameter und Algorithmen einfach ausgetauscht werden können und sie nur an wenigen Stellen geändert werden müssen. Dies ist wichtig, sollten diese nicht mehr aktuell sein, oder wenn der Nutzer diese für das eigene Projekt ändern möchte.
11. Am Ende der Vorführ-Funktion steht eine Ausgabe des *loggers*, welche aufzeigt, ob das ausgeführte Verfahren ein korrektes Ergebnis liefert.
12. Es wird mindestens eine Testfunktion implementiert. Diese Testfunktion ist ein sogenannter *unit test*, welcher die Vorführ-Funktion ausführt und ihr Ergebnis mit einem vordefinierten Sollwert vergleicht. So wird sichergestellt, dass das Verfahren im Code-Beispiel korrekt ausgeführt wird.
13. Es wird dem Programmierstil der entsprechenden Programmiersprache gefolgt.
14. Es wird Statische Code-Analyse betrieben. Dabei werden Werkzeuge verwendet, um die Testabdeckung herauszufinden, Code Richtlinien der Programmiersprache zu überprüfen und Sicherheitsprobleme zu entdecken.
15. Das Beispiel funktioniert mit der letzten stabilen Version der Programmiersprache.

3.1.3 Weiterentwicklung und Wartung der Beispiele

1. Es wird überprüft, ob die in dem Beispiel verwendeten Algorithmen, Schlüssellängen und sonstigen Parameter noch aktuell und sicher sind. Dabei wird nach dem Prinzip in 3.1.1 vorgegangen.
2. Die Programmiersprache wird auf neue Versionen geprüft und wenn nötig wird diese angepasst. Das Beispiel sollte immer für die aktuellste Version funktionieren. Wenn vorangegangene Versionen unterstützt werden können ist das gut, jedoch nicht das Ziel.
3. Die kryptographische Programmbibliothek, falls vorhanden, wird auf neue Versionen geprüft. Es wird dabei nachgesehen, ob Änderungen existieren, die das Verhalten und die Sicherheit der Beispiele beeinflussen. Es wird auch überprüft, ob neue Algorithmen oder Parameter implementiert wurden, welche sich eignen, um in einem Beispiel angewendet zu werden.

3.2 Umsetzungskonzept

In diesem Abschnitt soll auf das Konzept eingegangen werden, das erstellt wird um die Code-Beispiele zu schreiben. Im Detail wird darauf eingegangen, wie die im vorigen Abschnitt definierten Richtlinien auf Python angewendet werden und wie bei der Implementierung darauf geachtet wird, dass die Begriffe sicher, minimal, vollständig, kopierbar, ausführbar und getestet eingehalten werden.

3.2.1 Einhaltung der Richtlinien

Das Auswählen von Algorithmen, Schlüssellängen und sonstigen Parametern ist relativ unabhängig von der Programmiersprache. Es wird bei der Auswahl der Algorithmen darauf geachtet, dass diese in der gewählten kryptographischen Programmbibliothek cryptographie.io vorhanden sind. Außerdem werden auch die Beispiele und Vorschläge der Dokumentation in Betracht gezogen. Welche Algorithmen für einzelnen kryptographischen Verfahren gewählt wurden, wird im Abschnitt 3.2.2 begründet. Die Auswahl der Schlüssellänge und Parameter wird individuell für jedes Code-Beispiel getroffen Es wird meist auf die Angaben in den offiziellen Quellen vertraut und nur in dem Beispiel der digitalen Signatur, im Abschnitt 3.3.6, musste aufgrund der Programmbibliothek ein anderer Parameter gewählt werden.

Die Richtlinien zum Erstellen von Beispielen werden in Abschnitt 3.1.2 definiert. Es wird hier vor allem auf die Richtlinien eingegangen, die eine Python spezifische Anwendung finden. Alle Code-Beispiele werden in dem Ordner *src/cryptoexamples* gelegt und mit dem *example_* Präfix ausgestattet. Die Importe zu Beginn jedes Beispiels aufgelistet. Dabei werden Importe von `cryptography.io` gruppiert und es wird das in Python vorhandene `from ... import ...` verwendet, um die benötigten Module explizit zu importieren. Für den *logger* wird das von der Standardbibliothek bereitgestellte Modul *logging* verwendet. Alle Beispiele enthalten nur genau eine Funktion und zwar die Vorführ-Funktion, welche den Präfix *demonstrate_* trägt. Auf Hilfsfunktionen kann aufgrund der Übersichtlichkeit und der Einfachheit von Python verzichtet werden. In Python gibt es keine *main* Methode, sondern es wird das für Python übliche `if __name__ == "__main__"` Konstrukt verwendet. Dieses führt das Beispiel aus, wenn es direkt als Skript, zum Beispiel mit *python example.py*, aufgerufen wird. Wird die Datei nur als Modul importiert, wird dieser Code nicht ausgeführt. Dies ist zum Beispiel für die Tests wichtig, da in diesen der Code eventuell nicht beim Import ausgeführt werden soll.

In Python ist ein sogenannter *docstring* möglich, welcher direkt in der Zeile nach dem Methodenkopf mit ” ” ” begonnen und so auch später wieder geschlossen wird. Für das Abfangen der *Exceptions* wird der Code innerhalb der Vorführ-Funktion mit dem `try: except:` Konstrukt umschlossen. In Python gibt es die Möglichkeit, Parameter mit oder ohne Namen anzugeben. Es wird darauf geachtet, immer die namentliche Parameterübergabe zu verwenden, außer wenn nur ein Parameter übergeben wird. In allen Beispielen können die Parameter einfach ausgetauscht werden. Sollte man 256 statt 512 *bit* verwenden wollen, muss man nur genau das ändern. Auch Schlüssellängen können durch das einfache Tauschen von Integer Werten geändert werden. Jede Vorführ-Funktion wird mit einer `logger.info()` Ausgabe abgeschlossen, in der bei den meisten Beispielen der Originaltext mit dem einmal verschlüsselten und dann wieder entschlüsselten Text verglichen wird.

Die Testdatei liegt im Ordner *tests* und beinhaltet für jedes Beispiel einen *unit test*, der die Vorführ-Funktion ausführt. Weitere Informationen zu den Tests gibt es in Abschnitt 3.4.1. Aller Python Code, der im Zuge dieser Arbeit geschrieben wurde, folgt dem von Python vorgegebenen PEP8[5] Programmierstil. Für die Statische Code-Analyse werden drei Werkzeuge verwendet. Weitere Informationen dazu finden sich im Abschnitt 3.4.2. Bei dem Beginn der Implementierung der Code-Beispiele, war 3.6 die aktuellste Version von Python. Später wurde dann die Version Python 3.7 veröffentlicht, welche aber keine Veränderungen an dem Code erzwangen. Diese beiden Versionen werden unterstützt und von dem Werkzeug travis[6], welches kontinuierliche Integration für GitHub Projekte anbietet, verwendet. Zu einem späteren Zeitpunkt wurde entschieden auch Python 2.7 zu unterstützen. Python 2.7 ist eine immer noch stark vertretene Version und selbst neue Software wird oftmals noch in Python 2.7 geschrieben. Es mussten kleine Änderungen vorgenommen werden, aber nun unterstützen alle Beispiele Python 2.7, 3.6 und 3.7.

3.2.2 Wahl der Algorithmen

In diesem Abschnitt wird die Wahl der Algorithmen begründet. Dabei wird auf die verschiedenen Kategorien eingegangen. Es wird ein Algorithmus für die symmetrischen Verfahren, für die asymmetrischen Verfahren, für die Schlüsselableitung und Hashfunktionen ausgewählt. Die Auswahl basiert auf den in Abschnitt 3.1.1 festgelegten Richtlinien.

Symmetrische Verfahren Das BSI schreibt in ihren Technischen Richtlinien ein ganzes Kapitel für die symmetrische Verschlüsselung [21, S. 22 ff.]. Für die Wahl des Algorithmus muss ein Blockchiffre gewählt werden und eine sogenannte Betriebsart. Ein Blockchiffre verschlüsselt eine Eingabe fester Länge zu einer verschlüsselten Ausgabe gleicher Länge. Mittels einer Betriebsart wird aus dem Blockchiffre ein Verfahren gemacht, dass Eingaben beliebiger Größe erlaubt. Eine Betriebsart legt fest, wie die Blöcke nacheinander verschlüsselt werden. Als Blockchiffre wird der AES empfohlen. Dieser soll mit den Eingabegrößen 128, 192 oder 256 *bit* verwendet werden. Das NIST genehmigt für diesen Blockchiffre entweder AES oder Triple Data Encryption Algorithm (TDEA) [3, S. 23 f.]. Sie schreiben aber auch, dass der TDEA zum Ende des Jahres 2015 nicht mehr für neue Software verwendet werden soll. Sie empfehlen auch die Größen 128, 192 oder 256 *bit*. `Cryptography.io` bietet noch weitere Blockchiffren an, aber da die Empfehlungen der offiziellen Quellen übereinstimmen, wird der AES, welcher von Daemen und Rijmen [10] entwickelt und von NIST [15] als AES definiert ist, gewählt.

Die Betriebsverfahren, die laut BSI für den AES Algorithmus empfohlen werden, sind der Galois/-Counter Mode (GCM), Cipher-Block Chaining (CBC) oder Counter Mode (CTR) [21, S. 23 f.]. Der GCM beinhaltet schon eine Funktion für die Datenauthentifizierung, für die anderen beiden Betriebsmodi wird empfohlen einen separaten Mechanismus für die Datenauthentifizierung einzubauen. Das NIST empfiehlt eine Reihe von Betriebsmodi, wobei die empfohlenen des BSI in diesen enthalten sind [3, S. 24]. Es muss also zwischen GCM, CBC und CTR gewählt werden. CTR ist ein sogenannter Stromchiffre (englisch: stream cipher), was bedeutet, das der Klartext, der verschlüsselt werden soll, mit einem zufällig generierten Strom von *bytes* kombiniert wird. Ferguson et al. [16]

[5] `https://www.python.org/dev/peps/pep-0008/` (abgerufen 2018-09-19)
[6] `https://travis-ci.org/` (abgerufen 2018-09-19)

halten CBC und CTR für sicher und empfehlen aber CBC zu verwenden [16, S. 71 f.]. Bei dieser Empfehlung beachten sie aber nicht GCM, da dieser Modus nicht nur für die Verschlüsselung, sondern eine Verbindung aus Verschlüsselung und Authentifizierung ist. Da aber auf jeden Fall eine Verbindung aus Verschlüsselung und Authentifizierung gefordert ist, wird der neuere GCM verwendet. Bei CBC müsste man noch zusätzlich eine Authentifizierung implementieren, durch die Anforderung der Minimalität, empfiehlt es sich eher den GCM zu verwenden. Ferguson et al. [16] schreiben, dass man bei GCM unbedingt eine Schlüsselgröße von mindestens 128 *bit* verwenden soll, da sonst Sicherheitsbedenken bestehen [16, S. 113]. Diese Vorgabe ist aber sowieso gegeben für das Blockchiffre gegeben, wie oben beschrieben wird.

Alle hier genannten Betriebsmodi brauchen einen Initialisierungsvektor, eine sogenannte nonce (number only used once - deutsch: Zahl, die nur einmal verwendet wird). In der Definition vom GCM steht, dass bei allen Instanzen, bei denen ein gewisser Schlüssel verwendet wird, unterschiedliche noncen benutzt werden müssen, da sonst eine Anfälligkeit für Attacken besteht [13, S. 18].

Schlüsselableitung Für die Schlüsselableitung wird die Password-Based Key Derivation Function 2 (PBKDF2) Funktion in Verbindung mit Keyed-Hash Message Authentication Code (HMAC) Codes verwendet. Alternativ zu PBKDF2 gibt es noch die Verfahren scrypt, bcrypt und Argon2. Bcrypt wird als gleich sicher wie PBKDF2 betrachtet, jedoch wurde PBKDF2 mit HMAC von NIST als passwortbasierte Schlüsselableitung empfohlen [37, S. 7]. Argon2 ist ein neueres Verfahren, jedoch wird immer noch PBKDF2 empfohlen [24, S. 8]. Außerdem steht weder bcrypt noch Argon2 in der kryptographischen Programmbibliothek `cryptography.io` zur Verfügung. Das BSI gibt nur eine Empfehlung für Schlüsselableitungen bei Schlüsselaustauschverfahren [21, S. 72], aber das Dokument, auf das dabei verwiesen wird, wurde von NIST als veraltet eingestuft. Generell für MACs werden Cipher-based Message Authentication Code (CMAC), HMAC und Galois Message Authentication Code (GMAC) empfohlen [21, S. 42 f.]. Da in Verbindung mit PBKDF2 HMAC empfohlen wird, wird dieses verwendet.

Asymmetrische Verfahren Das BSI gibt für die asymmetrische Verschlüsselung drei Verfahren vor [21, S. 35 ff.]. Das Elliptic Curve Integrated Encryption Scheme (ECIES), das Discrete Logarithm Integrated Encryption Scheme (DCIES) und das RSA Verfahren, wobei alle Algorithmen auf unterschiedlichen mathematischen Verfahren basieren. Es wird aber nicht genannt, welches davon bevorzugt verwendet werden soll. Die `cryptography.io` Programmbibliothek bietet verschiedene Algorithmen zum Schlüsselaustausch an, die zum Teil die oben genannten Verfahren verwenden. Da aber eine einfache *public-key* Verschlüsselung stattfinden soll, wird das RSA Verfahren gewählt. Elliptische Kurven wurden in dieser Arbeit gar nicht behandelt, siehe Abschnitt 4.2. Für das Verwenden von RSA ist ein Padding (von englisch to pad - auffüllen) nötig, da die Eingabe auf eine entsprechende Größe gebracht werden muss. Das BSI empfiehlt dabei nur das Optimal Asymmetric Encryption Padding (OAEP) Verfahren, welches in [23] definiert ist [21, S. 38]. Das ältere PKCS#1v1.5 Padding wird nicht gebilligt. Das OAEP Padding benötigt eine Mask Generation Function (MGF), dabei kommt nur die MGF1 Funktion in Frage, da diese als einzige von `cryptography.io` angeboten wird, und auch die einzige ist, die in [23, S. 66 f.] definiert wird.

Für die digitale Signatur werden von NIST das RSA, das Digital Signature Algorithm (DSA) und das Elliptic Curve Digital Signature Algorithm (ECDSA) Verfahren angenommen [3, S. 25]. Auch das BSI empfiehlt unter anderem diese Verfahren [21, S. 44 ff.]. Der Vorteil von RSA ist, dass er

sowohl für die Verschlüsselung, als auch für die Signatur verwendet werden kann, weshalb in dem Code-Beispiel dieses verwendet wird. Elliptische Kurven wurden in dieser Arbeit, wie oben genannt, nicht bearbeitet. Auch die digitale Signatur mittels RSA benötigt ein Padding. `Cryptography.io` bietet hier nur das Probabilistic Signature Scheme (PSS) und das ältere PKCS#1v1.5 Padding an. Es wird das von Moriarty et al. [23] definierte PSS Padding ausgewählt, welches auch von BSI empfohlen wird [21, S. 45]. Auch dieses Padding benötigt eine MGF und es wird auch hier wieder wie oben die MGF1 Funktion verwendet.

Hashfunktionen Das NIST genehmigt für alle kryptographischen Anwendungen Hashfunktionen, welche in [32] und [14] definiert sind [3, S. 23]. In diesen Dokumenten werden die Secure Hash Algorithm (SHA) Funktionen mit verschiedenen Ausgabegrößen definiert. Die Funktionen der neuen SHA-3 Familie können leider noch nicht verwendet werden, da sie in der Programmbibliothek noch nicht vorhanden sind. Das BSI empfiehlt die SHA Funktionen, jedoch nur ab der Ausgabegröße von 256 *bit* [21, S. 39 f.]. Die Auswahl der verschiedenen Ausgabegrößen erfolgt in den einzelnen Code-Beispielen.

3.3 Code-Beispiele

In diesem Abschnitt werden die Code-Beispiele, die im Zuge dieser Arbeit erstellt wurden, vorgestellt. Es gibt die Kategorien asymmetrische Verschlüsselung, symmetrische Verschlüsselung, Speicherung von Schlüsselpaaren, digitale Signaturen und Hashing. Während es zu den anderen Kategorien je ein Beispiel gibt, wird die symmetrische Verschlüsselung nochmals unterteilt in passwortbasierte, schlüsselbasierte und Dateiverschlüsselung. Alle Beispiele können in dem GitHub Projekt der Arbeit[7] in dem Ordner *src/cryptoexamples* oder auf der Website von CryptoExamples[8] abgerufen werden.

3.3.1 Gemeinsamkeiten der Code-Beispiele

In den Code-Beispielen werden die Richtlinien und das Konzepte aus den vorangegangenen Abschnitten angewandt. Es werden nicht alle Beispiele im kompletten Umfang gezeigt, weshalb hier auf die Teile der Beispiele eingegangen wird, die überall gleich sind. Das erste Beispiel in Listing 3.1 dient hier als Vorlage. Am Anfang eines Beispiels stehen immer die Importe der Programmbibliotheken, die in dem Beispiel benötigt werden. Dabei stehen im ersten Abschnitt die allgemeinen Importe und im zweiten Abschnitt die der kryptographischen Programmbibliothek `cryptography.io`, die verwendet werden. Daraufhin folgt, durch einen Kommentar eingeleitet, die Initialisierung des *loggers*, welcher in den Beispielen hauptsächlich verwendet wird, um das Ergebnis zu präsentieren. Die Vorführ-Funktion, welche laut Richtlinien immer mit *demonstrate_* beginnt, ist selbstverständlich bei jedem Beispiel anders. Am Schluss wird die implementierte Vorführ-Funktion ausgeführt, wenn das Beispiel als Script verwendet wird.

[7] `https://github.com/cryptoexamples/python-cryptography-cryptoexamples` (abgerufen 2018-09-19)
[8] `https://www.cryptoexamples.com/` (abgerufen 2018-09-19)

3.3.2 Passwortbasierte, symmetrische Textverschlüsselung

Das erste Beispiel ist eine passwortbasierte, symmetrische Verschlüsselung von Text und wird in Listing 3.1 gezeigt. Die Vorführ-Funktion kann zwei Parameter annehmen. Der Erste ist der Text, der verschlüsselt werden soll. Der Zweite ist das Passwort, das zum Verschlüsseln verwendet werden soll. Dieses ist aber optional, weshalb es einen vorgegebenen *String* enthält, der leer ist. Der ganze Code der Funktion ist von einem `try: except:` umschlossen, welches *Exceptions* abfängt. Im Ersten Abschnitt des Codes wird ein sicheres, vierzig stelliges Passwort generiert, wenn keins, oder ein Leeres als Parameter übergeben wurde. Das Passwort wird erstellt indem aus einem Alphabet, welches alle Groß- und Kleinbuchstaben sowie alle Ziffern enthält, zufällig vierzig Zeichen gezogen werden und diese aneinander gereiht werden. Es wird das `random.SystemRandom` Modul verwendet, welches zufällige Bytes generiert und deshalb eine zufällige Auswahl treffen kann. Als Alternative wäre hier das `secrets` Modul favorisiert, aber da das Beispiel auch für Python 2.7 funktionieren soll und das `secrets` Modul erst ab Python 3.6 zur Verfügung steht, wird es nicht verwendet. Intern verwenden beide Module den sicheren Zufallszahlengenerator `os.urandom()`, der für Kryptographie geeignet ist. `os.urandom()` verwendet eine vom Betriebssystem zur Verfügung gestellte Quelle für Zufälligkeit. Wenn keine zufälligen *bytes* zur Verfügung stehen, kann es sein, dass die Methode das Programm blockiert. Das Passwort wird nun mit der *UTF-8* Kodierung kodiert, damit es als *bytes* Repräsentation vorliegt und damit im übernächsten Schnitt benutzt werden kann. Es wird auch ein *salt* generiert, welche für die Ableitung des Schlüssels benötigt wird. Die Länge des *salts* sollte mit der Ausgabelänge der Hashfunktion übereinstimmen [31]. Deshalb wird ein Wert von 64 *bytes* ausgewählt, welcher mit der Ergebnislänge der Hashfunktion übereinstimmt, die im Folgenden verwendet wird. Auch hier wird der sichere Nummerngenerator verwendet.

Es wird nun ein Schlüssel aus dem Passwort abgeleitet. Dabei wird das PBKDF2 Verfahren mit HMAC verwendet. Das Objekt benötigt mehrere Parameter, um initialisiert zu werden. Als Hashfunktion wird SHA-512 verwendet. An dieser Stelle wurden SHA-1, SHA-256 und SHA-512 in Betracht gezogen. Die NIST Definition schreibt nur vor, dass eine anerkannte Hashfunktion verwendet werden soll [37, S. 7]. SHA-1 gilt als veraltet und SHA-256 wäre zwar auch möglich und auch schneller, jedoch wird SHA-512 verwendet, da an dieser Stelle die Sicherheit wichtiger ist. Die Länge des Schlüssels wird in *bytes* angegeben. Es werden 32 *bytes* gewählt, was einem 256 *bit* Schlüssel entspricht. Warum AES verwendet wird, wird in 3.2.2 beschrieben. Wie dort genannt, sind nur die Werte 128, 192 und 256 *bit* empfohlen. Es wird hier die sicherste Variante mit 256 *bit* gewählt. Als Wert für die Anzahl der Iterationen wird 10.000 verwendet. NIST schreibt, dass der Wert der Iterationen größtmöglich gewählt werden soll, sodass die erforderliche Zeit für den Benutzer noch akzeptabel ist [37, S. 6]. Dabei wird 1.000 Iterationen als Minimum vorgegeben und 10.000.000 Iterationen für besonders sicherheitskritische Systeme empfohlen. Der Wert sollte für jedes System individuell bestimmt werden. Der hier vorgegebene Wert von 10.000 wird gewählt, um sich vom Minimum abzusetzen, das verwendete System aber nicht zu überfordern. Nun kann ein Schlüssel von dem vorgegebenen Passwort sicher abgeleitet werden.

Es wird der als Parameter übergebene Text verschlüsselt. Dazu wird zuerst eine *nonce* mittels sicherem Zufallszahlengenerator erstellt. Laut NIST, und BSI, welches dessen Aussage übernimmt, sollte die *nonce* bzw. der Initialisierungsvektor für den GCM Modus auf 12 *bytes* festgelegt werden [13, S. 8][21, S. 24]. Diese Größe wird hier verwendet. Nachdem das `AESGCM` Objekt mit dem abgeleiteten Schlüssel initialisiert wurde, kann der mit *UTF-8* kodierte Text verschlüsselt werden.

Dabei kommt noch die zuvor erstellte *nonce* zum Einsatz. Die Methode `aesgcm.encrypt()` liefert ihr Ergebnis in *bytes* zurück. Diese werden noch mit dem *BASE64* Verfahren kodiert, um einen Text zu schaffen, der gelesen werden kann. Nun wurde der übergebene Text erfolgreich verschlüsselt.

Für die Entschlüsselung des Textes benötigt man den Schlüssel und die *nonce*, mit dem der Text verschlüsselt wurde. Da mit dem Schlüssel schon ein Objekt des `AESGCM` angelegt wurde, kann dieses hier wieder verwendet werden. Für die Entschlüsselung muss nur noch der soeben kodierte Text mit Hilfe des *BASE64* Verfahrens wieder dekodiert werden. Auch das Ergebnis der `aesgcm.decrypt()` Methode liegt in *bytes* vor und muss, mit dem zu Beginn verwendeten *UFT-8*, dekodiert werden. Anschließend kann mit einer „==“ Überprüfung nachgewiesen werden, dass der ursprüngliche Text mit dem ver- und wieder entschlüsseltem Text übereinstimmt. Dies wird mit dem *logger* ausgegeben.

```
1  import base64
2  import logging
3  import os
4  from random import SystemRandom
5
6  from cryptography.exceptions import AlreadyFinalized
7  from cryptography.exceptions import InvalidTag
8  from cryptography.exceptions import UnsupportedAlgorithm
9  from cryptography.hazmat.backends import default_backend
10 from cryptography.hazmat.primitives import hashes
11 from cryptography.hazmat.primitives.ciphers.aead import AESGCM
12 from cryptography.hazmat.primitives.kdf.pbkdf2 import PBKDF2HMAC
13
14 # set up logger
15 logging.basicConfig(level=logging.INFO)
16 logger = logging.getLogger(__name__)
17
18
19 def demonstrate_string_encryption_password_based(plain_text, password=""):
20     """
21     Example for encryption and decryption of a string in one method.
22     - Random password generation using strong secure random number generator
23     - Random salt generation using OS random mode
24     - Key derivation using PBKDF2 HMAC SHA-512
25     - AES-256 authenticated encryption using GCM
26     - BASE64 encoding as representation for the byte-arrays
27     - UTF-8 encoding of Strings
28     - Exception handling
29     """
30     try:
31         # GENERATE password (not needed if you have a password already)
32         if not password:
33             alphabet =
   "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789"
34             password = "".join(SystemRandom().choice(alphabet) for _ in range(40))
```

```
35          password_bytes = password.encode('utf-8')
36
37          # GENERATE random salt (needed for PBKDF2HMAC)
38          salt = os.urandom(64)
39
40          # DERIVE key (from password and salt)
41          kdf = PBKDF2HMAC(
42              algorithm=hashes.SHA512(),
43              length=32,
44              salt=salt,
45              iterations=10000,
46              backend=default_backend()
47          )
48          key = kdf.derive(password_bytes)
49
50          # GENERATE random nonce (number used once)
51          nonce = os.urandom(12)
52
53          # ENCRYPTION
54          aesgcm = AESGCM(key)
55          cipher_text_bytes = aesgcm.encrypt(
56              nonce=nonce,
57              data=plain_text.encode('utf-8'),
58              associated_data=None
59          )
60          # CONVERSION of raw bytes to BASE64 representation
61          cipher_text = base64.urlsafe_b64encode(cipher_text_bytes)
62
63          # DECRYPTION
64          decrypted_cipher_text_bytes = aesgcm.decrypt(
65              nonce=nonce,
66              data=base64.urlsafe_b64decode(cipher_text),
67              associated_data=None
68          )
69          decrypted_cipher_text = decrypted_cipher_text_bytes.decode('utf-8')
70
71          logger.info("Decrypted and original plain text are the same: %s",
72                      decrypted_cipher_text == plain_text)
73      except (UnsupportedAlgorithm, AlreadyFinalized, InvalidTag):
74          logger.exception("Symmetric encryption failed")
75
76
77  if __name__ == '__main__':
78      # demonstrate method
79      demonstrate_string_encryption_password_based(
80          "Text that is going to be sent over an insecure channel and must be "
```

```
81          "encrypted at all costs!", "")
```

Listing 3.1: Code-Beispiel für die passwortbasierte, symmetrische Textverschlüsselung

3.3.3 Schlüsselbasierte, symmetrische Textverschlüsselung

Das zweite Beispiel ist die schlüsselbasierte, symmetrische Textverschlüsselung. Das Beispiel unterscheidet sich nur in wenigen Punkten von der passwortbasierten, symmetrischen Textverschlüsselung. Die wichtigen Änderungen werden in Listing 3.2 gezeigt. In der Vorführ-Funktion wird nur ein Parameter, und zwar der zu verschlüsselnde Text, übergeben. Der Teil, in dem der Schlüssel aus einem Passwort abgeleitet wird, entfällt in diesem Beispiel. Stattdessen wird ein Schlüssel der benötigten Größe 256 *bits* generiert. Dafür wird die Methode `generate_key()` verwendet. Diese verwendet intern wiederum die schon bekannte `os.urandom()` Methode, welche zufällige *bytes* für kryptographische Zwecke liefert. Der so generierte Schlüssel kann nun für die Ver- und Entschlüsselung verwendet werden. Der Rest des Beispiels entspricht dem obigen.

```
13  def demonstrate_string_encryption_key_based(plain_text):
14      """
15      Example for encryption and decryption of a string in one method.
16      - Random key generation using OS random mode
17      - AES-256 authenticated encryption using GCM
18      - BASE64 encoding as representation for the byte-arrays
19      - UTF-8 encoding of Strings
20      - Exception handling
21      """
22      try:
23          # GENERATE key
24          key = AESGCM.generate_key(bit_length=256)
```

Listing 3.2: Code-Beispiel für die schlüsselbasierte, symmetrische Textverschlüsselung

3.3.4 Passwortbasierte, symmetrische Dateiverschlüsselung

Dieses Beispiel für passwortbasierte, symmetrische Dateiverschlüsselung ist ebenfalls ähnlich der passwortbasierten, symmetrischen Textverschlüsselung. Das Beispiel kann in Listing 3.3 gefunden werden. Die Vorführ-Funktion bekommt statt dem Text den Dateinamen der Datei, die verschlüsselt werden soll, als Parameter übergeben. Die Datei wird zu Beginn geöffnet und der Inhalt wird eingelesen und kann dann wie bei der Textverschlüsselung verwendet werden. Wenn ein Schlüssel abgeleitet und der Text erfolgreich verschlüsselt wurde, wird wie in den Zeilen 62-64 zu sehen, das Ergebnis als Datei gespeichert. Dabei wird, wie man an dem `'wb'` Parameter sehen kann der *bytemode* verwendet, die *bytes* werden also direkt in die Datei geschrieben und müssen nicht vorher noch kodiert werden. Das Lesen der Datei verläuft ähnlich und danach kann, wie bei der Textverschlüsselung, mit dem Entschlüsseln fortgefahren werden.

```
18  def demonstrate_file_encryption_password_based(plain_text_file_name, password=""):
19      """
20      Example for encryption and decryption of a file in one method.
```

```
21      - Random password generation using strong secure random number generator
22      - Random salt generation using OS random mode
23      - Key derivation using PBKDF2 HMAC SHA-512
24      - AES-256 authenticated encryption using GCM
25      - UTF-8 encoding of Strings
26      - Exception handling
27      """
28      with open(plain_text_file_name, 'r') as plain_text_file:
29          plain_text = plain_text_file.read()
30
31      try:
32          # GENERATE password (not needed if you have a password already)
33          if not password:
34              alphabet =
    "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789"
35              password = "".join(SystemRandom().choice(alphabet) for _ in range(40))
36          password_bytes = password.encode('utf-8')
37
38          # GENERATE random salt (needed for PBKDF2HMAC)
39          salt = os.urandom(64)
40
41          # DERIVE key (from password and salt)
42          kdf = PBKDF2HMAC(
43              algorithm=hashes.SHA512(),
44              length=32,
45              salt=salt,
46              iterations=10000,
47              backend=default_backend()
48          )
49          key = kdf.derive(password_bytes)
50
51          # GENERATE random nonce (number used once)
52          nonce = os.urandom(12)
53
54          # ENCRYPTION
55          aesgcm = AESGCM(key)
56          cipher_text_bytes = aesgcm.encrypt(
57              nonce=nonce,
58              data=plain_text.encode('utf-8'),
59              associated_data=None
60          )
61
62          # WRITE to file
63          with open("res/encrypted_file.enc", 'wb') as encrypted_file:
64              encrypted_file.write(cipher_text_bytes)
65
66          # READ from file
```

```
67          with open("res/encrypted_file.enc", 'rb') as encrypted_file:
68              cipher_file_content = encrypted_file.read()
69
70          # DECRYPTION
71          decrypted_cipher_text_bytes = aesgcm.decrypt(
72              nonce=nonce,
73              data=cipher_file_content,
74              associated_data=None
75          )
76          decrypted_cipher_text = decrypted_cipher_text_bytes.decode('utf-8')
77
78          logger.info("Decrypted and original plain text are the same: %s",
79                      decrypted_cipher_text == plain_text)
```

Listing 3.3: Code-Beispiel für die passwortbasierte, symmetrische Dateiverschlüsselung

3.3.5 Asymmetrische Textverschlüsselung

Dieses Beispiel behandelt die asymmetrische Textverschlüsselung mittels des RSA Verfahrens. Das Beispiel wird in Listing 3.4 gezeigt. Der Beginn des Beispiels stimmt mit den anderen überein und wurde im ersten Paragraph behandelt. Die Vorführ-Funktion erwartet nur einen Parameter und zwar den Text, der verschlüsselt werden soll. Zuerst muss ein Schlüsselpaar erstellt werden. Dazu wird ein privater Schlüssel erzeugt, aus dem wiederum der öffentliche Schlüssel abgeleitet werden kann. Die Methode `rsa.generate_private_key()` erwartet mehrere Parameter. Die Schlüsselgröße wird auf 4096 *bit* festgelegt. Giry ist Autor der Seite keylength.com [17], welche die Informationen verschiedener Quellen für empfohlene Schlüssellängen vergleicht. Je nachdem ob man das Jahr, bis zu dem die Schlüssel sicher sein sollten, auf 2022 oder 2030 festlegt, bekommt man verschiedene Empfehlungen im Bereich von 1446 bis 15360 *bit* für die Schlüssellänge. Das BSI verlangt eine Schlüssellänge von mindestens 3000 *bit*, wenn diese auch bis nach 2022 sicher sein sollte. Das NIST schreibt eine Mindestlänge von 2048 *bit* vor [4]. Sie setzen aber auch die Stärke eines Schlüssels für die symmetrische Verschlüsselung mit der Länge von 256 *bit*, so wie sie in den vorigen Beispielen verwendet wird, mit einer Schlüssellänge von 15360 *bit* für das RSA Verfahren gleich [3, S. 53]. Ferguson et al. [16] schreiben eine Mindestlänge von 2048 *bit* vor, empfehlen allerdings 4096 *bit* zu verwenden und die Software so zu gestalten, dass auf 8192 *bit* aufgerüstet werden könnte [16, S. 203]. Auf Grund dieser Informationen wurde entschieden, die Schlüssellänge auf 4096 *bit* festzulegen. Für den öffentlichen Exponenten verwendet man Fermat-Zahlen, geläufig ist dabei 65537[9], wobei auch andere Zahlen möglichen wären.

Nachdem das Schlüsselpaar erstellt wurde, kann der übergebene Text, nachdem er mit *UTF-8* kodiert wurde, verschlüsselt werden. Dabei wird auf dem öffentlichen Schlüssel die Methode `encrypt()` ausgeführt. Diese benötigt noch ein Padding, um den Text zu verschlüsseln. In Abschnitt 3.2.2 wird erklärt wieso OAEP und damit verbunden MGF1 verwendet werden. Die PKCS #1 sagt, dass der vorgegebene Wert für die Hashfunktion, sowohl für die für OAEP als auch für MGF1, SHA-1

[9] `https://crypto.stackexchange.com/questions/3110/impacts-of-not-using-rsa-exponent-of-65537` (abgerufen 2018-09-19)

ist [23, S. 57]. Dieser ist jedoch nur festgesetzt, um Abwärtskompatibilität zu schaffen. Es wird empfohlen, einen sicheren Algorithmus, wie zum Beispiel SHA-256 oder SHA-512, zu verwenden [23, S. 65]. Das NIST verbietet sogar die Verwendung von SHA-1 für den kryptographischen Schutz von Regierungsinformationen [3, S. 54]. Aus diesem Grund wird für OAEP der SHA-512 und für MGF1, welcher weniger sicherheitsrelevant ist, der SHA-256 Algorithmus gewählt. Das Ergebnis der Methode liegt wieder als *bytes* vor und wird mit *BASE64* kodiert, um ein lesbares Ergebnis zu produzieren.

Für die Entschlüsselung wird nun auf dem privaten Schlüssel die Methode `decrypt()` ausgeführt. Es wird das gleiche Padding und die gleichen Parameter wie bei der Verschlüsselung verwendet. Zu beachten ist, dass der verschlüsselte Text erst wieder dekodiert wird. Abschließend wird das entschlüsselte Ergebnis mit *UTF-8* wieder dekodiert und kann mit dem ursprünglichen Text verglichen werden. Eine *logger* Ausgabe bestätigt das Ergebnis.

```
1  import base64
2  import logging
3
4  from cryptography.exceptions import UnsupportedAlgorithm
5  from cryptography.hazmat.backends import default_backend
6  from cryptography.hazmat.primitives import hashes
7  from cryptography.hazmat.primitives.asymmetric import padding
8  from cryptography.hazmat.primitives.asymmetric import rsa
9
10 # set up logger
11 logging.basicConfig(level=logging.INFO)
12 logger = logging.getLogger(__name__)
13
14
15 def demonstrate_asymmetric_string_encryption(plain_text):
16     """
17     Example for asymmetric encryption and decryption of a string in one method.
18     - Generation of public and private RSA 4096 bit keypair
19     - RSA encryption and decryption of text using OAEP and MGF1 padding
20     - BASE64 encoding as representation for the byte-arrays
21     - UTF-8 encoding of Strings
22     - Exception handling
23     """
24     try:
25         # GENERATE NEW KEYPAIR
26         private_key = rsa.generate_private_key(
27             public_exponent=65537,
28             key_size=4096,
29             backend=default_backend()
30         )
31         public_key = private_key.public_key()
32
33         # ENCRYPTION
34         cipher_text_bytes = public_key.encrypt(
```

```
35              plaintext=plain_text.encode('utf-8'),
36              padding=padding.OAEP(
37                  mgf=padding.MGF1(algorithm=hashes.SHA256()),
38                  algorithm=hashes.SHA512(),
39                  label=None
40              )
41          )
42          # CONVERSION of raw bytes to BASE64 representation
43          cipher_text = base64.urlsafe_b64encode(cipher_text_bytes)
44
45          # DECRYPTION
46          decrypted_cipher_text_bytes = private_key.decrypt(
47              ciphertext=base64.urlsafe_b64decode(cipher_text),
48              padding=padding.OAEP(
49                  mgf=padding.MGF1(algorithm=hashes.SHA256()),
50                  algorithm=hashes.SHA512(),
51                  label=None
52              )
53          )
54          decrypted_cipher_text = decrypted_cipher_text_bytes.decode('utf-8')
55
56          logger.info("Decrypted and original plain text are the same: %s",
57                      decrypted_cipher_text == plain_text)
58      except UnsupportedAlgorithm:
59          logger.exception("Asymmetric encryption failed")
60
61
62  if __name__ == '__main__':
63      # demonstrate method
64      demonstrate_asymmetric_string_encryption(
65          "Text that is going to be sent over an insecure channel and must be "
66          "encrypted at all costs!")
```

Listing 3.4: Code-Beispiel für asymmetrische Textverschlüsselung

3.3.6 Digitale Signatur

Das Beispiel für digitale Signatur verwendet ebenfalls das RSA Verfahren. Es hat also gewisse Ähnlichkeiten zu dem vorangegangenen Beispiel. Die Zeilen, bis das Schlüsselpaar erstellt wurde, sind gleich. Der restliche, in diesem Abschnitt beschriebene, Code wird in Listing 3.5 gezeigt. Es wird statt der `encrypt()` Methode die `sign()` Methode ausgeführt, um den Text zu signieren. Dabei wird der in *UTF-8* kodierte Text, ein Padding und eine Hashfunktion übergeben. Als Padding wird, wie in Abschnitt 3.2.2 begründet, das PSS Verfahren und damit verbunden MGF1 gewählt. Die Wahl der Hashfunktionen ist ähnlich wie bei der asymmetrischen Verschlüsselung. Die PKCS #1 gibt als vorgegebenen Wert den SHA-1 vor, empfiehlt jedoch einen sichereren Algorithmus, wie zum Beispiel SHA-256 oder SHA-512, zu verwenden [23, S. 33, 60, 66]. Weiter empfehlen sie, dass

der verwendete MGF1 die selbe Hashfunktion verwenden sollte wie das PSS Verfahren. Es wird in diesem Fall der schnellere SHA-256 gewählt. Nachdem die Signatur erstellt wurde, muss diese mit *BASE64* kodiert werden, um lesbar gemacht zu werden. Um die Signatur und den dazugehörigen Text zu überprüfen, wird ein weiteres `try: except:` Konstrukt eingeführt. Die `verify()` Methode wirft einen `InvalidSignature` (deutsch: ungültige Signatur) Fehler, wenn die Signatur nicht korrekt ist. Es werden die selben Parameter wie bei der `sign()` Methode verwendet und zusätzlich die Signatur übergeben. Zuletzt wird durch eine *logger* Ausgabe bestätigt, ob die Signatur korrekt ist.

```
34          # SIGN DATA/STRING
35          signature = private_key.sign(
36              data=plain_text.encode('utf-8'),
37              padding=padding.PSS(
38                  mgf=padding.MGF1(hashes.SHA256()),
39                  salt_length=padding.PSS.MAX_LENGTH
40              ),
41              algorithm=hashes.SHA256()
42
43          )
44          logger.info("Signature: %s", base64.urlsafe_b64encode(signature))
45
46          # VERIFY JUST CREATED SIGNATURE USING PUBLIC KEY
47          try:
48              public_key.verify(
49                  signature=signature,
50                  data=plain_text.encode('utf-8'),
51                  padding=padding.PSS(
52                      mgf=padding.MGF1(hashes.SHA256()),
53                      salt_length=padding.PSS.MAX_LENGTH
54                  ),
55                  algorithm=hashes.SHA256()
56              )
57              is_signature_correct = True
58          except InvalidSignature:
59              is_signature_correct = False
60
61          logger.info("Signature is correct: %s", is_signature_correct)
```

Listing 3.5: Code-Beispiel für eine digitale Signatur

3.3.7 Speicherung von Schlüsselpaaren

Bei der Speicherung von Schlüsselpaaren geht es darum, ein Schlüsselpaar mit einem Passwort verschlüsselt abzuspeichern. Dabei wird ein RSA Schlüsselpaar genommen, welches zum Beispiel für die digitale Signatur oder die asymmetrische Verschlüsselung verwendet werden kann. Das Beispiel wird in Listing 3.6 gezeigt. Zu Beginn des Beispiels wird wie bei der passwortbasierten, symmetrischen Textverschlüsselung ein Passwort generiert. Dabei wird nur ein zwanzig stelliges Passwort generiert und es wird auch mit dem *logger* ausgegeben. Dies dient dazu, dass man sich

dieses Passwort anschauen und eventuell auch merken kann. Auf sicherheitskritischen Systemen, bei denen das Passwort extern gespeichert wird, sollte man das Passwort vierzig stellig machen und die *logger* Ausgabe vermeiden. Anschließend wird ein Schlüsselpaar generiert. Das erfolgt genau gleich wie bei der asymmetrischen Textverschlüsselung. Details, unter anderem die Wahl der Parameter, werden in Abschnitt 3.3.5 beschrieben.

Nachdem ein Schlüsselpaar erstellt wurde, muss dieses serialisiert werden. Die Serialisierung des privaten Schlüssels erfordert drei Parameter. Für die Kodierung stehen in der Programmbibliothek das Privacy Enhanced Mail (PEM), Distinguished Encoding Rules (DER) und das OpenSSH Format zur Verfügung. Während DER eine binäre Kodierung ist, ist das PEM Format *BASE64* kodiert. Es wäre wünschenswert gewesen, das neuere OpenSSH Format zu verwenden, aber das funktioniert mit dem privaten Schlüssel nicht. Das PEM Format hat den Vorteil, dass es schon lange in Verwendung ist und wird an dieser Stelle ausgewählt. Als Format stehen nur das PKCS #8 und das traditionelle OpenSSL Format zur Verfügung. Das PKCS #8 ist das modernere Format und wird deshalb verwendet. Zusätzlich muss noch der Algorithmus für die Verschlüsselung gewählt werden. Hier steht nur die Methode `BestAvailableEncryption()` zur Verfügung[10]. Es werde die beste verfügbare Verschlüsselung verwendet, die das *Backend* des Schlüssels kann. Für die Serialisierung des öffentlichen Schlüssels wird die selbe Kodierung und das typische Format für öffentliche Schlüssel verwendet. Es wäre auch hier OpenSSH wünschenswert, aber beide Schlüssel sollten mit der selben Kodierung serialisiert werden.

Die so erhaltenen Schlüssel werden als *bytes* in zwei getrennte Dateien geschrieben. Beim Laden der Dateien werden die für das Laden von Schlüsseln im PEM Format vorgesehenen Methoden verwendet. Anschließend werden die Schlüssel nur noch auf ihre Richtigkeit geprüft, also ob das Schlüsselpaar nach dem Speichern in der Datei noch dasselbe ist.

```
24      try:
25          # GENERATE password (not needed if you have a password already)
26          if not password:
27              alphabet =
    "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789"
28              password = "".join(SystemRandom().choice(alphabet) for _ in range(20))
29          logger.info(password)
30          password_bytes = password.encode('utf-8')
31
32          # GENERATE NEW KEYPAIR
33          private_key = rsa.generate_private_key(
34              public_exponent=65537,
35              key_size=4096,
36              backend=default_backend()
37          )
38          public_key = private_key.public_key()
39
40          # SERIALIZATION
41          pem_private = private_key.private_bytes(
```

[10] `https://cryptography.io/en/latest/hazmat/primitives/asymmetric/serialization/#serialization-encryption-types` (abgerufen 2018-09-19)

```
42              encoding=serialization.Encoding.PEM,
43              format=serialization.PrivateFormat.PKCS8,
44
    encryption_algorithm=serialization.BestAvailableEncryption(password_bytes)
45          )
46          pem_public = public_key.public_bytes(
47              encoding=serialization.Encoding.PEM,
48              format=serialization.PublicFormat.SubjectPublicKeyInfo
49          )
50
51          # WRITE KEYS
52          with open("res/private_key.pem", 'wb') as key_file:
53              key_file.write(pem_private)
54          with open("res/public_key.pem", 'wb') as key_file:
55              key_file.write(pem_public)
56
57          # LOAD KEYS
58          with open("res/private_key.pem", "rb") as key_file:
59              private_key_after = serialization.load_pem_private_key(
60                  data=key_file.read(),
61                  password=password_bytes,
62                  backend=default_backend()
63              )
64          with open("res/public_key.pem", "rb") as key_file:
65              public_key_after = serialization.load_pem_public_key(
66                  data=key_file.read(),
67                  backend=default_backend()
68              )
69
70          # CHECK whether keys are the same
71          private_before = private_key.private_bytes(
72              encoding=serialization.Encoding.PEM,
73              format=serialization.PrivateFormat.PKCS8,
74              encryption_algorithm=serialization.NoEncryption()
75          )
76          private_after = private_key_after.private_bytes(
77              encoding=serialization.Encoding.PEM,
78              format=serialization.PrivateFormat.PKCS8,
79              encryption_algorithm=serialization.NoEncryption()
80          )
81          public_before = pem_public
82          public_after = public_key_after.public_bytes(
83              encoding=serialization.Encoding.PEM,
84              format=serialization.PublicFormat.SubjectPublicKeyInfo
85          )
86
87          logger.info("Private Key before and after storage is the same: %s",
```

```
88                      private_before == private_after)
89          logger.info("Public Key before and after storage is the same: %s",
90                      public_before == public_after)
91      except (UnsupportedAlgorithm, ValueError, TypeError):
92          logger.exception("Asymmetric key storage failed")
```

Listing 3.6: Code-Beispiel für das Speichern von Schlüsselpaaren

3.3.8 Hashing

Das letzte Beispiel ist ein einfaches Hashing. Es wird in Listing 3.7 gezeigt. Als Hashfunktion wird hier SHA-512 verwendet. Das BSI empfiehlt die Hashfunktionen SHA-256, SHA-512/256, SHA-384 und SHA-512 [21, S. 39 f.]. Das NIST listet Hashfunktionen nach ihrer Stärke auf und erlaubt alles ab SHA-224 [3, S. 54]. Auf keylength.com [17] können verschiedene Empfehlungen zu der Größe von Hashfunktionen verglichen werden. Wenn man das Jahr, bis zu dem die Hashfunktionen sicher sein sollen, auf 2030 stellt, werden Vorschläge von 186 bis 512 *bit* für die Funktion gezeigt. Die Stichwahl fiel durch diese Informationen auf SHA-256 und SHA-512, wobei SHA-512 den Vorzug bekam, da mehr Wert auf Sicherheit als auf Geschwindigkeit gesetzt wird. In Zukunft wäre es von Vorteil, eine Hashfunktion der neu eingeführten Familie SHA-3 zu verwenden, welche in [14] definiert sind. Diese sind in der Programmbibliothek jedoch noch nicht vorhanden. Nachdem eine Hash-Instanz erstellt wurde, wird mit `digest.update()` der eingegebene Text hinzugefügt. An dieser Stelle könnten durch mehrfaches Ausführen der Methode mehrere Texte hinzugefügt werden. Nachdem die Funktion `digest.finalize()` ausgeführt wurde, wird das Ergebnis, welches in *bytes* vorliegt noch mit *BASE64* kodiert und so eine lesbare Ausgabe geschaffen. Diese wird schlussendlich durch den *logger* ausgegeben.

```
22      try:
23          # Get digest instance
24          digest = hashes.Hash(
25              algorithm=hashes.SHA512(),
26              backend=default_backend()
27          )
28
29          # CREATE HASH
30          digest.update(plain_text.encode('utf-8'))
31          hash_bytes = digest.finalize()
32
33          # CONVERT/ENCODE IN BASE64
34          hash_string = base64.urlsafe_b64encode(hash_bytes)
```

Listing 3.7: Code-Beispiel für Hashing

3.3.9 Erfüllung der Anforderungen

An die Code-Beispiele werden im Ziel der Arbeit im Abschnitt 1.3.1 mehrere Anforderungen gestellt. Sie sollen sicher, minimal, vollständig, kopierbar, ausführbar und getestet sein. Die Richtlinien bieten Vorgaben, durch welche die Anforderungen an die Code-Beispiele konkretisiert werden. In diesem Abschnitt wird darauf eingegangen, wie die Anforderungen erfüllt werden, und welche Konflikte dabei auftreten.

Sicher Die Sicherheit ist eine sehr wichtige Anforderung, weil die Code-Beispiele erstellt werden, um dem Nutzer dabei zu helfen, sichere Software zu entwickeln. Für die Sicherheit ist die Wahl der Algorithmen, Schlüssellänge und sonstiger Parameter von extremer Wichtigkeit. Darauf wurde geachtet und in Abschnitt 3.2.2 wird die Wahl der Algorithmen begründet und in den einzelnen Beispielen die Wahl der Schlüssellängen und Parameter erläutert. Durch die Verwendung von offiziellen und den Vergleich verschiedener Quellen wird die Sicherheit erhöht. Außerdem trägt die Wartung der Beispiel zur Sicherheit bei. Wenn sie aktuell gehalten werden, können Sicherheitslücken durch kritische Algorithmen verhindert werden. Die Anforderung der Sicherheit kollidiert an manchen Stellen mit der der Minimalität. Um ein Beispiel sicher zu machen, benötigt es in manchen Fällen mehr Code, und das verletzt somit die Minimalität. Zum Beispiel wird im Code-Beispiel in Abschnitt 3.3.2 sehr umständlich ein Passwort generiert. Für die Minimalität, und dank Python, wurde die Generierung auf wenige Zeilen beschränkt, dennoch ist es ein komplizierter Prozess.

Minimal Die Code-Beispiele sollen Minimal sein, damit dem Nutzer kein überflüssiger Code sondern nur das Wesentliche präsentiert wird. Die Minimalität wird dadurch erreicht, dass es für jedes Verfahren genau eine Datei und in dieser Datei genau eine Funktion gibt. Auch die Einteilung in einzelne Blöcke fördert die Minimalität, da dadurch der Code genau gegliedert ist und unnötige Teile entfernt werden können. Es werden die geforderten Verfahren ohne zusätzlichen Code angewendet. Außer der kryptographischen Programmbibliothek und der Standardbibliothek, welche aber immer vorhanden ist, werden keine zusätzlichen Abhängigkeiten benötigt. Die Minimalität kollidiert wie im vorigen Abschnitt beschrieben mit der Sicherheit. Sie wird aber auch sehr durch die Vollständigkeit eingeschränkt. Es gibt in den Code-Beispielen viele Kommentare, die zur Vervollständigung des Verständnisses des Codes beitragen, aber dadurch auch die Minimalität verletzen. Auch das Testen bzw. das Überprüfen des Ergebnisses schränkt die Minimalität ein. So wird zum Beispiel im Code der Schlüsselspeicherung in Abschnitt 3.3.7 eine lange Überprüfung gemacht, ob die Schlüssel nach dem Speichern dieselben sind wie vor dem Speichern. Es wird, nachdem die Schlüssel wieder aus den Dateien geladen wurden, diese, sowie die ursprünglichen Schlüssel, nochmals serialisiert, damit sie verglichen werden können. Dieser Prozess könnte in die Tests verschoben werde, dann kann dem Nutzer aber nicht mehr das Ergebnis präsentiert werden.

Vollständig Die Code-Beispiele müssen vollständig sein, damit sie mit minmalem Aufwand verwendet werden können. Der Nutzer soll die Möglichkeit haben einen vollständigen kryptographischen Prozess einzusehen. Die Beispiele sind vollständig, da alle Operationen durchgeführt werden. Der vollständige Prozess aller Teiloperationen lässt sich am Beispiel der passwortbasierten, symmetrischen Textverschlüsselung in Abschnitt 3.3.2 zeigen. Von der Passwortgenerierung, über die Schlüsselableitung, zur Verschlüsselung und dann wieder Entschlüsselung sind alle Operationen vorhanden. Vollständigkeit bedeutet auch, dass alle Parameter gesetzt sind und keine Werte mehr

eingetragen werden müssen. In diesem Fall hängt die Vollständigkeit stark mit der Kopierbar- und Ausführbar-Eigenschaft zusammen. Weitere Punkte, die zur Vollständigkeit beitragen, sind, dass unter anderem Tests existieren, *Exceptions* abgefangen werden und eine Überprüfung des Ergebnisses stattfindet. Wie oben beschrieben, schränken sich Vollständigkeit und Minimalität gegenseitig ein.

Kopierbar Die Code-Beispiele müssen kopierbar sein, damit der Nutzer sie auch verwenden kann. Dabei ist wichtig, dass auch rechtlich nichts im Weg steht. Die Anforderung kopierbar wird durch die Plattform als Website an sich garantiert und durch die Lizenz wie in Abschnitt 2.3 beschrieben ermöglicht. Durch die Gliederung des Codes in Blöcke können auch einzelne Teile des Codes unabhängig kopiert werden. Die Minimalität unterstützt die Eigenschaft kopierbar, da große, verschachtelte und überflüssige Code Fragmente das Kopieren erschweren. Vor allem wenn der Nutzer versucht das Kopierte in den eigenen Code zu integrieren. Die auführbar und getestet Eigenschaft sind Grundlage für das Kopieren. Wenn der Code nicht ausgeführt werden kann oder nicht richtig funktioniert, kann der Nutzer die Beispiele zwar kopieren, sie sind aber nutzlos für ihn. Auch die Einhaltung des Programmierstils und die Statische Code-Analyse helfen, dass der Code möglichst direkt vom Nutzer übernommen werden kann. Das Abfangen von *Exceptions* und das Benutzen des *loggers* schränken die Möglichkeit, die Beispiele in Einzelteilen zu kopieren, ein.

Ausführbar Die Code-Beispiele müssen ausführbar sein, damit der Nutzer sie direkt verwenden kann und keine Änderungen mehr vornehmen muss. Sie werden durch die `if __name__ == "__main__"` Konstruktion ausführbar gemacht. Eine Code-Beispiel Datei kann dann einfach als Script ausgeführt werden. Die Code-Beispiele hängen nur von den Modulen ab, die im Code importiert werden. Wenn die `cryptography.io` Programmbibliothek installiert ist, können die Dateien mit den unterstützten Python Versionen ausgeführt werden. Der Code ist auch ausführbar, da alle Parameter und Werte eingetragen sind. Vom Nutzer werden keine zusätzlichen Eingaben erwartet, auch wenn er den Code natürlich noch verändert werden kann. Hier hängt die Eigenschaft ausführbar mit der Vollständigkeit zusammen. Sie hängt aber auch stark mit der Getestet-Eigenschaft zusammen, da der Code nicht ausgeführt werden kann, wenn er Fehler enthält.

Getestet Als letzte Anforderung sollen die Code-Beispiele getestet sein. Dies wird gefordert, damit dem Nutzer funktionierender Code zur Verfügung gestellt werden kann und er davon ausgehen kann, dass keine Fehler vorhanden sind. Dies wird durch die im Verzeichnis *tests/* liegende Testdatei und das pytest[11] Modul garantiert. Weitere Informationen zu den Tests finden sich im folgenden Abschnitt.

3.4 Tests und Statische Code-Analyse

In diesem Abschnitt werden die erstellten *unit tests* vorgestellt und es wird auf die Statische Code-Analyse eingegangen. Bei dieser ist es möglich, eigene Regeln zu definieren. Diese Funktion wird genutzt, um einige der Richtlinien automatisch prüfen zu lassen und damit durchzusetzen.

[11] `https://docs.pytest.org` (abgerufen 2018-09-19)

3.4.1 Tests

Die Testdatei findet sich in Listing 3.8. Es wird das Werkzeug pytest verwendet. Dieses findet Testdateien automatisch, wenn sie die Form *test_*.py* oder **_test.py* haben. In diesen Dateien werden die Testfunktionen mit *test_* begonnen. Zu Beginn der Datei werden noch die Funktionen der einzelnen Code-Beispiele importiert. Dabei ist wichtig, dass der Ordner in dem die Beispiele liegen in der Umgebungsvariable *PYTHONPATH*[12] vorhanden ist. Für jedes Code-Beispiel existiert genau eine Testfunktion. Diese erwarten als Parameter *caplog*, welcher von dem Testwerkzeug bereitgestellt wird. *caplog* beinhaltet die Ausgaben des *loggers*, die während der Ausführung des Tests gemacht werden. Die Ausgaben enthalten, wie in Richtlinie 11 in Abschnitt 3.1.2 festgelegt, eine Ausgabe, welche aufzeigt, ob das Beispiel korrekt durchgeführt wurde oder nicht. Es wird im Test überprüft, ob die erwartete Ausgabe bei korrektem Ergebnis in den aufgezeichneten Ausgaben vorhanden ist. Die Code-Beispiele werden mit Hilfe der Ausgabe des *loggers* getestet, da durch diese Maßnahme der Code nicht verändert werden muss. Eine alternative Möglichkeit zum Testen wäre, dass jede Vorführ-Funktion zusätzlich zur Präsentation des Ergebnisses diesen auch als Rückgabewert zurück gibt. Das wäre aber unnötiger Code, der die Anforderung der Minimalität an die Code-Beispiele verletzen würde. In Abbildung 3.1 kann man die Testabdeckung der Tests mit 88,2% sehen. Mehr Testabdeckung wird nicht erreicht, da die Ausnahmefälle von *Exceptions* nicht getestet werden und der Code nach dem `if __name__ == "__main__"` Konstrukt ebenfalls nicht überprüft wird.

```
1  from example_asymmetric_key_storage import demonstrate_asymmetric_key_storage
2  from example_asymmetric_string_encryption import \
3      demonstrate_asymmetric_string_encryption
4  from example_hash import demonstrate_string_hash
5  from example_string_signature_rsa import demonstrate_signature_rsa
6  from example_symmetric_file_encryption_password_based import \
7      demonstrate_file_encryption_password_based
8  from example_symmetric_string_encryption_key_based import \
9      demonstrate_string_encryption_key_based
10 from example_symmetric_string_encryption_password_based import \
11     demonstrate_string_encryption_password_based
12
13
14 def test_string_encryption_password_based(caplog):
15     demonstrate_string_encryption_password_based(
16         "Text that is going to be sent over an insecure channel and must be
   encrypted at all costs!",
17         "")
18     assert "Decrypted and original plain text are the same: True" in caplog.text
19
20
21 def test_string_encryption_key_based(caplog):
22     demonstrate_string_encryption_key_based(
23         "Text that is going to be sent over an insecure channel and must be
   encrypted at all costs!")
```

[12] https://docs.python.org/3/using/cmdline.html#envvar-PYTHONPATH (abgerufen 2018-09-19)

```
24      assert "Decrypted and original plain text are the same: True" in caplog.text
25
26
27  def test_file_encryption_password_based(caplog):
28      demonstrate_file_encryption_password_based("res/plain_text_file.txt", "")
29      assert "Decrypted and original plain text are the same: True" in caplog.text
30
31
32  def test_string_hash(caplog):
33      # uses string: "Text that should be authenticated by comparing the hash of it!"
34      demonstrate_string_hash("Text that should be authenticated by comparing the
    hash of it!")
35      assert
    "jg0X629-SmdP0_LTHZV_3zXBrizM3_hptRZVIuTXSCtyaqAe0NB8KMld2qebBIXFS1yowCUpCPu93l_fPmKEXg=="
    in caplog.text
36
37
38  def test_asymmetric_string_encryption(caplog):
39      demonstrate_asymmetric_string_encryption(
40          "Text that is going to be sent over an insecure channel and must be "
41          "encrypted at all costs!")
42      assert "Decrypted and original plain text are the same: True" in caplog.text
43
44
45  def test_signature_rsa(caplog):
46      demonstrate_signature_rsa(
47          "Text that should be signed to prevent unknown tampering with its
    content.")
48      assert "Signature is correct: True" in caplog.text
49
50
51  def test_asymmetric_key_storage(caplog):
52      demonstrate_asymmetric_key_storage("")
53      assert "Private Key before and after storage is the same: True" in caplog.text
54      assert "Public Key before and after storage is the same: True" in caplog.text
```

Listing 3.8: Tests für die Code-Beispiele

3.4.2 Statische Code-Analyse

Statische Code-Analyse inspiziert und analysiert den Code noch bevor er ausgeführt wird. Dabei wird der Quellcode Prüfungen unterzogen, die Fehler und andere Probleme, wie zum Beispiel fehleranfälligen Code, entdecken. Die Statische Code-Analyse wird durchgeführt, um schon früh automatisch Fehler im Code zu finden. Außerdem können so Programmierstil und -richtlinien überprüft werden.

Für die Statische Code-Analyse werden drei Werkzeuge verwendet. Zum einen pylint[13], welches Statische Code-Analyse speziell für Python durchführt. Außerdem SonarQube[14] bzw., für das Analysieren von Onlineprojekten, die Variante SonarCloud[15]. Außerdem wird bandit[16] ausgeführt, ein Werkzeug, um geläufige Sicherheitsprobleme in Python Code zu finden.

pylint setzt unter anderem den Programmierstil PEP8[17] für Python durch. Es findet aber auch Fehler, *Code Smells* (deutsch: [schlechter] Geruch) und schlägt Maßnahmen vor, um den Code umzustellen. Ein großer Vorteil von pylint ist, dass man es für die eigenen Anforderungen verändern kann. Durch eine Konfigurationsdatei kann angepasst werden, was durch pylint alles überprüft werden soll. Außerdem kann man benutzerspezifische Abfragen selbst programmieren. Eigentlich wird in Python für Module, also Dateien, ein Kommentar erwartet, der diese beschreibt. Da diese Beschreibung in dem Methodenkommentar geschieht, ist das für unsere Beispiele nicht nötig. Neben dieser Abfrage wurde auch die Überprüfung auf duplizierten Code deaktiviert. Bei den Code-Beispielen ist es nur selbstverständlich, wenn es gleiche Zeilen gibt und es macht keinen Sinn das zu verhindern und den Code auszulagern. Die Code-Beispiele sollen nicht als Gesamtprojekt gesehen werden, sondern jedes Beispiel steht für sich. Auch die maximal erlaubte Anzahl an lokalen Variablen wurde geändert sowie andere kleine Anpassungen wurden gemacht, um dem Anwendungsfall gerecht zu werden.

Auch die Möglichkeit von benutzerspezifischen Abfragen wurde genutzt. Das Ziel ist es die generellen Richtlinien bzw. die speziell für Python angepassten Abwandlungen zu überprüfen. Wenn Entwickler neue Code-Beispiele schreiben, oder Bestehende warten und weiterentwickeln, soll die Code-Analyse Warnungen ausgeben, wenn die Richtlinien verletzt werden. Dabei sind manche Richtlinien, wie zum Beispiel alle des ersten Abschnitts 3.1.1, nicht möglich durch Statische Code-Analyse zu überprüfen. Die Tabelle 3.1 zeigt, welche Richtlinien möglich sind, und welche schon umgesetzt wurden. Dabei wird nur auf den Abschnitt 3.1.2, Erstellung von Beispielen, eingegangen, weil die anderen Richtlinien nicht automatisch umsetzbar sind. Die Richtlinien werden durch die Nummern in der ursprünglichen Liste referenziert.

Drei Richtlinien wurden schon umgesetzt und der entsprechende Code wird in Listing 3.9 gezeigt. Der Ordner in dem diese Datei liegt muss im PYTHONPATH vorhanden sein, damit es von pylint verwendet werden kann. Im oberen Teil des Codes werden die Fehler bzw. Warnungen definiert und die Ausgaben für den Nutzer festgelegt. Im unteren Teil werden Funktionen und Module iteriert, um die Richtlinien zu überprüfen. Die einfachste Überprüfung ist am Ende der `visit_module()` Methode. Es wird überprüft, ob der Name *logger* in den globalen Variablen vorhanden ist. Wenn dies nicht der Fall ist, wird die Nachricht für den Fehler zur Ausgabe hinzugefügt.

```
1  from pylint.checkers import BaseChecker
2  from pylint.interfaces import IAstroidChecker
3
4
5  class GuidelineChecker(BaseChecker):
6      __implements__ = IAstroidChecker
```

[13] https://www.pylint.org/ (abgerufen 2018-09-19)
[14] https://www.sonarqube.org/ (abgerufen 2018-09-19)
[15] https://sonarcloud.io/ (abgerufen 2018-09-19)
[16] https://bandit.readthedocs.io/ (abgerufen 2018-09-19)
[17] https://www.python.org/dev/peps/pep-0008/ (abgerufen 2018-09-19)

Richtlinie	Art der Prüfung
1.	automatisch (umgesetzt)
2.	automatisch
3.	automatisch (umgesetzt)
4.	automatisch (ob der Name korrekt ist, umgesetzt), manuell (ob die Funktion den kompletten Ablauf repräsentiert)
5.	automatisch
6.	automatisch (ob ein *docstring* existiert), manuell (ob die Informationen korrekt sind)
7.	manuell
8.	automatisch
9.	manuell
10.	manuell
11.	manuell
12.	automatisch
13.	automatisch (pylint)
14.	automatisch (pylint)
15.	manuell

Tabelle 3.1: Prüfung der Richtlinien

```
7
8       name = 'guidelines'
9       ONE_DEMONSTRATE = 'one-demonstrate'
10      EXAMPLE_PREFIX = 'example-prefix'
11      MISSING_LOGGER = 'missing-logger'
12
13      priority = -1
14      msgs = {
15          'C5000': ('Only one "demonstrate_" method allowed',
16                    ONE_DEMONSTRATE,
17                    'Refer to guidelines'),
18          'C5001': ('Files have to start with "example_"',
19                    EXAMPLE_PREFIX,
20                    'Refer to guidelines'),
21          'C5002': (
22              'A logger has to be specified. Should be named "logger" and be
    available as global variable',
23              MISSING_LOGGER,
24              'Refer to guidelines')
25      }
26
27      options = ()
28
```

```
29      def __init__(self, linter=None):
30          super(GuidelineChecker, self).__init__(linter)
31          self.demonstrate = 0
32
33      def visit_functiondef(self, node):
34          # CHECK if only one demonstrate_method exists
35          if node.name.startswith('demonstrate_'):
36              self.demonstrate += 1
37          if self.demonstrate > 1:
38              self.add_message(self.ONE_DEMONSTRATE, node=node)
39
40      def visit_module(self, node):
41          # reset demonstrate count for each file
42          self.demonstrate = 0
43          # check filename begins with example_
44          if not (node.name.startswith('example_') or node.name.startswith(
45                  'cryptoexamples.example_')):
46              self.add_message(self.EXAMPLE_PREFIX, node=node)
47          # check for logger
48          if 'logger' not in node.globals:
49              self.add_message(self.MISSING_LOGGER, node=node)
50
51
52  def register(linter):
53      """required method to auto register this checker"""
54      linter.register_checker(GuidelineChecker(linter))
```

Listing 3.9: Benutzerspezifische Code-Analyse für Richtlinien

SonarCloud erhält Berichte über die Tests und deren Testabdeckung von pytest. Es überprüft die Testabdeckung, duplizierten Code, *Code Smells*, die Sicherheit und andere Faktoren. Wie man in Abbildung 3.1 sehen kann, schlägt aktuell die Qualitätsprüfung fehl, da zu viel duplizierter Code besteht. Da der duplizierte Code gewollt ist, weil einige Beispiele ähnlich sind, kann das Fehlschlagen vernachlässigt werden. Auf die Testabdeckung wird in Abschnitt 3.4.1 eingegangen. Alle anderen Werte, wie *Code Smells*, Fehler oder Schwachstellen weisen keine Probleme auf. Als weitere Werte analysiert SonarCloud auch, wie viele Zeilen der Dateien wirklich Code sind. Bei insgesamt 509 Zeilen sind 313 davon Codezeilen und 109 Kommentare, der Rest leere Zeilen. 109 Kommentare sind im Verhältnis zu 313 Codezeilen relativ viel. Dies lässt sich damit begründen, dass die Code-Beispiele anschaulich sein sollen und der Nutzer diese verstehen muss. SonarCloud ist ein gutes Werkzeug, um die verschiedensten Metriken anzeigen zu lassen. Es bietet eine sehr übersichtliche Oberfläche an, die es dem Entwickler erlaubt, den eigenen Code zu analysieren und etwaige Probleme zu lösen. SonarCloud wird von allen Projekten von CryptoExamples verwendet.

Bandit ist ein Werkzeug speziell für Python, welches geläufige Sicherheitsprobleme im Code entdeckt. Probleme werden in die Sicherheitsklassen Undefiniert, Niedrig, Mittel und Hoch eingestuft. Bei der Überprüfung wurden nur drei Probleme der Klasse Niedrig gefunden. Erwartet hätte man keine, da die Code-Beispiele einen besonderen Fokus auf Sicherheit legen. Die Probleme werden in den Beispielen, in denen im Methodenkopf `password=""` steht, gefunden. Es wird angemerkt, dass

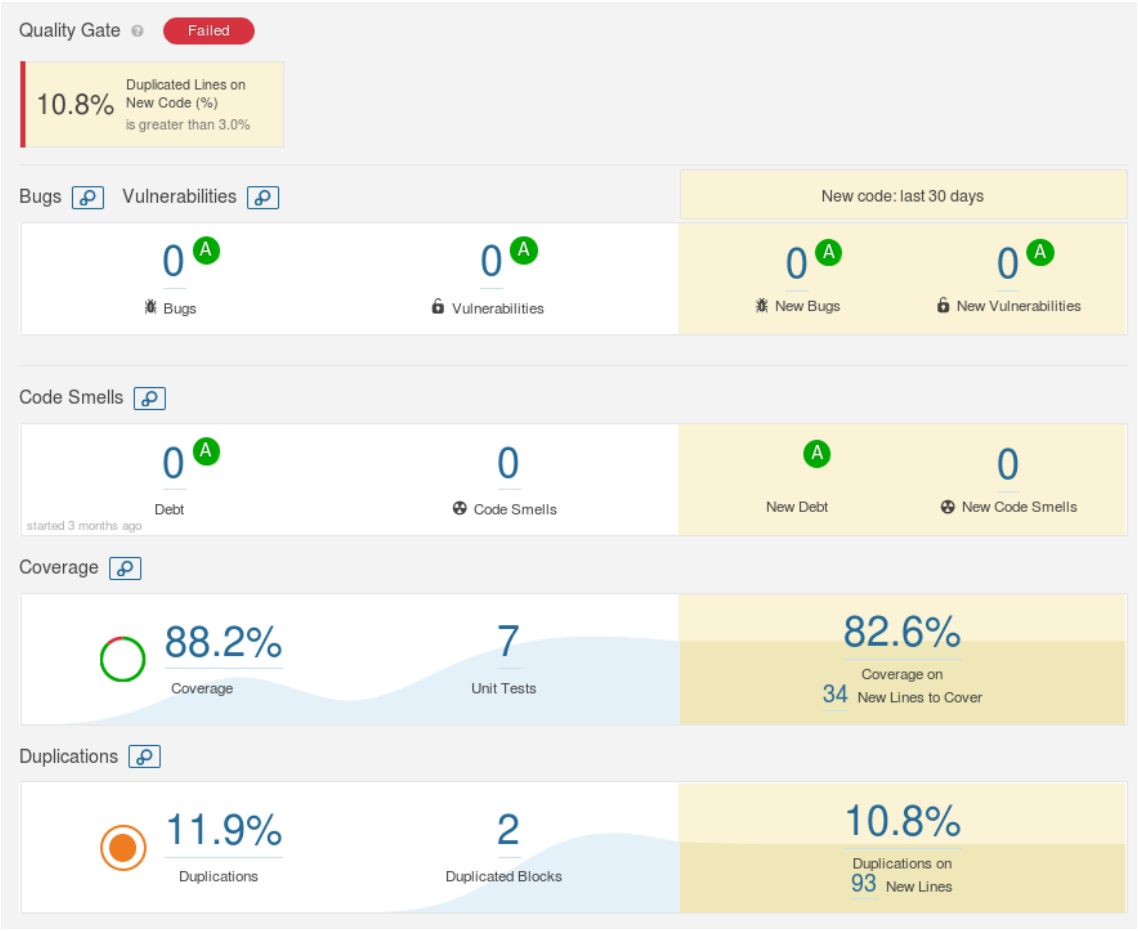

Abbildung 3.1: Übersichtsseite des Projekts bei SonarCloud

es möglich sei, dass an dieser Stelle ein fest implementiertes Passwort steht. Da bei allen diesen Funktionen aber eine Überprüfung dieses Parameters stattfindet, und wenn das Passwort leer ist ein sicheres generiert wird, können diese Meldungen ignoriert werden.

Zwei der genannten Werkzeuge, pylint und SonarCloud, und das für das Testen verwendete pytest werden online automatisch auf GitHub mit dem Werkzeug travis[18] ausgeführt. Da bei bandit ein Bericht generiert wird, den der Entwickler der Code-Beispiele anschauen sollte und notfalls auch ignorieren kann, wird dieses Werkzeug nicht mit travis ausgeführt, sondern muss lokal verwendet werden.

[18] `https://travis-ci.org/` (abgerufen 2018-09-19)

4 Zusammenfassung und Ausblick

4.1 Zusammenfassung

Zu Beginn der Arbeit wurde motiviert, warum CryptoExamples Sinn macht, und genauer, warum Code-Beispiele für die Programmiersprache Python sinnvoll sind. Anschließend wurden grundlegende kryptographische Verfahren beschrieben. Dabei wurden die Verfahren beschrieben, für welche später auch Code-Beispiele erstellt wurden. Aufgrund der unzureichenden Standardbibliothek von Python im Bereich der Kryptographie wurde eine externe kryptographische Programmbibliothek ausgewählt. Die Wahl viel auf das Open Source Projekt `cryptography.io`, welches sehr populär ist und auch aktiv weiterentwickelt wird.

Es wurden generelle Richtlinien für CryptoExamples entworfen, welche die Auswahl von Algorithmen und Parametern, die Erstellung von neuen Beispielen und die Weiterentwicklung und Wartung bestehender Beispiele umfassen. Durch die Richtlinien wurden die Anforderungen an die Code-Beispiele sicher, minimal, vollständig, kopierbar, ausführbar und getestet konkretisiert. Weiter konkretisiert wurde durch ein erstelltes Umsetzungskonzept. Dieses beschreibt, wie die Richtlinien auf die Programmiersprache Python umgesetzt werden können und welche Algorithmen für die einzelnen Verfahren gewählt wurden. Die Code-Beispiele wurden präsentiert und ausführlich erklärt. Dabei wurde auch die Wahl der einzelnen Parameter, also zum Beispiel wie groß ein Schlüssel sein sollte, begründet. Zuletzt wurde noch auf die Tests und die Statische Code-Analyse eingegangen. Es wurden *unit tests* für die einzelnen Beispiele erstellt und drei verschiedene Werkzeuge verwendet, um den Code zu analysieren. Hervorzuheben sind dabei die selbst erstellten Prüfungen, welche die Code-Beispiele auf Konformität mit den Richtlinien überprüfen.

4.2 Ausblick

CryptoExamples muss laufend weiter entwickelt werden und vor allem die Code-Beispiele müssen regelmäßig gewartet werden. Dabei sollten die in Abschnitt 3.1.3 definierten Richtlinien verwendet werden. Neben der Wartung der bestehenden Beispiele, müssen neue erstellt werden. Zum Beispiel wurde in dieser Arbeit nicht auf elliptische Kurven in der Kryptographie eingegangen. Außerdem wurde kein Code-Beispiel erstellt, dass den korrekten Vorgang bei einem Schlüsselaustausch implementiert. Bei diesen zwei fehlenden Techniken muss überprüft werden, ob ein eigenes Beispiel dafür erstellt werden sollte, oder ob sie ein bestehendes Verfahren ersetzen. Zum Beispiel könnte der RSA Algorithmus durch elliptische Kurven ersetzt werden. Weiter muss auf neue Verfahren geachtet werden. So wurde zum Beispiel die SHA-3 Familie von Hash-Funktionen definiert. Sie schließt bekannte Sicherheitslücken von SHA-1, welche prinzipiell auch auf SHA-2 anwendbar sind, obwohl diese Familie weiter als sicher gilt. Da diese in der Programmbibliothek `cryptography.io` noch nicht vorhanden sind, können sie aktuell noch nicht verwendet werden. Es sollte aber darauf geachtet

werden, diese möglichst schnell nachdem sie zur Verfügung stehen auch in CryptoExamples für Python einzubauen. Auf CryptoExamples gibt es die Möglichkeit für Experten, die Code-Beispiele zu begutachten. Dies muss noch für die Code-Beispiele für Python geschehen.

Bei der Statischen Code-Analyse wurden Prüfungen vorgestellt, welche die Richtlinien in den Code-Beispielen überprüfen und notfalls Änderungen fordern. Es bestehen jedoch nur wenige, einfache Prüfungen. Es müssen die in Tabelle 3.1 markierten Prüfungen implementiert werden. Dabei sollte auch darauf geachtet werden, wie man die Prüfungen für alle Code-Beispiele, also auch die der anderen Programmiersprachen, übernehmen könnte.

Es weiteres Manko, welches aber für CryptoExamples im Allgemeinen gilt, ist, dass die Code-Beispiele kein Ablaufdatum oder Ähnliches haben. Es wird nirgends festgelegt, bis wann diese Beispiele als sicher erachtet werden können. Somit steht auch CryptoExamples in der Gefahr, irgendwann veraltet zu sein und nur noch, oder wenigstens zum Teil, unsichere Beispiele zur Verfügung zu stellen. CryptoExamples benötigt auch noch weitere Beispiele anderer Programmiersprachen. Zum Zeitpunkt zu der diese Arbeit geschrieben wurde, gibt es nur Code-Beispiele für Python und für zwei Programmbibliotheken von Java. Weitere Beispiele sind in Arbeit, wie man auf dem GitHub Projekt von CryptoExamples sehen kann. Sobald Code-Beispiele in ein paar Sprachen vorliegen, sollte man sich um die Verbreitung von CryptoExamples kümmern. Dabei spielen Suchmaschinenoptimierungen eine Rolle, aber auch auf anderen Plattformen sollte für CryptoExamples geworben werden.

Literaturverzeichnis

[1] Y. Acar, M. Backes, S. Fahl, S. Garfinkel, D. Kim, M. L. Mazurek, C. Stransky. „Comparing the Usability of Cryptographic APIs“. In: *2017 IEEE Symposium on Security and Privacy (SP)*. Mai 2017, S. 154–171. DOI: 10.1109/SP.2017.52 (zitiert auf S. 15–17, 25, 26).

[2] Y. Acar, M. Backes, S. Fahl, D. Kim, M. L. Mazurek, C. Stransky. „You Get Where You're Looking for: The Impact of Information Sources on Code Security“. In: *2016 IEEE Symposium on Security and Privacy (SP)*. Mai 2016, S. 289–305. DOI: 10.1109/SP.2016.25 (zitiert auf S. 15, 17).

[3] E. Barker. *Recommendation for Key Management, Part 1: General*. Techn. Ber. Jan. 2016. DOI: 10.6028/NIST.SP.800-57pt1r4 (zitiert auf S. 31–33, 39, 40, 45).

[4] E. Barker, Q. Dang. *Recommendation for Key Management, Part 3: Application-Specific Key Management Guidance*. Techn. Ber. Jan. 2015. DOI: 10.6028/NIST.SP.800-57Pt3r1 (zitiert auf S. 39).

[5] A. Braga, R. Dahab. „Mining Cryptography Misuse in Online Forums“. In: *2016 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*. Aug. 2016, S. 143–150. DOI: 10.1109/QRS-C.2016.23 (zitiert auf S. 18).

[6] P. Carbonnelle. *PYPL PopularitY of Programming Language*. [Online; abgerufen 2018-09-19]. URL: https://pypl.github.io/PYPL.html (zitiert auf S. 16).

[7] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein. *Introduction to Algorithms, Third Edition*. 3rd. The MIT Press, 2009. ISBN: 9780262033848 (zitiert auf S. 24).

[8] *CryptoExamples GitHub*. [Online; abgerufen 2018-09-19]. URL: https://github.com/cryptoexamples (zitiert auf S. 19).

[9] *Cryptography*. [Online; abgerufen 2018-09-19]. URL: https://cryptography.io (zitiert auf S. 18, 25).

[10] J. Daemen, V. Rijmen. *The Design of Rijndael*. Springer-Verlag, 2002. ISBN: 9783642076466. DOI: 10.1007/978-3-662-04722-4 (zitiert auf S. 31).

[11] S. Das, V. Gopal, K. King, A. Venkatraman. *IV= 0 security: Cryptographic misuse of libraries*. Techn. Ber. 2014. URL: https://courses.csail.mit.edu/6.857/2014/files/18-das-gopal-king-venkatraman-IV-equals-zero-security.pdf (zitiert auf S. 17).

[12] *Developer Survey Results 2018*. [Online; abgerufen 2018-09-19]. URL: https://insights.stackoverflow.com/survey/2018/ (zitiert auf S. 16).

[13] M. Dworkin. *Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC*. Techn. Ber. Nov. 2007. DOI: 10.6028/NIST.SP.800-38D (zitiert auf S. 32, 34).

[14] M. J. Dworkin. *SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions*. Techn. Ber. Aug. 2015. DOI: 10.6028/NIST.FIPS.202 (zitiert auf S. 33, 45).

[15] M. J. Dworkin, E. Barker, J. R. Nechvatal, J. Foti, L. E. Bassham, E. Roback, J. F. Dray Jr. *Advanced Encryption Standard (AES)*. Techn. Ber. Nov. 2001. DOI: 10.6028/NIST.FIPS.197 (zitiert auf S. 31).

[16] N. Ferguson, B. Schneier, T. Kohno. *Cryptography Engineering: Design Principles and Practical Applications*. Wiley Publishing, 2010. ISBN: 978-0-470-47424-2. DOI: 10.1002/9781118722367 (zitiert auf S. 21–23, 31, 32, 39).

[17] D. Giry. *Cryptographic Key Length Recommendation*. [Online; abgerufen 2018-09-19]. URL: https://www.keylength.com/en/compare/ (zitiert auf S. 39, 45).

[18] *GitHut - A small place to discover languages in GitHub*. [Online; abgerufen 2018-09-19]. URL: https://githut.info/ (zitiert auf S. 16).

[19] B. Kaliski. *PKCS #5: Password-Based Cryptography Specification Version 2.0*. RFC 2898. Sep. 2000. DOI: 10.17487/rfc2898 (zitiert auf S. 24).

[20] *Keyczar*. [Online; abgerufen 2018-09-19]. URL: https://github.com/google/keyczar (zitiert auf S. 26).

[21] *Kryptographische Verfahren: Empfehlungen und Schlüssellängen*. Techn. Ber. [Online; abgerufen 2018-09-19]. Mai 2018. URL: https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Publikationen/TechnischeRichtlinien/TR02102/BSI-TR-02102.pdf?__blob=publicationFile&v=8 (zitiert auf S. 31–34, 45).

[22] K. Mindermann, S. Wagner. „Usability and Security Effects of Code Examples on Crypto APIs - CryptoExamples: A platform for free, minimal, complete and secure crypto examples“. In: *Proceedings of the 16th Annual Conference on Privacy, Security and Trust* (Aug. 2018). URL: https://arxiv.org/abs/1807.01095 (zitiert auf S. 15, 16, 25).

[23] K. Moriarty, B. Kaliski, J. Jonsson, A. Rusch. *PKCS #1: RSA Cryptography Specifications Version 2.2*. RFC 8017. Nov. 2016. DOI: 10.17487/rfc8017 (zitiert auf S. 32, 33, 40, 41).

[24] K. Moriarty, B. Kaliski, A. Rusch. *PKCS #5: Password-Based Cryptography Specification Version 2.1*. RFC 8018. Jan. 2017. DOI: 10.17487/rfc8018 (zitiert auf S. 24, 32).

[25] M. Nosrati. „Python: An appropriate language for real world programming“. In: *World Applied Programming* 1.2 (2011), S. 110–117. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.678.2551&rep=rep1&type=pdf (zitiert auf S. 16).

[26] *Programming Languages for Cryptography*. [Online; abgerufen 2018-09-19]. Jan. 2017. URL: https://cryptsec.wordpress.com/2016/01/15/programming-languages-for-cryptography/ (zitiert auf S. 17).

[27] *PyCryptodome*. [Online; abgerufen 2018-09-19]. URL: https://www.pycryptodome.org/en/latest/src/introduction.html (zitiert auf S. 25).

[28] *PyNaCl: Python binding to the libsodium library*. [Online; abgerufen 2018-09-19]. URL: https://pynacl.readthedocs.io (zitiert auf S. 25).

[29] *pyOpenSSL – A Python wrapper around the OpenSSL library*. [Online; abgerufen 2018-09-19]. URL: https://github.com/pyca/pyopenssl (zitiert auf S. 26).

[30] *Python Cryptography Toolkit*. [Online; abgerufen 2018-09-19]. URL: https://github.com/dlitz/pycrypto (zitiert auf S. 25).

[31] *Salted Password Hashing - Doing it Right*. [Online; abgerufen 2018-09-19]. URL: https://crackstation.net/hashing-security.htm (zitiert auf S. 34).

[32] *Secure Hash Standard (SHS)*. Techn. Ber. Aug. 2015. DOI: 10.6028/NIST.FIPS.180-4 (zitiert auf S. 33).

[33] *The Incredible Growth of Python*. [Online; abgerufen 2018-09-19]. URL: https://stackoverflow.blog/2017/09/06/incredible-growth-python/ (zitiert auf S. 16).

[34] *The Open Source Definition*. [Online; abgerufen 2018-09-19]. März 2017. URL: https://opensource.org/docs/osd (zitiert auf S. 26).

[35] *The State of the Octoverse 2017*. [Online; abgerufen 2018-09-19]. URL: https://octoverse.github.com/ (zitiert auf S. 16).

[36] *TIOBE Index*. [Online; abgerufen 2018-09-19]. URL: https://www.tiobe.com/tiobe-index/ (zitiert auf S. 16).

[37] M. S. Turan, E. Barker, W. Burr, L. Chen. *Recommendation for Password-Based Key Derivation: Part 1: Storage Applications*. Techn. Ber. Dez. 2010. DOI: 10.6028/NIST.SP.800-132 (zitiert auf S. 32, 34).

[38] *Unlicense Yourself: Set Your Code Free*. [Online; abgerufen 2018-09-19]. URL: https://unlicense.org/ (zitiert auf S. 26).

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum, Unterschrift