



**Hochschule
Augsburg** University of
Applied Sciences

Fakultät für
Informatik

Masterarbeit

Studienrichtung Informatik

Christoph Piechula

Sicherheitskonzepte und Evaluation dezentraler
Dateisynchronisationssysteme am Beispiel »brig«

Prüfer: Prof. Dr.-Ing. Thorsten SCHÖLER
Zweitprüfer: Prof. Dr. Hubert HÖGL

Verfasser:
Christoph Piechula
Holzbachstr. 35
86152 Augsburg
+49 (0) 15 156 619 131
christoph@nullcat.de
Matrikelnr.: 944082

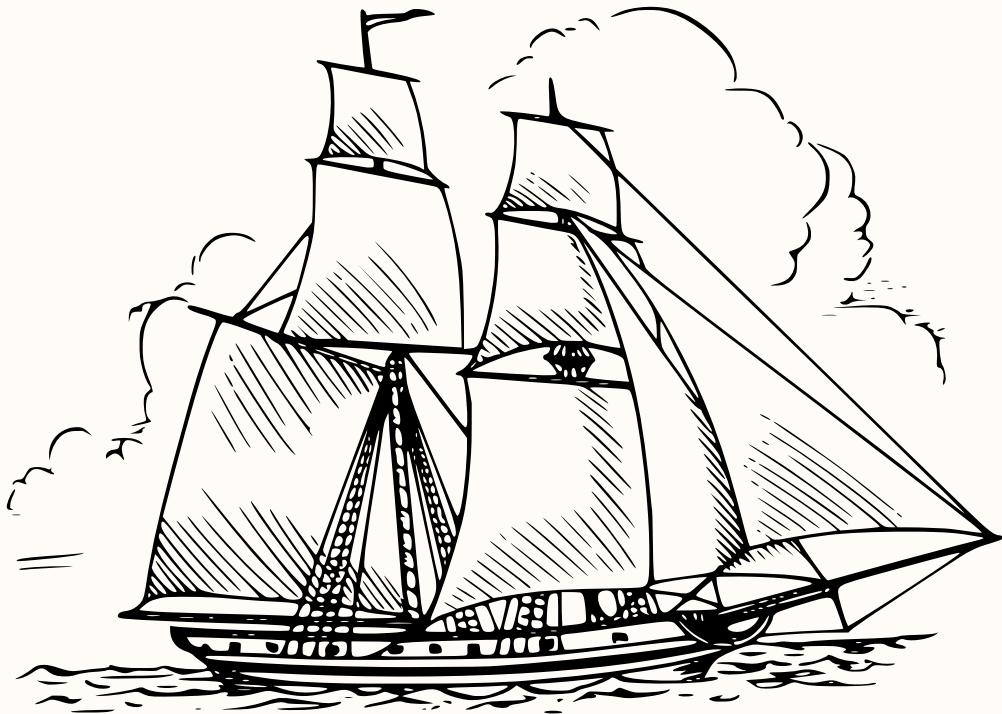
Abgabe der Arbeit am 15. Februar 2017

Hochschule für angewandte
Wissenschaften Augsburg
An der Hochschule 1
86161 Augsburg
Telefon: +49 (0)821-5586-0
Fax: +49 (0)821-5586-3222
info@hs-augsburg.de

HOCHSCHULE AUGSBURG

University of Applied Sciences

MASTERARBEIT AN DER FAKULTÄT INFORMATIK



Sicherheitskonzepte und Evaluation dezentraler Dateisynchronisationssysteme am Beispiel »brig«

Verfasser:

Christoph PIECHULA

Prüfer:

Prof. Dr.-Ing. Thorsten SCHÖLER

Holzbachstraße 35 / 86152 Augsburg

Telefon: +49151/56619131

E-Mail: christoph@nullcat.de

Zweitprüfer:

Prof. Dr. Hubert HÖGL

Abstract

In parallel to this work, the decentralized and secure file sharing solution »brig« has been developed. The purpose of this software is to provide an alternative to the prevailing cloud storage services mainly used today. The objective of »brig« is to combine well established security standards and usability.

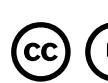
The subject of this work is the evaluation of security of decentralized and centralized data synchronisation solutions. It is also part of the objective to specify advantages and disadvantages of the commonly used technologies and define possible concepts that are essential for the development of a secure and decentralized file sharing solution. Furthermore it is also part of the objective to show possibilities for a secure and trustworthy software development process.

Zusammenfassung

Begleitend zur dieser Arbeit wird die dezentrale und sichere Dateiverteilungslösung »brig« entwickelt. Ziel ist es, ein dezentrales und sicheres Dateisynchronisationssystem als Alternative zu den heute vorherrschenden Cloud-Speicher-Diensten zu entwickeln. Dabei soll die Software aktuelle Sicherheitsstandards und Usability möglichst gut vereinen.

Gegenstand dieser Arbeit ist die Betrachtung der Sicherheit von dezentralen und zentralen Dateisynchronisationssystemen. Dabei sollen Vor- und Nachteile der verwendeten Technologien definiert werden und zugleich mögliche Konzepte ausgearbeitet werden, welche für die Entwicklung einer dezentralen und sicheren Dateiverteilungssoftware essentiell sind. Weiterhin sollen Möglichkeiten für einen sicheren und vertrauenswürdigen Softwareentwicklungsprozess aufgezeigt werden.

© Copyleft Christoph Piechula 2017.
Some rights reserved.



Diese Arbeit ist unter den Bedingungen der
Creative Commons Attribution-3.0 lizenziert.
<http://creativecommons.org/licenses/by/3.0/de>

Danksagung: Mein Dank gilt folgenden Personen: Prof. Dr.-Ing. Thorsten Schöler (Projektbetreuung), Megascops asio, Alces alces (Bovistenverteiler), Isbert und allen Regisseuren, die mich mit Filmkunst am Leben gehalten haben.

Inhaltsverzeichnis

Tabellenverzeichnis	viii
Abbildungsverzeichnis	ix
1. Einleitung	1
1.1. Motivation	1
1.2. Organisation und Schwerpunkte der Arbeit	2
1.3. Zielgruppen und Einsatzszenarien	3
1.4. Projektname und Lizenzierung	3
2. Stand von Wissenschaft und Technik	4
2.1. Sicherheit und Usability von Dateiverteilungssystemen	4
2.1.1. Allgemeines	4
2.1.2. Der »Sicherheitsbegriff«	4
2.1.3. Angriffe und Bedrohungen	4
2.1.4. Datenaustausch über zentrale Lösungen	5
2.1.5. Private Cloud	11
2.1.6. Datenaustausch über dezentrale Lösungen	11
2.1.7. Ähnliche Arbeiten	16
2.2. Markt und Wettbewerber	17
2.3. Closed-Source vs. Open-Source	17
2.4. Gesellschaftliche und politische Aspekte	18
3. Anforderungen	20
3.1. Einleitung	20
3.2. Usability	20
3.3. Sicherheit	21
4. Kryptographische Primitiven und Protokolle	22
4.1. Einleitung	22
4.2. Verschlüsselung	22
4.2.1. Symmetrische Verschlüsselungsverfahren	22
4.2.2. Asymmetrische Verschlüsselungsverfahren	26
4.2.3. Hybride Verschlüsselungsverfahren	27
4.3. Diffie-Hellman-Schlüsselaustausch	28
4.4. Hashfunktionen	28
4.4.1. Kryptographische Hashfunktionen	28
4.4.2. Message Authentication Codes	29
4.5. Authentifizierungsverfahren	29

4.6. Keymanagement	30
5. Sicherheit und Angriffsszenarien	31
5.1. Beurteilung von Sicherheit	31
5.2. Angriffsfläche bei »brig«	32
5.2.1. Allgemein	32
5.2.2. Praxisorientierte Herausforderungen an die Software	32
6. Evaluation IPFS	34
6.1. Einleitung IPFS	34
6.2. IPFS-Basis	35
6.3. IPFS-Backend	35
6.3.1. Speicherung und Datenintegrität	35
6.3.2. Datendeduplizierung	37
6.3.3. IPFS-Daten und IPFS-Blöcke	38
6.4. IPFS-ID	38
6.4.1. Aufbau	38
6.4.2. Authentifizierung	39
6.5. IPFS-Netzwerk	40
6.6. Zusammenfassung IPFS-Evaluation	41
7. Evaluation von »brig«	42
7.1. Einleitung »brig«	42
7.2. Datenverschlüsselung	42
7.2.1. Datenverschlüsselungsschicht	43
7.2.2. Verwendete Algorithmen	44
7.2.3. Geschwindigkeitsevaluation	44
7.2.4. Testumgebung	45
7.2.5. Benchmarks	46
7.3. Metadatenverschlüsselung	55
7.4. »brig«-Identifier	56
7.5. Authentifizierung	57
7.6. Repository-Zugriff	58
7.7. Aufbau einer verschlüsselten Verbindung	60
7.8. Entwicklung und Entwicklungsumgebung	61
7.8.1. Sichere Softwareentwicklung	61
7.8.2. »brig«-Quellcode-Repository	61
7.8.3. Update-Mechanismus	61
8. Verbesserungen und Erweiterungen	62
8.1. Datenverschlüsselung	62
8.2. Keymanagement	63
8.2.1. Sicherung und Bindung der kryptographischen Schlüssel an eine Identität . .	63
8.2.2. GnuPG als Basis für »externe Identität«	65

8.3.	Authentifizierungskonzept	70
8.3.1.	Authentifizierungskonzept mit IPFS–Bordmitteln	70
8.3.2.	Authentifizierungskonzept auf Basis des Web of Trust	74
8.4.	Smartcards und RSA–Token als 2F–Authentifizierung	75
8.4.1.	Allgemein	75
8.4.2.	OpenPGP Smartcard	75
8.4.3.	Zwei–Faktor–Authentifizierung	76
8.4.4.	YubiKey NEO Einleitung	77
8.4.5.	Yubico OTP Zwei–Faktor–Authentifizierung	78
8.4.6.	Konzept zur Zwei–Faktor–Authentifizierung von »brig« mit der YubiCloud . .	80
8.4.7.	Konzept mit eigener Serverinfrastruktur	82
8.4.8.	Einsatz des YubiKey zur Passworthärtung	85
8.4.9.	YubiKey als Smartcard	86
8.5.	Sichere Entwicklung und Entwicklungsumgebung	90
8.5.1.	Bereitstellung der Software	90
8.5.2.	Update–Management	92
8.5.3.	Signieren von Quellcode	92
8.5.4.	Sichere Authentifizierung für Entwickler	94
9.	Fazit	99
9.1.	Zusammenfassung	99
9.1.1.	Allgemein	99
9.1.2.	Schlüsselverwaltung	99
9.1.3.	Verschlüsselung	100
9.1.4.	Authentifizierung	100
9.1.5.	Softwareentwicklung und Softwareverteilung	100
9.2.	Selbstkritik und aktuelle Probleme	100
9.3.	Ausblick	101
A.	Umfang IPFS–Codebasis	103
B.	IPFS–Grundlagen	104
C.	IRC–Log zur TLS–Verschlüsselung	106
D.	Details zur CPU–Architektur	107
E.	Benchmark–Skripte	110
F.	Schlüsselgenerierung auf der Smartcard	127
G.	Unterschlüssel erstellen	130
H.	Ablaufdatum ändern	134
I.	Exportieren der GnuPG–Schlüssel	136

J. Schlüssel auf die Smartcard verschieben	137
K. User- und Admin-PIN ändern	141
L. QR-Code-Snippet	143
M. YubiCloud Zwei-Faktor-Authentifizierung	144
Literaturverzeichnis	146
Eidesstattliche Erklärung	149

Tabellenverzeichnis

4.1. Laut ISO 10116 Standard definierte Betriebsarten für blockorientierte Verschlüsselungsalgorithmen.	25
4.2. Auf ECRYPTII-Einschätzung basierende effektive Schlüsselgrößen asymmetrischer und symmetrischer Verfahren im direkten Vergleich.	27
7.1. Evaluerte Testsysteme mit und ohne AES-NI-Befehlserweiterungssatz.	45
7.2. Zeigt die effizientesten Blockgrößen beim Entschlüsseln. Der erste Wert entspricht der Zeit in Millisekunden, der zweite Wert der Geschwindigkeit in MiByte/s beim Lesen einer 128 MiByte großen Datei.	47
7.3. Zeigt die effizientesten Blockgrößen beim Verschlüsseln. Der erste Wert entspricht der Zeit in Millisekunden, der zweite Wert der Geschwindigkeit in MiByte/s beim Schreiben einer 128 MiByte großen Datei.	48
7.4. Zeigt die effizientesten Blockgrößen beim Entschlüsseln. Der erste Wert entspricht der Zeit in Millisekunden, der zweite Wert der Geschwindigkeit in MiByte/s beim Lesen einer 32 MiByte großen Datei.	51
7.5. Zeigt die effizientesten Blockgrößen beim Verschlüsseln. Der erste Wert entspricht der Zeit in Millisekunden, der zweite Wert der Geschwindigkeit in MiByte/s beim Schreiben einer 32 MiByte großen Datei.	52
7.6. Geschätzte Passwort-Entropie und Crackdauer von <i>unsicheren</i> Passwörtern.	58

Abbildungsverzeichnis

2.1.	Scherzhafte Darstellung eines möglichen Angriffs auf eine Festplattenverschlüsselung mit optimalem Kosten–Nutzen–Verhältnis.	5
2.2.	Datensynchronisation über zentrale Cloud–Speicher–Dienste, wie beispielsweise Dropbox. Alice und Bob teilen gemeinsam Dateien über das shared storage aus. Mallory stellt einen potentiellen Angreifer dar. Mallory kann ein externer Angreifer oder auch ein interner Mitarbeiter sein, der Zugriff auf die Daten hat.	6
2.3.	Quick Double Switch Attack Flow. Die Buchstaben a, b, c und d repräsentieren dabei jeweils die Synchronisationsvorgänge.	9
2.4.	Dezentraler Datenaustausch über Peer-to-Peer–Kommunikation. Es existiert keine zentrale Instanz, jeder Peer im Netzwerk ist gleichberechtigt.	12
2.5.	Zeigt einen Swarm. Alice lädt die Datei image.iso von mehreren Teilnehmern gleichzeitig, die Datei mydog.png jedoch nur von Dan.	13
4.1.	Konzept beim Austausch von Daten über einen unsicheren Kommunikationsweg unter Verwendung symmetrischer Kryptographie. Alice und Bob teilen einen gemeinsamen Schlüssel, um die Daten zu ver- und entschlüsseln.	23
4.2.	Unterschied in der Arbeitsweise zwischen Block- und Stromchiffre. Die Blockchiffre verschlüsselt die Daten blockweise, eine Stromchiffre hingegen verschlüsselt den Datenstrom »on the fly«.	24
4.3.	Bild zur graphischen Verdeutlichung des ECB–Modus im Vergleich zu einem block chaining cipher.	24
4.4.	ECB–Modus (links): Datenblöcke werden unabhängig voneinander verschlüsselt. CFB–Modus (rechts): Datenblöcke hängen beim Verschlüsseln voneinander ab.	25
4.5.	Prinzip asymmetrischer Verschlüsselung. Verschlüsselt wird mit dem öffentlichen Schlüssel des Empfängers. Der Empfänger entschlüsselt mit seinem privaten Schlüssel die Nachricht. Die Signatur erfolgt mit dem privaten Schlüssel des Senders, validiert wird diese mit dem öffentlichen Schlüssel des Senders.	26
4.6.	Grafische Darstellung, Ablauf des Diffie–Hellman–Schlüsseltausch.	28
4.7.	Message–Übertragung mit HMAC.	29
6.1.	Das Multihash–Format.	36
6.2.	IPFS Block–Level Deduplizierung von Daten. Eine 4 MiB große Textdatei wurde viermal kopiert und jeweils an verschiedenen Stellen geändert.	38
7.1.	»brig« als Overlay–Netzwerk für IPFS	42
7.2.	»brig« Verschlüsselungsschicht mit Datenverschlüsselung mit Authentizität.	43
7.3.	»brig«–Container–Format für Datenverschlüsselung mit Authentizität.	44
7.4.	Testablauf in der RAM–Disk bei der Erhebung der Messdaten.	46

7.5. Lesegeschwindigkeit des Kryptographielayers bei der Benutzung verschiedener Blockgrößen.	47
7.6. Schreibgeschwindigkeit des Kryptographielayers bei der Benutzung verschiedener Blockgrößen.	48
7.7. Geschwindigkeitszuwachs durch AES-NI.	49
7.8. Lesegeschwindigkeit der Verschlüsselungsschicht auf schwächeren Systemen bei der Benutzung verschiedener Blockgrößen.	51
7.9. Schreibgeschwindigkeit der Verschlüsselungsschicht auf schwächeren Systemen bei der Benutzung verschiedener Blockgrößen.	52
7.10. Geschwindigkeitseinbruch unter Verwendung von Verschlüsselung auf schwächeren Systemen.	53
7.11. Für die gleiche Datei werden aktuell unterschiedliche Schlüssel generiert. Das hat zur Folge, dass die Deduplizierungsfunktionalität von IPFS weitestgehend nicht mehr funktioniert.	54
7.12. Geschwindigkeitseinbruch verursacht durch Schlüsselableitung mittels der scrypt-Schlüsselableitungsfunktion.	55
7.13. User-Lookup mittels »brig«-ID (hier bestehend aus gekürzter Peer-ID + User-Hash). Nur bei Übereinstimmung vom Peer-Fingerprint und Benutzernamen-Fingerprint wird der Benutzer als valide erkannt.	57
7.14. Vereinfachte Ansicht bei der Aushandlung eines Sitzungsschlüssel beim Verbindungsauftbau.	60
 8.1. Bei Convergent Encryption resultiert aus gleichem Klartext der gleiche Geheimtext da der Schlüssel zur Verschlüsselung vom Klartext selbst abgeleitet wird.	62
8.2. Externes asymmetrisches Schlüsselpaar dient als Hauptschlüssel zur Sicherung der ungesicherten IPFS-Daten. Das Signieren der öffentlichen IPFS-ID ermöglicht eine zusätzliche Schicht der Authentifizierung.	64
8.3. »brig« stellt mittels VFS eine Zugriffsschnittstelle für IPFS dar.	64
8.4. GnuPG-Schlüsselpaar RSA/RSA bestehend aus einem Haupt- und Unterschlüsselpaar. Beide Schlüssel haben unterschiedliche Fähigkeiten und bestehen jeweils aus einem öffentlichen und einem privaten Schlüssel. In Hexadezimal ist jeweils der Fingerprint eines Schlüssels dargestellt. Der Fingerprint ist 20 Bytes groß. Die »Long-Key-ID« entspricht den letzten 8 Bytes, die »Short-Key-ID« entspricht den letzten 4 Bytes. . .	66
8.5. GPG-Schlüsselbund mit Unterschlüsseln für den regulären Einsatz. Jeder Unterschlüssel ist an einen bestimmten Einsatzzweck gebunden.	69
8.6. Fragwürdige Entropieschätzung im GnuPG-Pinentry Dialog.	69
8.7. »brig« QR-Code um einen Synchronisationspartner auf einfache Art und Weise zu authentifizieren.	70
8.8. Frage-Antwort-Authentifizierung. Alice stellt Bob eine persönliche Frage auf die Bob die Antwort weiß.	71
8.9. Authentifizierung über ein gemeinsames Geheimnis unter Verwendung des Socialist Millionaire-Protokoll.	73
8.10. Authentifizierung auf Basis des Web of Trust.	74

8.11. Von g10 code vertriebene Smartcard für den Einsatz mit GnuPG.	76
8.12. YubiKey NEO mit USB-Kontaktschnittstelle und Push-Button, welcher bei Berührung reagiert.	77
8.13. GUI des YubiKey Personalization Tool. Das Konfigurationswerkzeug ist eine QT-Anwendung, diese wird von den gängigen Betriebssystemen (Linux, MacOs, Windows) unterstützt.	78
8.14. Yubico OTP Aufbau	79
8.15. Yubico OTP-Authentifizierungsprozess an der YubiCloud.	79
8.16. YubiCloud Response bei Zwei-Faktor-Authentifizierung. Seriennummer des YubiKeys wurde retuschiert.	80
8.17. Schematische Darstellung der Zwei-Faktor-Authentifizierung gegenüber einem »brig«-Repository.	81
8.18. Validierungsserver, welcher über einen Reverse-Proxy angesprochen wird.	84
8.19. Updateprozess zur sicheren Softwareverteilung.	92
8.20. Nach dem Absetzen eines signierten Commits/Tags erscheint auf der GitHub-Plattform ein zusätzliches Label »Unverified«, wenn der öffentliche Schlüssel des Entwicklers bei GitHub nicht hinterlegt ist.	93
8.21. Verifiziertes GitHub-Signatur-Label eines Commits/Tags welches aufgeklappt wurde.	94
8.22. GitHub-Anmeldung mit Zwei-Faktor-Authentifizierung.	95
8.23. SSH-Authentifizierungsvorgang mittels Public-Key-Verfahren.	96

1.1. Motivation

Der Austausch von Dokumenten beziehungsweise Dateien wurde früher hauptsächlich über Datenträger (Diskette, USB-Stick) oder E-Mail durchgeführt. Mit dem Aufkommen der Cloud-Lösungen der letzten Jahre werden Dateien immer öfters über Cloud-Dienste, wie beispielsweise Apple iCloud, Dropbox, Microsoft OneDrive oder Google Drive, ausgetauscht.

Diese Dienste basieren auf einer zentralen Architektur und ermöglichen dem Benutzer, seine Dateien über mehrere Computer hinweg zu synchronisieren und mit Freunden – oder im geschäftlichen Umfeld mit Partnern – auszutauschen. Hierbei ist man in der Regel auf die Verfügbarkeit des jeweiligen Dienstes angewiesen. Fällt der Dienst aus oder wird beispielsweise von Strafverfolgungsbehörden geschlossen (Megaupload-Schließung¹), bleibt einem der Zugriff auf die eigenen Dateien verwehrt.

Auch wenn das Aufkommen dieser Dienste auf den ersten Blick eine Abhilfe sein mag, so werden beim genaueren Hinsehen Risiken und Nachteile, welche erst durch Aufkommen dieser Dienste entstanden sind, deutlich sichtbar. Eines der Hauptprobleme ist, aufgrund mangelnder Transparenz, der Schutz der Daten beziehungsweise der Privatsphäre.

Hier muss dem Dienstanbieter vollständig vertraut werden, da nicht bekannt ist welche Drittparteien Zugriff auf die Daten haben. Dies mag bei der persönlichen Musik-Sammlung – im Gegensatz zur Speicherung sensibler medizinischer oder finanzieller Unterlagen – weniger ein Problem sein. Spätestens seit den Snowden-Enthüllungen und des Bekanntwerdens vom PRISM-Überwachungsprogramm ist offiziell bekannt[1], dass der Zugriff auf persönliche Daten durch Drittparteien erfolgt ist, beziehungsweise erzwungen wurde. Neben dem geduldeten oder rechtlich erzwungenen Zugriff durch Drittparteien, haben Cloud-Speicher-Anbieter wie beispielsweise Dropbox in der Vergangenheit immer wieder für Schlagzeilen gesorgt. Durch diverse Softwarefehler war beispielsweise der Zugriff über mehrere Stunden mit beliebigen Passwörtern möglich². Ein weiterer Softwarefehler hat bei der Aktivierung bestimmter Features Daten unwiderruflich gelöscht³. Daneben wird die Sicherheit von Cloud-Speicher-Diensten in Studien bemängelt, vgl. [2].

Will man Daten privat oder geschäftlich austauschen, so muss man sich in der Regel auf einen Anbieter einigen. Hierbei stellt die Fragmentierung⁴ der Cloud-Speicher-Anbieter den Benutzer oft vor weitere Herausforderungen.

Will man aus persönlichen Bedenken auf den Einsatz von Cloud-Speicher-Diensten verzichten, bleibt einem immer noch die Möglichkeit, Dateien über E-Mail zu versenden. Hier erschließt sich aber bei

¹Hinweise zur Schließung von Megaupload: <https://de.wikipedia.org/w/index.php?title=Megaupload&oldid=161927073>

²Authentifizierungs-Bug: <https://blogs.dropbox.com/dropbox/2011/06/yesterdays-authentication-bug/>

³Selective-Sync-Bug: <https://plus.google.com/+MichaelArmogan/posts/E9sVnrLTB5C>

⁴Übersicht Online-Backup-Provider:

https://en.wikipedia.org/w/index.php?title=Comparison_of_online_backup_services&oldid=760247797

näherer Betrachtung ein ähnliches Problem wie bei den Cloud-Speicher-Anbietern. Die Privatsphäre, beziehungsweise der Schutz der Daten vor dem Zugriff durch Dritte, ist mangelhaft. Es gibt die Möglichkeit E-Mails beispielsweise mittels Pretty Good Privacy (PGP) zu verschlüsseln, jedoch ist der Einsatz und Aufwand für den Otto Normalverbraucher schlichtweg zu kompliziert und wird daher kaum genutzt.

Der Austausch über einen Cloud-Speicher-Dienst oder E-Mail ist nichtsdestotrotz der Quasi-Standard. Es gibt zwar technisch gesehen weitere Möglichkeiten, Daten auch ohne eine zentrale Instanz auszutauschen, diese sind jedoch entweder recht unbekannt, für den Otto Normalverbraucher unbenutzbar oder unsicher. Zu den bekanntesten Vertretern gehören hier wahrscheinlich Syncthing⁵, git-annex⁶ oder auch Resilio⁷.

Diese Ausgangssituation hat letztendlich nicht nur aus persönlichem Interesse und mangels Alternativen dazu geführt, sich mit der Thematik näher zu befassen und weitere Möglichkeiten zu erschließen. In Zusammenarbeit mit meinem Kommilitonen, Christopher Pahl, wurde die Entwicklung an dem dezentralen Dateisynchronisationswerkzeug »brig« gestartet, welches die aktuelle Situation verbessern soll.

1.2. Organisation und Schwerpunkte der Arbeit

Projektschwerpunkte:

Ziel ist es eine Software zu entwickeln, welche aktuelle Sicherheitsstandards und Usability möglichst gut vereint und dabei ohne zentrale Instanz auskommt.

Aufgegliedert liegen die Schwerpunkte wie folgt:

Vollständige Transparenz: Der Benutzer soll vollständige Transparenz bezüglich der Software und dem Entwicklungsprozess haben. Nur so lässt ich eine vertrauenswürdige Implementierung gewährleisten.

Aktuelle Sicherheitsstandards: Die Software soll die Daten des Benutzers zu jeder Zeit möglichst gut schützen. Hierzu sollen bewährte Sicherheitsstandards verwendet werden.

Möglichst intuitive Benutzung: Der Benutzer soll von der Sicherheitskomplexität so wenig wie möglich mitbekommen. Die Software soll dabei jedoch mindestens so einfach nutzbar sein wie die heutzutage gängigen Cloud-Dienste.

Organisation:

Die Basis für die Entwicklung der Software, Validierung einzelner Komponenten und Prozesse, sowie die Ausarbeitung möglicher zukünftiger Konzepte stellen die folgenden Arbeiten dar:

- 1) »brig«: Ein Werkzeug zur sicheren und verteilten Dateisynchronisation, *Christopher Pahl*
- 2) Sicherheitskonzepte und Evaluation dezentraler Dateisynchronisationssysteme am Beispiel »brig«, *Christoph Piechula*

⁵Syncthing: <https://syncthing.net/>

⁶git-annex: <https://git-annex.branchable.com/>

⁷Resilio: <https://getsync.com/>

Aktuell wird die Software im Rahmen der genannten Masterarbeiten bei Prof. Dr.-Ing. Thorsten Schöler in der *Distributed Systems Group*⁸ der Hochschule Augsburg entwickelt.

Die vorliegende Arbeit hat dabei aufgrund des Umfangs des Gesamtprojekts folgende Schwerpunkte, Fragestellungen und Ziele:

- ▶ Sensibilisierung von Entwicklern und Benutzern für die Thematik der sicheren Software.
Warum ist sichere Software und Kryptographie für unsere Gesellschaft wichtig? Worauf ist bei der Implementierung zu achten, wie sind die Zusammenhänge?
- ▶ Zusammenhängende Betrachtung der Sicherheit von Software.
- ▶ Betrachtung der Sicherheitskonzepte, Vor- und Nachteile zentraler und dezentraler Lösungen.
- ▶ Evaluation bisheriger Ansätze (Sicherheit, Usability) und Definition möglicher Verbesserungen, welche in die Weiterentwicklung und in den Entwicklungsprozess einfließen sollen.
- ▶ Betrachtung von Sicherheitskonzepten, welche für eine sichere, transparente und vertrauenswürdige Softwareentwicklung und Softwareverteilung essentiell sind.

1.3. Zielgruppen und Einsatzszenarien

Da die Software nicht auf ein bestimmtes Fachgebiet begrenzt werden kann, ist die Nutzung sowohl durch Individuen, Unternehmen als auch öffentliche Einrichtungen möglich.

Aufgrund der Ausrichtung der Projektziele, sollen jedoch vor allem Benutzer mit erhöhten Sicherheitsanforderungen von der Software profitieren. Hierzu gehören neben einzelnen Individuen vor allem bestimmte Personengruppen wie beispielsweise:

- ▶ Journalisten und Aktivisten
- ▶ Fachkräfte im medizinischen Bereich
- ▶ Fachkräfte in öffentlichen Einrichtungen

1.4. Projektname und Lizenzierung

Der Name »brig« ist eine Anlehnung an das zweimastige Handelsschiff *brigg*⁹, welches gegen Ende des 18. Jahrhunderts zum Einsatz kam. Die Namensanlehnung soll analog den dezentralen Transport von Daten darstellen.

Aufgrund der Projektziele kommt als Lizenzierung die Open-Source-Lizenz¹⁰ AGPLv3¹¹ zum Einsatz. Denkbar wären jedoch im späteren Verlauf des Projektes kombinierte Lizenzen für Unternehmen.

⁸Distributed Systems Group: <http://dsg.hs-augsburg.de>

⁹Brigg Handelsschiff: <https://de.wikipedia.org/w/index.php?title=Brigg&oldid=155265558>

¹⁰Open-Source-Software: https://de.wikipedia.org/w/index.php?title=Open_Source&oldid=162165962

¹¹AGPLv3 Lizenz: <https://www.gnu.org/licenses/agpl-3.0.de.html>

2.1. Sicherheit und Usability von Dateiverteilungssystemen

2.1.1. Allgemeines

Zentrale und dezentrale Systeme sind die Basis für den Austausch von Informationen. Ob ein System zentral oder dezentral fungiert ist nicht immer klar abgrenzbar. Oft kommen auch hybride Systeme zum Einsatz, welche zwar dezentral funktionieren, jedoch teilweise zentrale Instanzen benötigen. Hier wäre beispielsweise das Torrent-Konzept zu nennen. Weitere Informationen hierzu, vgl. [3], S. 232 ff.

2.1.2. Der »Sicherheitsbegriff«

Betrachtet man die Sicherheit von Dateiverteilungssystemen, so müssen verschiedene Teilespekte betrachten werden. Leider ist das Umfeld der Sicherheit sehr groß und die Begrifflichkeiten nicht immer eindeutig definiert, teilweise werden auch bestimmte Begrifflichkeiten synonym verwendet¹. In der Fachliteratur (vgl. [4], S. 21 f.) spricht man bei Sicherheit oft von den folgenden fünf Sicherheitsaspekten:

- ▶ *Vertraulichkeit*: Schutz der Daten vor Zugriff durch Dritte.
- ▶ *Integrität*: Schutz der Daten vor Manipulation.
- ▶ *Authentifizierung*: Eindeutige Identifikation von Benutzern.
- ▶ *Autorisierung*: Definiert die Zugangs- und Zugriffssteuerung auf Dienste.
- ▶ *Verfügbarkeit*: Dienste stehen legitimen Benutzern tatsächlich zur Verfügung.

Dies sind auch die Sicherheitsaspekte, die bei der Verwendung von Cloud-Speicher-Diensten entscheidend sind. Zur Umsetzung der genannten Sicherheitsaspekte reicht in der Regel die alleinige Implementierung technischer Komponenten nicht aus. Es muss viel mehr ein Prozess entwickelt werden, der Sicherheit permanent sicherstellt und evaluiert. Hierzu gehört auch im großen Maße eine gewisse Sensibilität, für welche der Anwender geschult werden muss.

2.1.3. Angriffe und Bedrohungen

Die Sicherheit eines Systems lässt sich nicht mit einem einfachen »Ja« oder »Nein« beantworten. Betrachtet man ein System bezüglich seiner Sicherheit, so muss auch genau definiert werden, gegen welches Angriffsszenario ein System sicher ist. Auch ein System das aus kryptographischer Sicht als »sicher« zu betrachten wäre, kann im einfachsten Fall, durch die Weitergabe von Zugangsdaten an Dritte, kompromittiert werden.

¹BSI Glossar: https://www.bsi.bund.de/DE/Themen/ITGrundschutz/ITGrundschutzKataloge/Inhalt/Glossar/glossar_node.html

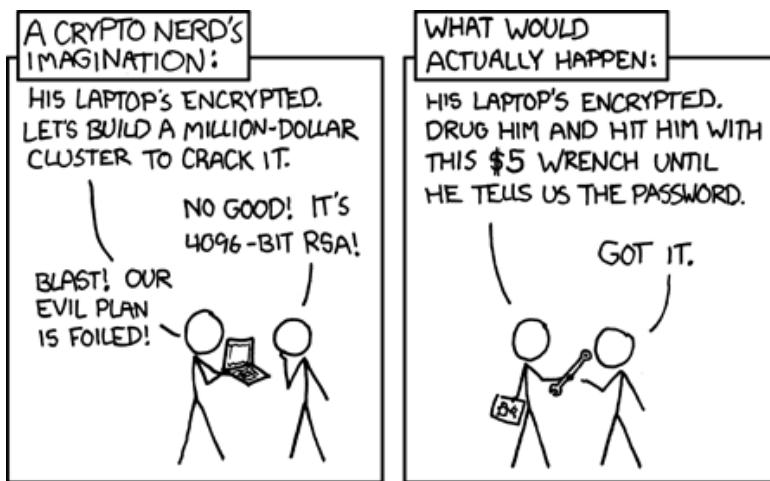


Abbildung 2.1.: Scherhaftige Darstellung eines möglichen Angriffs auf eine Festplattenverschlüsselung mit optimalem Kosten–Nutzen–Verhältnis.²

Neben dem technischen Ansatz beim Angriff auf ein System, gibt es auch die psychologische Komponente beim Menschen, welche wahrscheinlich die größte Schwachstelle in den meisten Systemen darstellt.

Beim technischen Ansatz werden in der Regel Fehler in der Software oder Infrastruktur ausgenutzt, um sich unbefugten Zugriff auf Informationen zu verschaffen.

Beim nicht technischen Angriff wird der Benutzer auf »psychologischer Ebene« manipuliert und mit sogenannten »Social Hacking«-, auch »Social Engineering«-Methoden dazu verleitet beispielsweise sein Passwort weiterzugeben. Auch der Einsatz von »Phishing« ist eine Variante von »Social Engineering«. Abb. 2.1 zeigt scherhaft eine weitere Variante für welche Menschen anfällig sind.

Um Sicherheit zu gewährleisten, ist es wichtig ein System im Ganzen zu betrachten. Die Implementierung bestimmter Sicherheitsfeatures ist nur die technische Maßnahme. Der Benutzer eines Systems erwartet in erster Linie Funktionalität und möchte sich in den wenigsten Fällen mit dem System oder der Sicherheit des Systems auseinandersetzen. Benutzer sind oft nicht genug sensibilisiert was den Datenschutz oder auch die Gefahren und Konsequenzen bei einem Sicherheitsproblem angeht.

Weiterhin sollte bedacht werden, dass die Definition eines »sicheren Systems«, in der Regel ein Kompromiss aus den folgenden Punkten ist:

- ▶ Finanzieller Aufwand
- ▶ Sicherheit
- ▶ Usability

2.1.4. Datenaustausch über zentrale Lösungen

2.1.4.1. Funktionsweise zentraler Dienste

Zentrale Dienste klassifizieren sich im Kontext dieser Arbeit durch die Eigenschaft, dass es eine zentrale Instanz gibt, welche zum Austausch der Daten benötigt wird. Dies sind in den meisten Fällen

²Bildquelle: <http://imgs.xkcd.com/comics/security.png>

die Server des Cloud-Speicher-Anbieters, welche für die Synchronisation und Speicherung der Daten verantwortlich sind.

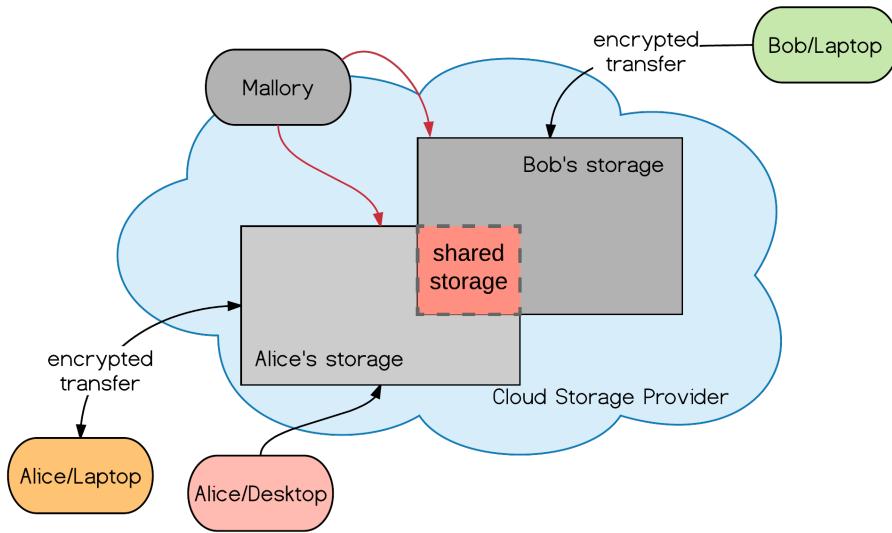


Abbildung 2.2.: Datensynchronisation über zentrale Cloud-Speicher-Dienste, wie beispielsweise Dropbox. Alice und Bob teilen gemeinsam Dateien über das shared storage aus. Mallory stellt einen potentiellen Angreifer dar. Mallory kann ein externer Angreifer oder auch ein interner Mitarbeiter sein, der Zugriff auf die Daten hat.

Abb. 2.2 zeigt schematisch das Konzept beim Austausch von Daten über einen Cloud-Speicher-Dienst. Die Daten des Benutzers werden hierbei über eine zentrale Stelle synchronisiert. In der Regel legt der Benutzer einen Account beim Cloud-Speicher-Anbieter an, installiert dessen Client-Software, legt einen Ordner zur Synchronisation fest und registriert sich abschließend online beim Cloud-Speicher-Anbieter.

Dieser Ordner lässt sich dann beispielsweise auf weitere Geräte des Benutzers synchronisieren. Weiterhin gibt es in der Regel die Möglichkeit, Dateien mit anderen Benutzern zu teilen (siehe Abb. 2.2). Welche genauen Einstellungen sich vornehmen lassen und wie feingranular die Benutzerverwaltung und die Möglichkeiten beim Synchronisieren sind, ist von dem jeweiligen Cloud-Speicher-Anbieter abhängig.

Mittlerweile werben die Anbieter mit starker Verschlüsselung und dass die Daten in der Cloud »sicher« sind. Spätestens seit den Snowden-Enthüllungen ist es jedoch klar, dass die Anbieter dazu gezwungen werden können, die Daten eines Benutzers herauszugeben (vgl. [1]).

2.1.4.2. Synchronisationssoftware

Die verwendete Software zur Synchronisation ist ebenfalls vom jeweiligen Anbieter abhängig. Das Problem hierbei ist, dass die Software in der Regel proprietär ist und der Benutzer weder die genaue Funktionalität kennt, noch das Vorhandensein von Hintertüren ausschließen kann. Die Software wird in den meisten Fällen für verschiedene Plattformen bereitgestellt. Weiterhin ermöglichen die Anbieter auch den Zugriff auf die Daten mittels eines Webbrowser-Interface.

2.1.4.3. Sicherheit von Cloud-Speicher-Anbietern

Es ist sehr schwierig, die Sicherheit der Cloud-Speicher-Anbieter realistisch zu bewerten, da sowohl die Infrastruktur als auch die verwendete Software intransparent und proprietär sind.

Die Daten werden laut Aussagen der Hersteller³⁴ verschlüsselt übertragen und mittlerweile auch verschlüsselt gespeichert.

Beim Einsatz der Cloud-Speicher-Dienste hängt die Sicherheit der Daten somit in erster Linie vom Dienstanbieter ab. Beim iCloud-Dienst von Apple beispielsweise werden die Daten verschlüsselt bei Drittanbietern wie der Amazon-S3- oder Windows-Azure-Cloud gespeichert⁵. Die Metadaten und kryptographischen Schlüssel verwaltet Apple auf seinen eigenen Servern. Dropbox hat laut Medienberichten mittlerweile von der Amazon-Cloud auf eine eigene Infrastruktur migriert⁶.

Das Problem hierbei ist die Umsetzung der Datenverschlüsselung der gängigen Cloud-Speicher-Anbieter. Anbieter wie Dropbox verschlüsseln laut eigener Aussage die Daten in der Cloud nach aktuellen Sicherheitsstandards. Das Problem im Fall von Dropbox ist jedoch, dass Dropbox und nicht der Endbenutzer der Schlüsselinhaber ist. Es ist also, auch wenn es laut internen Dropbox-Richtlinien verboten ist, möglich, dass Mitarbeiter beziehungsweise dritte Parteien die Daten des Nutzers einsehen können (vgl. [5] S. 103).

Ein weiteres Problem ist, dass ein Cloud-Speicher-Anbieter aufgrund seiner zentralen Rolle ein gutes Angriffsziel bildet. Erst kürzlich wurde bekannt, dass Angreifer im Jahr 2012 ungefähr 70 Millionen Zugangsdaten⁷ bei Dropbox entwendet haben. Hat ein Angreifer also die Zugangsdaten erbeutet, bringt die Verschlüsselung, die der Cloud-Dienst betreibt, in diesem Fall nichts. Die gestohlenen Passwörter waren nicht im Klartext einsehbar, moderne Angriffsmöglichkeiten auf Passwörter zeigen jedoch, dass das nichtsdestotrotz ein großes Problem ist (siehe auch Kapitel 5).

Abhilfe könnte in diesem Fall eine zusätzliche Verschlüsselung auf Seiten des Nutzers schaffen. Diese ist jedoch für den Endverbraucher oft zu kompliziert, aufgrund von Fehlern in der Implementierung nicht optimal geeignet (EncFS Audit⁸) oder proprietär (Boxcryptor⁹).

Den meisten Anbietern muss man vertrauen, dass diese mit den Daten und Schlüsseln sorgsam umgehen. Auch wenn sich viele Anbieter wie beispielsweise Dropbox bemühen, aus den Fehlern der Vergangenheit zu lernen und verbesserte Sicherheitsmechanismen wie beispielsweise Zwei-Faktor-Authentifizierung¹⁰ in ihre Software integrieren, bleibt jedoch die Krux der Intransparenz und der proprietären Software. Es ist für den Benutzer nicht ohne Weiteres möglich, die Sicherheit der Client-Software zu validieren.

³Apple iCloud Security: <https://support.apple.com/en-us/HT202303>

⁴Dropbox Security: <https://www.dropbox.com/security>

⁵Apple iOS Security: http://www.apple.com/business/docs/iOS_Security_Guide.pdf

⁶Dropbox Exodus Amazon Cloud Empire:

<http://www.wired.com/2016/03/epic-story-dropbox-exodus-amazon-cloud-empire/>

⁷Dropbox hackers stole 68 million passwords:

<http://www.telegraph.co.uk/technology/2016/08/31/dropbox-hackers-stole-70-million-passwords-and-email-addresses/>

⁸EncFS Audit: <https://defuse.ca/audits/encfs.htm>

⁹Boxcryptor: <https://de.wikipedia.org/w/index.php?title=Boxcryptor&oldid=161953026>

¹⁰Zwei-Faktor-Authentifizierung:

<https://de.wikipedia.org/w/index.php?title=Zwei-Faktor-Authentifizierung&oldid=160891860>

2011 hat der Sicherheitsforscher *Derek Newton* den Authentifizierungsmechanismus von Dropbox kritisiert. Nach einmaligem Registrieren und Einrichten des Dropbox-Client, werden für die Synchronisation keine weiteren Zugangsdaten mehr benötigt. Der Authentifizierungsmechanismus benötigt nur ein sogenanntes Authentifizierungs-Token (dieses wird dem Client nach der Registrierung vom Server zugewiesen), die sogenannte HOST_ID. Mit dieser authentifiziert sich der Dropbox-Client bei zukünftigen Synchronisationsvorgängen gegenüber dem Dropbox-Dienst.

Ein großes Problem war hierbei auch, dass die HOST_ID unverschlüsselt in einer Konfigurationsdatei (sqlite3-Datenbank) abgelegt war. Diese ID bleibt anscheinend auch nach Änderung der Zugangsdaten weiterhin bestehen.

Eine weitere Arbeit aus dem Jahr 2011 beschreibt verschiedene Angriffsszenarien und Probleme, welche die Datensicherheit und Privatsphäre von Cloud-Speicher-Benutzern in Frage stellt (vgl. [6]).

2013 haben weitere Sicherheitsforscher den Dropbox-Client mittels Reverse Engineering analysiert. Ab der Version 1.2.48 wird die HOST_ID in einer verschlüsselten sqlite3-Datenbank abgespeichert. Diese Nachbesserung seitens Dropbox war nicht besonders effektiv, da sich die Schlüssel zum Entschlüsseln weiterhin auf dem Client-PC befinden. Zusätzlich wird für die Authentifizierung in neueren Dropbox-Versionen ein HOST_INT-Wert benötigt, welcher ebenfalls vom Client-PC extrahiert werden kann.

Mittels dieser beiden Werte kann die Zwei-Faktor-Authentifizierung (2FA), wie sie von Dropbox implementiert ist, umgangen werden. Die Client-API verwendet anscheinend keine Zwei-Faktor-Authentifizierung. Darüber hinaus lassen sich auf Basis der beiden Parameter sogenannte Autologin-URLs generieren. Den Forschern ist es auch gelungen, einen Open-Source-Prototypen zu entwickeln. Für weitere Details vgl. [7], beziehungsweise siehe Vortrag USENIX Open Access Content¹¹.

2015 wurde bekannt, dass die vorherrschenden Cloud-Speicher-Anbieter für sogenannte Man-in-the-Cloud-Angriffe anfällig sind. Die bereits im Jahr 2011 entdeckten Schwächen bei der Authentifizierung von Dropbox sind weiterhin präsent und auch auf andere Cloud-Speicher-Anbieter übertragbar. Um die Client-Software gegenüber dem Cloud-Speicher-Dienst zu authentifizieren, werden wie auch bei Dropbox, Authentifizierungs-Tokens verwendet. Für den Angriff haben die Forscher ein sogenanntes Switcher-Programm entwickelt, welches in der Lage ist, ein Authentifizierungs-Token auf dem Computer des potentiellen Opfers auszutauschen. Abb. 2.3 zeigt den Ablauf eines möglichen Man-in-the-Cloud-Angriffs.

¹¹USENIX Vortrag »Looking Inside the (Drop) Box«:

<https://www.usenix.org/conference/woot13/workshop-program/presentation/kholia>

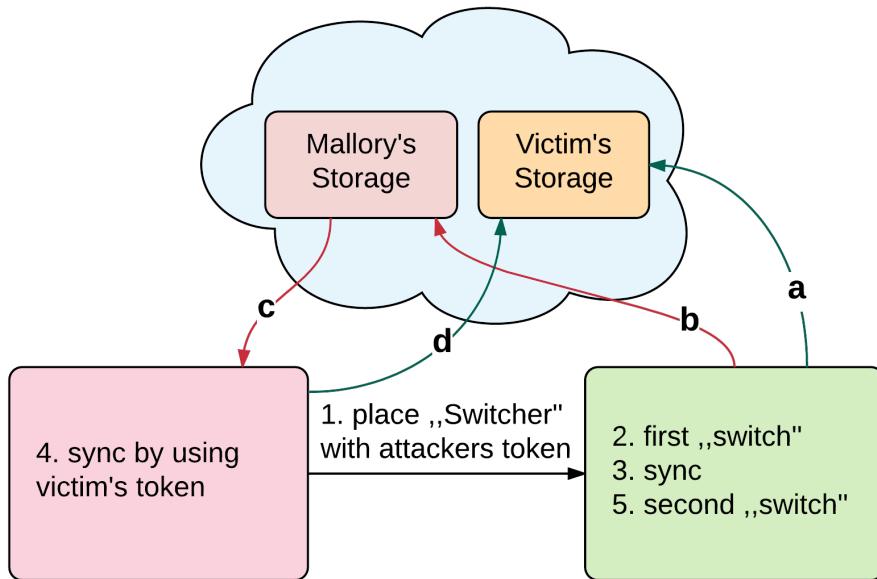


Abbildung 2.3.: Quick Double Switch Attack Flow. Die Buchstaben a, b, c und d repräsentieren dabei jeweils die Synchronisationsvorgänge.

1. Der Angreifer platziert den Switcher auf dem Rechner des Opfers (beispielsweise mittels Social Engineering oder Phishing–Methoden)
2. Der Switcher ändert den Token des Benutzers. Hierbei wird der Synchronisationssoftware der Token vom Angreifer »injiziert« (first switch) und anschließend der Original-Token vom Opfer in den nun vom Angreifer kontrollierten Synchronisationsordner kopiert. (a) wird inaktiv, (b) wird aktiv.
3. Die Synchronisationssoftware synchronisiert nun den Token des Opfers zum Angreifer (b).
4. Der Angreifer kann sich nun mittels des gestohlenen Token mit dem Account des Opfers synchronisieren (c).
5. Anschließend wird der Switcher noch einmal ausgeführt, um beim Opfer wieder den ursprünglichen Synchronisationszustand herzustellen (second switch).

Der Ablauf in Abb. 2.3 zeigt den Quick Double Switch Attack Flow. Im Bericht der IMPERVA – Hacker Intelligence Initiative werden noch weitere Angriffe auf Basis dieses Verfahrens aufgezeigt (vgl. [8]).

Neben dem Dropbox–Client wurden auch die Synchronisationsapplikationen Microsoft OneDrive, Box und Google Drive untersucht. Diese verwenden zum Authentifizieren den offenen OAuth 2.0–Authentifizierungsstandard. Dropbox hingegen verwendet ein proprietäres Verfahren. Problematisch bei Dropbox ist, dass die gesamte Sicherheit von der HOST_ID (und HOST_INT) abhängt. Hat ein Angreifer diese erbeutet, so kann er auch über den Dropbox–Webzugang sämtliche administrativen Aufgaben durchführen.

Laut Meinung der Autoren von »brig«, sowie auch vieler Sicherheitsexperten, wird beim Einsatz proprietärer Software die Sicherheit untergraben, da bei proprietärer Software explizit eingebaute Hintertüren nicht ausgeschlossen werden können und es auch keine einfache Möglichkeit der Prüfung auf solche durch den Endbenutzer gibt.

Insbesondere hat die Veröffentlichung der Snowden-Dokumente weiterhin zu der Schlussfolgerung geführt, dass der Einsatz von Freier Software empfehlenswerter ist. Bekannte Sicherheitsexperten wie *Bruce Schneier*¹²¹³ oder auch *Rüdiger Weis* sehen Freie Software als eine der wenigen Möglichkeiten, der Überwachung durch Geheimdienste (oder auch anderen Institutionen) entgegen zu wirken. Weiterhin kann Kryptographie dank Freier Software von unabhängigen Sicherheitsforschern bewertet werden.

Auch wenn für viele Benutzer die Geheimhaltung der Software und Infrastruktur auf den ersten Blick als sicherer erscheinen mag, widerspricht sie dem Kerckhoffs'schen Prinzip, bei welchem die Sicherheit eines System nur von der Geheimhaltung des Schlüssels, jedoch nicht von der Geheimhaltung weiterer Systemelemente abhängen sollte. Die Vergangenheit hat beispielsweise beim GSM-Standard oder beim DVD-Kopierschutz CSS¹⁴ gezeigt, dass durch die Geheimhaltung von Systemkomponenten erfolgreiche Angriffe, höchstens erschwert, jedoch nicht unterbunden werden können (vgl. [9], S. 11 und [10], S. 23).

Abgesehen von den Snowden-Enthüllungen, gibt es für den Endverbraucher viel näherliegendere Gefahren, welche die Daten und Privatsphäre gefährden. Neben dem soeben genannten Dropbox Datenleck, welches rund 70 Millionen Benutzerdaten betraf und über fast vier Jahre unentdeckt war, gibt es immer wieder Probleme mit zentralen Diensten. Ein Ausschnitt von bekannt gewordenen Vorfällen in letzter Zeit:

Datenlecks:

- ▶ Datenleck bei Dropbox¹⁵
- ▶ Google Drive Datenleck¹⁶
- ▶ Microsoft OneDrive Datenleck¹⁷
- ▶ iCloud-Hack auf private Fotos von Prominenten¹⁸

Weitere Probleme:

- ▶ Dropbox-Client greift auf Daten außerhalb des Sync-Ordners zu¹⁹.
- ▶ Microsoft synchronisiert Bitlocker-Schlüssel (Festplattenverschlüsselung) standardmäßig in die Cloud²⁰
- ▶ Dropbox akzeptiert beliebige Passwörter über mehrere Stunden²¹
- ▶ Ausfallzeit über zwei Stunden²²

¹²Defending Against Crypto Backdoors: https://www.schneier.com/blog/archives/2013/10/defending_again_1.html

¹³How to Remain Secure Against the NSA: https://www.schneier.com/blog/archives/2013/09/how_to_remain_s.html

¹⁴Cryptanalysis of Contents Scrambling System: <http://www.cs.cmu.edu/~dst/DeCSS/FrankStevenson/analysis.html>

¹⁵Dropbox Datenleck: <https://www.heise.de/security/meldung/Dropbox-bestätigt-Datenleck-1656798.html>

¹⁶Google Drive Datenleck:

<https://www.heise.de/security/meldung/Auch-Google-schliesst-Datenleck-im-Cloud-Speicher-2243366.html>

¹⁷OneDrive Datenleck: <https://www.heise.de/security/meldung/Microsoft-dichtet-OneDrive-Links-ab-2227485.html>

¹⁸iCloud-Hack:

https://de.wikipedia.org/w/index.php?title=Hackerangriff_auf_private_Fotos_von_Prominenten_2014&oldid=159942418

¹⁹Dropbox-Schnüffelverdacht:

<http://www.heise.de/security/meldung/Dropbox-unter-Schnueffelverdacht-2565990.html>

²⁰Bitlocker-Schlüssel werden standardmäßig in die Cloud synchronisiert:

<https://theintercept.com/2015/12/28/recently-bought-a-windows-computer-microsoft-probably-has-your-encryption-key/>

²¹Dropbox-Authentifizierungs-Bug:

<https://techcrunch.com/2011/06/20/dropbox-security-bug-made-passwords-optional-for-four-hours/>

²²Dropbox-Ausfall: <https://www.heise.de/security/meldung/Dropbox-Ausfall-war-kein-Angriff-2083688.html>

Auch wenn viele Unternehmen ihre Priorität nicht in der Sicherung ihrer Daten sehen mögen, sollten die Folgekosten von Datenlecks nicht unterschätzt werden. Laut einer jährlich durchgeföhrten Studie vom *Ponemon Institute* belaufen sich die Kosten im Zusammenhang mit Datenlecks auf mehrere Millionen Dollar (vgl. [11]). Die Tendenz ist von Jahr zu Jahr steigend wenn man die Berichte aus dem jeweiligen Vorjahr zuzieht.

Abgesehen von den Datenlecks verschiedener Cloud-Speicher-Anbieter, haben zentrale Dienste immer wieder Probleme mit größeren Datenlecks. Welcher Dienst und welche Daten betroffen sind, sammelt der Sicherheitsforscher *Troy Hunt* auf seiner Webseite²³.

2.1.5. Private Cloud

Weiterhin gibt es bei der Cloud-Speicher-Lösung auch die Möglichkeit, einen eigenen Cloud-Speicher aufzusetzen. Hierfür wird oft die Open-Source-Lösung Owncloud genommen. Der Nachteil hierbei ist, dass der Benutzer selbst für die Bereitstellung der Infrastruktur verantwortlich ist. Für Unternehmen mag die Owncloud durchaus interessant sein, für die meisten Privatanwender ist der Aufwand höchstwahrscheinlich zu hoch. Weiterhin haben Endanwender in der Regel nicht das nötige Know-How, welches für das Betreiben eines Cloud-Speicher-Dienstes essentiell ist.

2.1.6. Datenaustausch über dezentrale Lösungen

2.1.6.1. Funktionsweise dezentraler Dienste

Der dezentrale Bereich klassifiziert sich durch den Dateiaustausch, welcher in der Regel ohne eine zentrale Instanz auskommt. Es handelt sich hierbei um Systeme aus dem Bereich des Peer-to-Peer-Modells (P2P)²⁴. Eines der frühen Peer-to-Peer-Protokolle ist das Napster-Protokoll der gleichnamigen Anwendung Napster, welche Ende der 90' Jahre für den Tausch von Musik verwendet wurde. Später sind weitere Peer-to-Peer-Protokolle wie das Multisource-File-Transfer-Protocol oder das BitTorrent-Protokoll hinzugekommen.

Abb. 2.4 zeigt schematisch den Austausch von Daten in einem dezentralen Netzwerk. Bei einem dezentralen System liegen die Daten in der Regel nur auf den Rechnern der Benutzer. Die Speicherung auf zentralen Speicher-Servern wie bei den zentralen Diensten ist nicht vorgesehen, jedoch aufgrund der Architektur realisierbar.

Bei der Nutzung eines dezentralen Netzwerks zum Austausch beziehungsweise zur Synchronisation von Daten muss der Benutzer in der Regel eine spezielle Software installieren und einen Synchronisationsordner, wie bei den zentralen Diensten definieren. Dieser Ordner wird dem Netzwerk bekannt gemacht. Je nach eingesetztem Protokoll, variiert die Funktionsweise und Sicherheit.

²³Gesammelte Informationen zu Datenlecks: <https://haveibeenpwned.com/>

²⁴Peer-to-Peer: <https://de.wikipedia.org/w/index.php?title=Peer-to-Peer&oldid=159040176>

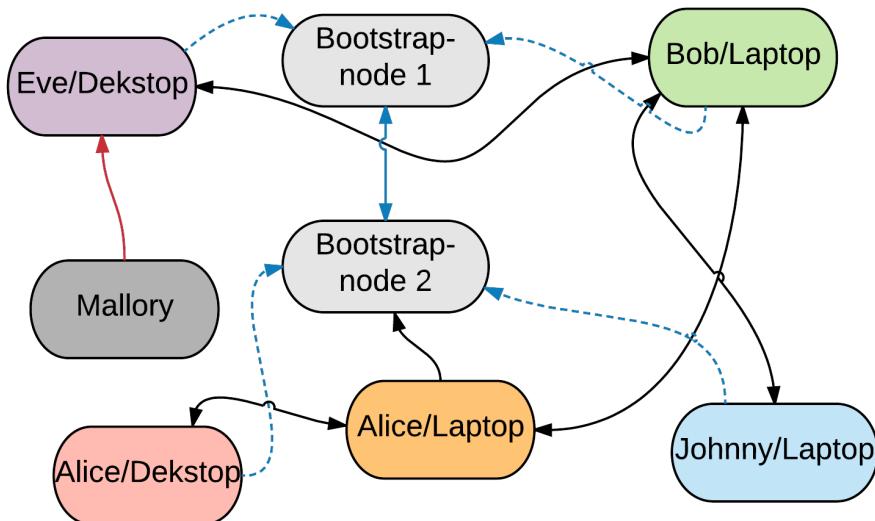


Abbildung 2.4.: Dezentraler Datenaustausch über Peer-to-Peer-Kommunikation. Es existiert keine zentrale Instanz, jeder Peer im Netzwerk ist gleichberechtigt.

Die dezentralen Systeme unterliegen in der Regel keiner Regulierung durch eine zentrale Instanz. Je nach verwendeter Technologie zum Datenaustausch, existieren sogenannte rendezvous hosts²⁵, welche für die initiale Konfiguration und als »Einstiegspunkt« benötigt werden. Hier unterscheiden sich die verschiedenen Protokolle und Netzwerke voneinander.

Ein bekannter Vertreter der P2P-Protokolle ist BitTorrent²⁶. Das Protokoll kommt beispielsweise bei der Verbreitung von Software, Computerspielen (HumbleIndieBundle.com), dem Blender Movie-Projekten, Linux-Distributionen, der Verteilung von Updates (Windows 10), bei diversen Spieleherstellern und auch anderen Anwendungen zum Einsatz²⁷.

Ein Vorteil bei den dezentralen Systemen ist, dass es im Vergleich zu zentralen Architekturen keinen Single Point Of Failure²⁸ gibt. Ein weiterer Unterschied zur zentralen Lösung ist bei dezentralen Netzwerken der Datenfluss. Die Daten werden nicht von einer zentralen Instanz »besorgt«, sondern liegen im jeweiligen Netzwerk, verteilt auf die Netzwerkteilnehmer (peers). Jeder Teilnehmer des Netzwerks fungiert in der Regel als Client und als Server. Daten werden beim Austausch nicht zwangsläufig von einem einzelnen Teilnehmer geladen, sondern von einer Gruppe aus Teilnehmern, welche die gleiche Datei besitzen, siehe Abb. 2.5.

²⁵Rendezvous Host/Bootstrapping node:

https://en.wikipedia.org/w/index.php?title=Bootstrapping_node&oldid=693889298

²⁶BitTorrent: <https://de.wikipedia.org/w/index.php?title=BitTorrent&oldid=160095352>

²⁷BitTorrent Einsatzgebiete: <https://en.wikipedia.org/w/index.php?title=BitTorrent&oldid=761280798#Adoption>

²⁸Single Point of Failure: https://de.wikipedia.org/w/index.php?title=Single_Point_of_Failure&oldid=156306981

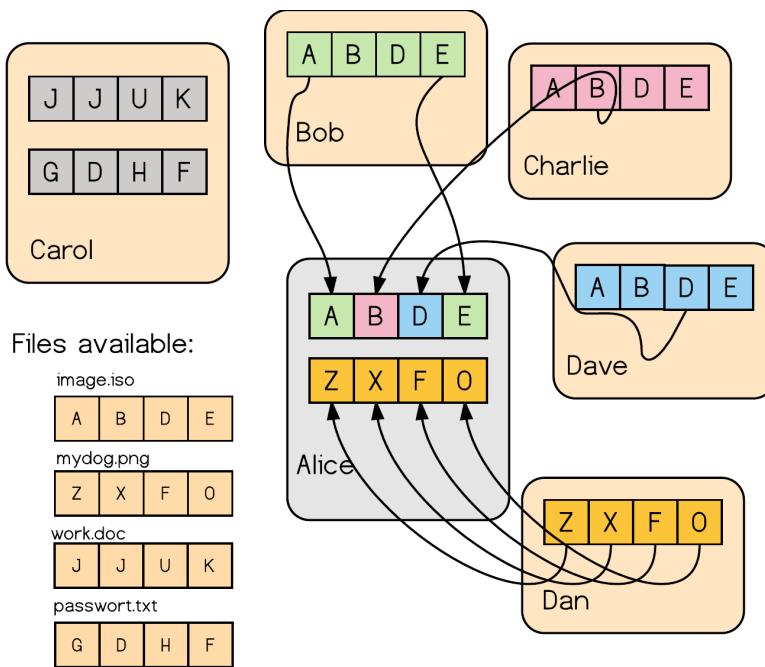


Abbildung 2.5.: Zeigt einen Swarm. *Alice* lädt die Datei *image.iso* von mehreren Teilnehmern gleichzeitig, die Datei *mydog.png* jedoch nur von *Dan*.

Aktuell verbreitete Peer-to-Peer-Protokolle:

- ▶ BitTorrent (Filesharing allgemein)
- ▶ Skype-Protokoll (VOIP-Telephonie)

Früher eingesetzte Peer-to-Peer-Filesharing-Protokolle:

- ▶ Direct Connect
- ▶ Multisource File Transfer Protocol (Einsatz: eDonkey2000, Overnet)
- ▶ Fasttrack (Einsatz: Kazaa, Grokster, iMesh, Morpheus file sharing)
- ▶ Gnutella (Einsatz: Gnutella Client)
- ▶ Napster

2.1.6.2. Synchronisationssoftware

Zu den Vertretern der etablierten dezentralen Systeme gibt es im Vergleich zu den Cloud-Speicher-Anbietern nur wenige Produkte, welche für die Synchronisation von Daten beziehungsweise den Austausch von Dokumenten eingesetzt werden können.

Bekannte Lösungen sind:

- ▶ Resilio (ehem. BitTorrent-Sync, proprietär)
- ▶ Infinit (proprietär, [12])
- ▶ git-annex (Open-Source)
- ▶ Syncthing (Open-Source)
- ▶ Librevault (Open-Source)

2.1.6.3. Sicherheit von Peer-to-Peer-Synchronisationsanwendungen

Bei den bekannten Vertretern des BitTorrent-Netzwerks wie dem BitTorrent-Client werden die Daten in der Regel unverschlüsselt übertragen und gespeichert. Auch eine Authentifizierung findet in der Regel nicht statt. Die Benutzer geben die Daten mit ihrem Synchronisationsordner automatisch für alle Teilnehmer des Netzwerks zum Teilen frei.

Wie bei zentralen Diensten, ist es auch bei dezentralen Netzwerken schwierig die Sicherheit zu beurteilen. Diese hängt in diesem Fall nicht zwangsläufig von einem Dienst-Anbieter ab, sondern vielmehr von der Umsetzung der Software, der Infrastruktur des Netzwerks, der Sicherung der Daten (verschlüsselte Speicherung, verschlüsselte Übertragung) und der Möglichkeit, einen Kommunikationspartner zu authentifizieren.

Resilio-Sync (ehemals BitTorrent Sync), verwendet eine modifizierte Variante des BitTorrent-Protokolls²⁹. Alle Daten werden laut Hersteller zusätzlich symmetrisch mit AES-128 (siehe Abschnitt 4.2.1) verschlüsselt übertragen. Die getestete Version entspricht der Standardversion, welche kostenfrei benutzbar ist, jedoch nur einen eingeschränkten Funktionsumfang bietet. Weiterhin gibt es eine Pro-Version, welche selektive und kollaborative Synchronisation ermöglicht.

Bei Resilio (Webbasierte GUI³⁰) werden Daten mittels verschiedener Schlüssel synchronisiert. Beim Anlegen eines Synchronisationsordners erscheinen dem Benutzer folgende Schlüssel, über welche er den Synchronisationsordner mit anderen Synchronisationspartnern teilen kann. Anhand des gewählten Schlüssels wird folgende Funktionalität beim Synchronisieren realisiert:

- ▶ Nur Leserechte
- ▶ Lese- und Schreibrechte
- ▶ Verschlüsselte Speicherung

Mittels dieser Schlüssel lässt sich die Synchronisation mit anderen Peers/Synchronisationsordnern steuern. Neue Peers können über das Teilen eines Schlüssels hinzugefügt werden. Die Anwendung macht einen undurchsichtigen Eindruck. Eine 2014 durchgeführte Analyse von BitTorrent-Sync auf der Hackito kommt aufgrund mehrerer Sicherheits- und Designprobleme zur Einschätzung³¹, dass BitTorrent-Sync nicht für sensible Daten verwendet werden sollte.

Infinit ist eine weitere proprietäre Lösung, welche es ermöglicht, Dateien zwischen verschiedenen Benutzern, ohne Server-Instanz, auszutauschen. Die Basis für Infinit stellt [12] dar. Bei Infinit findet bei der Installation der Anwendung eine Registrierung statt. Diese Daten (Benutzername/E-Mail-Adresse) können anschließend verwendet werden, um mit der Anwendung andere Infinit-Partner zu finden oder von diesen gefunden zu werden. Anschließend kann eine Datei über eine Drag & Drop-Fläche an den gefundenen Partner gesendet werden. Es ist unklar, welche Daten/Metadaten an die Infinit-Plattform übertragen werden. Infinit wirbt mit »point-to-point encryption« und »bank-level encryption algorithms such as AES-256 and RSA 2048«. Eine Authentifizierung des

²⁹Inoffizielle Protokoll-Spezifikation <https://forum.resilio.com/topic/21338-inofficial-protocol-specification/>

³⁰Grafische Benutzeroberfläche:

https://de.wikipedia.org/w/index.php?title=Grafische_Benutzeroberfl%C3%A4che&oldid=155859783

³¹Security analysis of BTsync:

<http://2014.hackitoergosum.org/bittorrentsync-security-privacy-analysis-hackito-session-results/>

Kommunikationspartners findet rudimentär anhand von Benutzernamen/E-Mail statt. Daten werden lokal nicht verschlüsselt.

Syncthing basiert auf einem eigens entwickelten Protokoll, dem Block Exchange Protocol³². Der Syncthing-Client (Web-GUI Variante) ermöglicht das Hinzufügen von Synchronisationsordner und Remote Devices. Diese GUI ist standardmäßig von außen nicht zugreifbar, da sie auf »localhost only« läuft.

Die Peers werden durch eine eindeutige Device-ID identifiziert. Diese leitet sich aus einem asymmetrischen Schlüsselpaar (3072 Bit RSA³³) ab, welches beim ersten Start der Anwendung erstellt wird. Abgelegt wird ein privater Schlüssel und ein selbst signiertes Zertifikat. Der private Schlüssel scheint nicht geschützt zu sein:

```
$ ~/.config/syncthing » cat key.pem
-----BEGIN EC PRIVATE KEY-----
MIGkAgEBBDCQIMwVr730vKzoyHCbIqDoxNxAjKvdFYL+XnKk65GurCc9q2qiZJEU
zMNWSD+N/eCgBwYFK4EEACKhZANiAASJ0YZUMQVAuW8tT7DvuLFkanCw2gpgD1DE
P69XHqMS0MFg6ZwMqzv1V65WXQMOHfsNw/xKMagSGlyTh17W/Up0y2PPygUlj6H1
d0vMI0guPD9heeqYjU67R4Gx1HMj54=
-----END EC PRIVATE KEY-----
```

Das selbst signierte Zertifikat bringt keine zusätzliche Sicherheit, ermöglicht jedoch die Nutzung von Transport Layer Security (TLS)³⁴. Diese ID ist für jeden Teilnehmer eindeutig (aufgrund der asymmetrischen Kryptographie). Sie besteht aus einer kryptographischen Prüfsumme (SHA-256, siehe Abschnitt 4.4) eines eindeutigen kryptographischen Zertifikates, welches für die verschlüsselte Kommunikation und Authentifizierung zwischen den einzelnen Peers verwendet wird.

Weiterhin ist das aktuelle Design für Discovery Spoofing³⁵ anfällig. Das heißt, dass ein Angreifer der im Netzwerk mitliest, Device-IDs mitlesen kann und sich somit als ein bestimmter Peer ausgeben kann. Das würde einem Angreifer die Information liefern, mit welchen Peers sich eine bestimmte Device-ID synchronisiert. Mehr zu Device-IDs, sowie möglichen damit in Verbindung stehenden Problemen, findet sich in der offiziellen Syncthing-Dokumentation³⁶.

Eine lokale Verschlüsselung der Daten findet nicht statt. Schlüssel, welche die Device-ID eindeutig identifizieren, sind nicht weiter gesichert.

Librevault ist ein sich noch im Frühstadium befindlicher Prototyp. Die aktuell getestete Alphaverision ist beim Hinzufügen eines Synchronisationsordners reproduzierbar abgestürzt. Laut Projekt-Beschreibung scheint sich Librevault an Resilio/Syncthing zu orientieren. Weitere Details zur Spezifikation und Projektzielen sind auf dem Blog des Entwicklers³⁷ zu finden.

³²Block exchange protocol: <https://docs.syncthing.net/specs/bep-v1.html>

³³Syncthing-Keys: <https://docs.syncthing.net/dev/device-ids.html#keys>

³⁴Transport Layer Security: https://de.wikipedia.org/w/index.php?title=Transport_Layer_Security&oldid=160767422

³⁵Problems and Vulnerabilities: <https://docs.syncthing.net/dev/device-ids.html#problems-and-vulnerabilities>

³⁶Understanding Device-IDs: <https://docs.syncthing.net/dev/device-ids.html>

³⁷Librevault Entwicklerblog: <https://librevault.com/blog/>

git-annex ist ein sehr stark am git-Versionsverwaltungssystem orientiertes Synchronisationswerkzeug. Prinzipiell ist es für die Kommandozeile entwickelt worden, es existiert mittlerweile jedoch ein Webfrontend (Webapp).

git-annex verwaltet nur die Metadaten in git. Es funktioniert als git-Aufsatz, welcher es dem Benutzer ermöglicht, auch große binäre Dateien mittels git³⁸ zu verwalten, beziehungsweise zu synchronisieren. Zum Synchronisieren der Metadaten wird git verwendet, zum Synchronisieren der eigentlichen Daten wird git-annex genutzt. Es überträgt die Daten verschlüsselt mit rsync³⁹ über ssh⁴⁰. Mittels der git-Erweiterung gcrypt⁴¹ ist es möglich, vollständig verschlüsselte git-remotes anzulegen.

Neben normalen git-Repositories werden sogenannte special remotes⁴² unterstützt. Diese werden verwendet, um Daten auf ein System, auf welchem git nicht installiert ist, zu synchronisieren.

Neben der Verschlüsselung von git-remotes mit gcrypt gibt es auch die Möglichkeit, die Daten auf special remotes zu verschlüsseln. Hierfür gibt es die vier Verfahren:

- ▶ **hybrid encryption:** Gemeinsamer »shared key« wird mit einem »public key« verschlüsselt im Repository gespeichert.
- ▶ **shared encryption:** Gemeinsamer »shared key« wird im Klartext im Repository gespeichert.
- ▶ **public key encryption:** Hierbei wird der »public key« verwendet, zum Entschlüsseln benötigt man den »private key«.
- ▶ **shared public key encryption:** Wie beim »public key«-Verfahren, jedoch nur Besitzer des geheimen Schlüssel dürfen die Daten von einem special remote beziehen.

2.1.7. Ähnliche Arbeiten

Neben den genannten dezentralen Projekten existieren weitere Ansätze im Bereich des dezentralen Datenaustausches:

Infinit: Neben dem Datei-Austauschwerkzeug existiert ein dezentrales Dateisystem. Die Arbeit (vgl. [12]) von Julien Quintard setzt eine Ebene tiefer als »brig« an und befasst sich mit den Eigenschaften und dem Entwurf eines dezentralen Dateisystems.

Bazil⁴³ ist ein weiteres Projekt, welches als dezentrales Dateisystem entwickelt wird. Bazil hat ähnliche Ziele wie »brig«. Es verschlüsselt die Daten, arbeitet dezentral und hat auch Features wie Deduplikierung von Daten oder Snapshots. Weiterhin ist die Go-FUSE-Bibliothek⁴⁴ aus dem Projekt entstanden, welche auch von »brig« verwendet wird.

IPFS (InterPlanetary-File-System⁴⁵) ist ein relativ neuer dezentraler Ansatz, welcher verschiedene bekannte Technologien kombiniert. Dadurch lassen sich Schwächen aktuell genutzter Systeme und Protokolle abmildern oder gar vermeiden. In der aktuellen Implementierung ist das Projekt jedoch

³⁸Versionsverwaltungssystem git: <https://de.wikipedia.org/w/index.php?title=Git&oldid=161769023>

³⁹rsync: <https://de.wikipedia.org/w/index.php?title=Rsync&oldid=162199260>

⁴⁰Secure Shell: https://de.wikipedia.org/w/index.php?title=Secure_Shell&oldid=162392397

⁴¹gcrypt git addon: <https://spwhitton.name/tech/code/git-remote-gcrypt/>

⁴²git-annex special remotes: https://git-annex.branchable.com/special_remotes/

⁴³Projektseite: <https://bazil.org/>

⁴⁴Projektseite GO-FUSE: <https://bazil.org/fuse/>

⁴⁵InterPlanetary-File System: https://en.wikipedia.org/w/index.php?title=InterPlanetary_File_System&oldid=757419434

eher als fortgeschrittenen Prototyp anzusehen. Aufgrund des vielversprechenden Ansatzes, ist IPFS die Grundlage von »brig«.

2.2. Markt und Wettbewerber

Da der Cloud-Speicher-Markt sehr dynamisch und fragmentiert ist, ist es schwierig hier zuverlässige Daten zu finden. Laut einem Online-Beitrag der »Wirtschafts Woche«⁴⁶ gehören folgende Anbieter zu den größten Cloud-Speicher-Anbietern:

- ▶ Dropbox
- ▶ Apple iCloud
- ▶ Microsoft OneDrive
- ▶ Google Drive

In Deutschland gehört Dropbox zu den bekannteren Anbietern, Apple iCloud ist in erster Linie für Mac-Benutzer interessant.

Im Open-Source-Bereich können die Projekte

- ▶ OwnCloud
- ▶ NextCloud (OwnCloud fork)

als zentrale Konkurrenz-Produkte angesehen werden.

Diese zentralen Systeme stellen in gewisser Weise ein indirektes Konkurrenz-Produkt zu »brig« dar. Als weitere Wettbewerber können auch die bereits genannten dezentralen Synchronisationswerkzeuge angesehen werden.

2.3. Closed-Source vs. Open-Source

Es ist schwierig zu beantworten, ob Open-Source-Software als sicherer anzusehen ist. Es spielen hierbei sehr viele Faktoren eine Rolle, weswegen eine eindeutige Aussage nicht möglich ist. Oft wird mit »Linus's Law« — *Given enough eyeballs, all bugs are shallow.*⁴⁷ — für die Sicherheit freier Software argumentiert. Software-Bugs wie

- ▶ Debian Random Number Generator Bug⁴⁸
- ▶ OpenSSL Heartbleed Bug⁴⁹

haben jedoch gezeigt, dass auch freie Software⁵⁰ (beziehungsweise das Open-Source-Modell) von »Sicherheitskatastrophen« nicht verschont bleibt. Die besagten Fehler wurden erst nach mehreren Jahren entdeckt und es ist unbekannt, ob und in welchem Ausmaß diese ausgenutzt werden konnten. Diese Beispiele zeigen, dass man sich auf das »More eyeballs principle« allein nicht verlassen darf.

⁴⁶Größte Cloud-Speicher-Anbieter:

<http://www.wiwo.de/unternehmen/it/cloud-wer-sind-die-groessten-cloud-anbieter-und-was-kosten-sie/11975400-7.html>

⁴⁷Linus's Law: https://en.wikipedia.org/w/index.php?title=Linus%27s_Law&oldid=761677049

⁴⁸Random Number Bug in Debian Linux: https://www.schneier.com/blog/archives/2008/05/random_number_b.html

⁴⁹Heartbleed: <https://www.schneier.com/blog/archives/2014/04/heartbleed.html>

⁵⁰Freie Software: https://de.wikipedia.org/w/index.php?title=Freie_Software&oldid=162291001

Weiterhin kann das Open-Source-Modell auch dazu verwendet werden, um automatisiert nach Sicherheitslücken im Quelltext auf GitHub bei jedem Commit zu suchen. Der Google Softwareentwickler Kees Cook verweist auf diese Problematik auf dem aktuellsten *Linux Security Summit 2016*⁵¹.

Untersuchungen von Closed- und Open-Source-Projekten haben gezeigt, dass es keine signifikanten Unterschiede bezüglich der Sicherheit zwischen den beiden Entwicklungsmodellen gibt. Open-Source scheint jedoch ein extrem schlechtes Patch-Management seitens des Herstellers zu verhindern (vgl. [13]). Wenn man davon ausgeht, dass schlechte Quellcode-Qualität zu mehr Fehlern und somit zu mehr Sicherheitslücken führt, kann man auch einen Teil der Sicherheit über die Quellcode-Qualität definieren. Eine Studie, welche die Quellcode-Qualität von vier großen Betriebssystemkernen (Linux, FreeBSD, Solaris, Windows) mit Hilfe verschiedener Metriken vergleicht, kommt zum Ergebnis, dass es keine signifikanten Unterschiede zwischen dem Closed-Source- und Open-Source-Softwareentwicklungsmodell bezüglich Quellcode-Qualität gibt (vgl. [14]). Neuere Studien widersprechen hier und attestieren Open-Source-Software-Projekten eine tendenziell bessere Quellcode-Qualität als Closed-Source-Projekten (vgl. [15] und [16]).

Durch Freie Software hat der Verbraucher jedoch immer die Möglichkeit, den Quelltext zu validieren und auch an die eigenen Sicherheitsbedürfnisse anzupassen. Wenn man beispielsweise wissen möchte, ob und wie die Festplattenverschlüsselungssoftware (in diesem Fall Cryptsetup/LUKS) den Volumeschlüssel aus dem Speicher entfernt, kann man sich den Quelltext auf GitHub anschauen. Im Fall von LUKS wird der Speicherbereich mit Nullen überschrieben⁵².

2.4. Gesellschaftliche und politische Aspekte

Seit den Snowden-Enthüllungen⁵³ ist offiziell bekannt, dass Unternehmen rechtlich gezwungen werden können, personenbezogene Daten an Behörden weiter zu geben. Betrachtet man die großflächige Verbreitung von Facebook und WhatsApp, macht es den Anschein, dass viele Menschen für die Thematik der Privatsphäre nicht genug sensibilisiert sind. Diskutiert man über sichere Alternativen oder macht Personen auf den problematischen Datenschutz von zentralen Diensten wie Facebook, Dropbox und Co. aufmerksam, bekommt man oft das Argument »Ich habe nichts zu verbergen!« zu hören.

Edward Snowden sagte, dass dies der falsche Ansatz ist, weil dadurch das Grundprinzip der Demokratie »umgedreht« wird. Alle Macht geht vom Volke aus, aber wenn das Volk überwacht wird – kann dann noch Demokratie gewährleistet werden? Ist das Recht auf Privatsphäre mit dem Recht auf freie Meinungsäußerung vergleichbar?

Arguing that you don't care about the right to privacy because you have nothing to hide is no different than saying you don't care about free speech because you have nothing to say. – Edward Snowden

⁵¹ Status of the Kernel Self Protection Project:

<http://events.linuxfoundation.org/sites/events/files/slides/KernelSelfProtectionProject-2016.pdf>

⁵² LUKS Volume-Key Free-Funktion:

<https://github.com/mhfan/cryptsetup/blob/ae9c9cf369cb24ac5267376401c80c2c40ada6a2/lib/volumekey.c#L46>

⁵³ Globale Überwachungs- und Spionageaffäre:

https://de.wikipedia.org/w/index.php?title=Globale_%C3%9Cberwachungs-_und_Spionageaff%C3%A4re&oldid=161594120

Die eigene Privatsphäre aus der »Ich habe nichts zu verbergen«-Perspektive zu betrachten, ist ein diskussionswürdiger Ansatz. Es geht bei der Privatsphäre nicht darum, irgend etwas illegales zu verbergen, es geht um den Schutz der eigenen Persönlichkeit und um die Wahrung der eigenen Persönlichkeitsrechte. Die Privatsphäre stellt ein Grundrecht dar und ist in allen modernen Demokratien verankert⁵⁴.

Ob man etwas zu verbergen hat, wird weiterhin durch den Beobachter entschieden. In unserem heutigen gesellschaftlichen Kontext heißt das, dass sich die politische Lage auch jederzeit ändern kann. Durch eine Änderung auf politischer Ebene kann eine vorher als unschuldig geglaubte Person plötzlich aufgrund ihrer politischen Einstellung, ihrem Glauben oder der sexuellen Orientierung zu einer politisch verfolgten Minderheit gehören.

Dass ein Datensatz über Leben und Tod entscheiden kann, zeigt die Geschichte. Anfang des 19ten Jahrhunderts wurden in Amsterdam alle Bürger bereitwillig in einem Bevölkerungsregister erfasst. Die dort gespeicherten Informationen enthielten Beziehungsstatus, Beruf und Religionszugehörigkeit. Die Bürger der Stadt glaubten sicherlich an ihre Unschuld und hatten nichts zu verbergen. Als im Mai 1940 die deutschen Besatzer einmarschierten, konnten sie anhand des Registers alle jüdischen Bürger in kürzester Zeit identifizieren. Diese wurden in Konzentrationslager deportiert. In diesem Fall hat ein einfacher Datensatz über Leben und Tod entschieden (vgl. [17], S. 3).

Nicht nur Daten, die von öffentlichen Einrichtungen erhoben werden sind als problematisch anzusehen. Auch Daten, die wir jeden Tag unfreiwillig in der digitalen Welt zurücklassen, werden täglich missbraucht — Personen die sich als unschuldig glaubten, geraten plötzlich ins Visier von Ermittlern. Ermittlungsfehler, welche unschuldige Menschen hinter Gittern bringen oder zu Selbstmord treiben, sind hier leider an der Tagesordnung⁵⁵.

Laut Meinung des Autors ist es allgemein sinnvoll und wichtig, Daten heutzutage vorwiegend verschlüsselt zu speichern. Der Wert der Daten wird vom Benutzer oft nicht korrekt eingestuft. Das Paradebeispiel hierfür ist die oben genannte geschichtliche Entwicklung.

Auch wenn bei mobilen Geräten noch Einsicht herrscht (Diebstahl, Verlust des Smartphone), wird in anderen Fällen oft argumentiert, dass sensible Daten bei Bedarf sicher gelöscht werden können. Auch wenn dies für die meisten Fälle korrekt sein mag, sind folgende Fälle weiterhin als problematisch anzusehen:

- ▶ Diebstahl auch bei nicht-mobilen Geräten möglich
- ▶ Garantiefall (defekte Festplatte kann vielleicht nicht mehr gelöscht werden)
- ▶ Software-Wipe-Tools bei SSDs⁵⁶ sind problematisch (vgl. [18])

⁵⁴ Privatsphäre: <https://de.wikipedia.org/w/index.php?title=Privatsph%C3%A4re&oldid=161564657>

⁵⁵ Datenmissbrauch und Irrtümer:

<http://www.daten-speicherung.de/index.php/faelle-von-datenmissbrauch-und-irrtuemern/>

⁵⁶ Solid-State-Drive: <https://de.wikipedia.org/w/index.php?title=Solid-State-Drive&oldid=160808976>

3.1. Einleitung

Die Betrachtung des aktuellen wissenschaftlichen und technischen Standes zeigt, dass die Thematik im Detail komplex und kompliziert ist. Nichtsdestotrotz ergeben sich gewisse Mindestanforderungen, die für die Entwicklung einer sicheren und dezentralen Dateisynchronisationslösung nötig sind. Um eine möglichst gute Sicherheit und Usability zu gewährleisten, muss die Software und der Softwareentwicklungsprozess gewissen Mindestanforderungen genügen.

3.2. Usability

Die Usability von Software ist teilweise subjektiv und ist stark von dem Erfahrungshorizont des Nutzers abhängig. Unter Usability ist im Allgemeinen die Gebrauchstauglichkeit/Benutzerfreundlichkeit¹² eines Systems zu verstehen. Bereits bekannte Konzepte werden oft als intuitiv empfunden, neue Konzepte hingegen oft nur mühsam vom Benutzer angenommen. Die Umstellung der Benutzeroberfläche von Windows 7 zu Windows 8 (vgl. [19]) ist hierfür ein gutes Beispiel.

Einen weiteren Punkt zeigt die Praxis. Zwar gibt es seit Jahrzehnten Software zur sicheren Kommunikation, wie beispielsweise OpenPGP, jedoch hat sich das Konzept wenig durchgesetzt. Über die genauen Gründe, warum sich PGP nicht durchgesetzt hat, kann man sich streiten. Laut Meinung des Autors, liegt es einerseits an der hohen Komplexität beziehungsweise Einstiegshürde, andererseits zeigen Umfragen, dass eine gewisse Gleichgültigkeit gegenüber dem Schutz der eigenen Privatsphäre vorherrscht³.

Um dieser Problematik möglichst aus dem Weg zu gehen, sollen folgende Anforderungen umgesetzt werden:

Bekannte Umgebung: Der Benutzer soll nach der Installation der Software weiterhin seine bekannte Umgebung in Form eines Ordners vorfinden. In diesem Ordner kann er seine Daten sicher speichern und synchronisieren, wie er es bereits von bekannten Cloud-Speicher-Diensten, wie beispielsweise Dropbox, kennt.

Versteckte Sicherheit: Die Sicherheitskomplexität soll möglichst hinter nur »einem Passwort« versteckt werden. Dies soll sicherstellen, dass der Benutzer nicht unnötig mit einer unangenehmen, beziehungsweise vorgangsfremden Thematik konfrontiert wird.

Entkopplung der Metadaten: Die Datenübertragung soll entkoppelt von der Metadatenübertragung passieren. Dies ermöglicht eine gezielte Synchronisation bestimmter Daten und ermöglicht dem

¹Benutzerfreundlichkeit: <https://de.wikipedia.org/w/index.php?title=Benutzerfreundlichkeit&oldid=159056605>

²Gebrauchstauglichkeit: [https://de.wikipedia.org/w/index.php?title=Gebrauchstauglichkeit_\(Produkt\)&oldid=159056626](https://de.wikipedia.org/w/index.php?title=Gebrauchstauglichkeit_(Produkt)&oldid=159056626)

³Umfrage DIVSI: <https://www.divsi.de/abhoeren-egal-ich-habe-nichts-zu-verbergen/>

Benutzer so Ressourcen (Speicherplatz und Zeit) zu sparen. Dabei sollen die Daten bei der Übertragung und Speicherung verschlüsselt sein.

3.3. Sicherheit

Wie bereits unter [Abschnitt 2.1.2](#) erwähnt, ist Sicherheit ein sehr weitläufiger Begriff und stark von einem bestimmten Angriffszenario abhängig.

Eine Software zur dezentralen Dateiverteilung benötigt Sicherheitskonzepte, welche folgende Punkte gewährleisten:

- ▶ *Vertraulichkeit*: Kein Zugriff auf Daten durch unbefugte Personen.
- ▶ *Integrität*: Manipulation von Daten sind erkennbar.
- ▶ *Authentifizierung*: Kommunikationspartner sind eindeutig identifizierbar.

Die beiden unter [Abschnitt 2.1.2](#) gelisteten Punkte Autorisierung und Verfügbarkeit stellen keine erfüllbaren Sicherheitsaspekte für eine dezentrale Softwarelösung dar.

Weiterhin sollen sich die Anforderungen an die Sicherheit an den aktuell vorherrschenden und bewährten Sicherheitsstandards orientieren (vgl. [20]).

Die Entwicklung einer sicheren Software setzt einen sicheren und transparenten Entwicklungsprozess voraus. Wird eine Software produktiv eingesetzt, so ist ein stetiges Patchmanagement essentiell. Bei einer dezentralen sicheren Synchronisationslösung, wie »brig«, ist es weiterhin essentiell, dass ein durchdachtes Sicherheitskonzept existiert.

Bei Open-Source-Projekten kommt erschwerend hinzu, dass Benutzern und Entwicklern neben dem Vertrauen in die Software auch eine akzeptable Möglichkeit geboten werden muss, den Entwicklungsprozess sicher und transparent mitgestalten zu können.

Aus diesen Herausforderungen ergeben sich bezogen auf das Projekt folgende Mindestanforderungen sowie Fragestellungen:

- ▶ *Passwortmanagement* – Wie sieht für Benutzer und Entwickler ein sicheres Passwortmanagement aus?
- ▶ *Schlüsselmanagement* – Welche Konzepte sollen für die Verwaltung kryptographischer Schlüssel verwendet werden?
- ▶ *Authentifizierung* – Wie kann eine sichere Authentifizierung von Benutzern in einem dezentralen Netzwerk erfolgen?
- ▶ *Sichere Softwareverteilung* – Wie kann der Benutzer/Entwickler sicherstellen, keine Schadsoftware erhalten zu haben?
- ▶ *Sichere und transparente Entwicklungsumgebung* – Wie sieht eine transparente und sichere Entwicklungsumgebung aus?

Diese Anforderungen resultieren aus bestimmten Angriffszenarien, welche im [Abschnitt 5.2](#) detaillierter behandelt werden.

4.1. Einleitung

Software-Entwickler sind in der Regel keine Sicherheitsexperten. Nicht nur Fehler in der Software gefährden ganze Systeme und Benutzerdaten, sondern auch der fehlerhafte Einsatz von Kryptographie ist immer wieder für katastrophale Sicherheitsprobleme verantwortlich. Es ist *nicht trivial*, Kryptographie *korrekt* zu implementieren. Sogar der früher weit verbreitete Standard IEEE 802.11, WEP (Wired Equivalent Privacy), zur verschlüsselten drahtlosen Kommunikation, weist gleich mehrere Designschwächen auf. Eine Analyse¹ kommt zu der Einschätzung, dass kryptographische Primitiven missverstanden und auf ungünstige Art kombiniert wurden. Weiterhin ist es ein Hinweis dafür, dass man Experten aus dem Bereich der Kryptographie hätte einbeziehen sollen, um solche Fehler zu vermeiden (vgl. auch [21], S. 430).

Sogar bei Unternehmen, welche explizit mit *starker Kryptographie* für ihre Produkte werben² und auch für welche Kryptographie zum Tagesgeschäft gehört, machen immer wieder fatale Fehler bei der Implementierung ihrer Produkte.

Die oben genannten Beispiele zeigen, dass selbst Systeme, die von Experten entwickelt werden, genauso kritische Fehler aufweisen können. Das BSI (Bundesamt für Sicherheit in der Informationstechnik) hat aus diesem Grund einen Leitfaden (vgl. [20]) für die Implementierung kryptographischer Verfahren zusammengestellt. Im Leitfaden wird explizit darauf hingewiesen, dass der Leitfaden je nach Anwendungsfall nicht blind angewendet werden darf und dass bei sicherheitskritischen Systemen stets Experten zu Rate zu ziehen sind.

Folgend werden selektiv gewählte Sicherheitsprinzipien betrachtet, einerseits um zu sensibilisieren, andererseits aber auch, um einen sinnvollen Einsatz für »brig« definieren zu können.

4.2. Verschlüsselung

4.2.1. Symmetrische Verschlüsselungsverfahren

4.2.1.1. Grundlegende Funktionsweise

Abb. 4.1 zeigt die Verschlüsselung von Daten mittels symmetrischer Kryptographie. Bei symmetrischer Kryptographie wird der gleiche Schlüssel zum Ver- und Entschlüsseln der Daten verwendet.

Beim Datenaustausch über unsichere Netze, muss der Schlüssel zuerst zwischen den Kommunikationspartnern ausgetauscht werden. In Abb. 4.1 verschlüsselt Alice die Daten mit einem *gemeinsamen* Schlüssel. Anschließend sendet sie die verschlüsselten Daten an Bob, welcher den *gemeinsamen* Schlüssel verwendet, um die Daten wieder zu entschlüsseln.

¹WEP Analysis: <http://www.isaac.cs.berkeley.edu/isaac/wep-faq.html>

²Festplattenverschlüsselung: <http://www.heise.de/security/artikel/Verschüsselt-statt-verschlüsselt-270058.html>

Symmetrische Verfahren sind im Vergleich zu asymmetrischen Verfahren sehr ressourceneffizient. Die Grundlage für symmetrische Algorithmen stellen Manipulationen (Substitutionen, Permutationen³ oder Feistelrunden⁴) auf Bit-Ebene dar, welche ohne Schlüssel nicht effizient umkehrbar sind.

Das grundsätzliche Problem, welches bei der Anwendung symmetrischer Verschlüsselung besteht, ist der *sichere* Schlüsselaustausch.

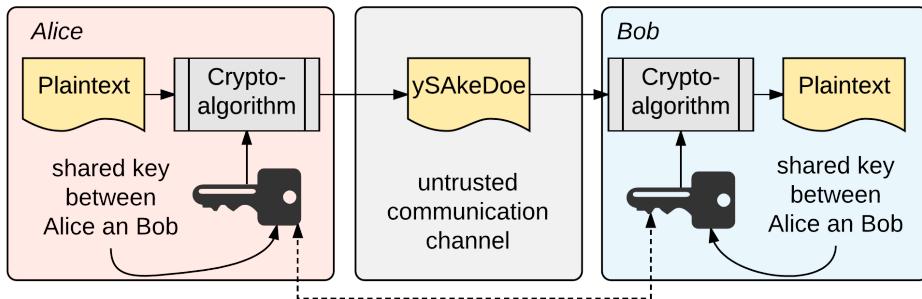


Abbildung 4.1.: Konzept beim Austausch von Daten über einen unsicheren Kommunikationsweg unter Verwendung symmetrischer Kryptographie. Alice und Bob teilen einen *gemeinsamen* Schlüssel, um die Daten zu ver- und entschlüsseln.

4.2.1.2. Unterschied zwischen Block- und Stromverschlüsselung

Das symmetrische Verschlüsseln unterteilt sich in die beiden Verschlüsselungsverfahren Stromverschlüsselung und Blockverschlüsselung. Bei der Stromverschlüsselung wird direkt jedes Zeichen (Bit) des Klartextes mittels eines kryptografischen Schlüssels direkt (XOR) in ein Geheimtextzeichen umgewandelt.

Bei der Blockverschlüsselung hingegen sind die Daten in Blöcke einer bestimmten Größe unterteilt. Die Verschlüsselung funktioniert auf Blockebene. Wie oder ob die Datenblöcke untereinander abhängig sind und welche Informationen bei der Verschlüsselung neben dem Schlüssel mit in die Verschlüsselung einfließen, bestimmt die sogenannte Betriebsart. Abb. 4.2 zeigt exemplarisch den Unterschied zwischen Strom- und Blockverschlüsselung.

³Substitutions-Permutations-Netzwerk:

<https://de.wikipedia.org/w/index.php?title=Substitutions-Permutations-Netzwerk&oldid=150385470>

⁴Feistelchiffre: <https://de.wikipedia.org/w/index.php?title=Feistelchiffre&oldid=159236443>

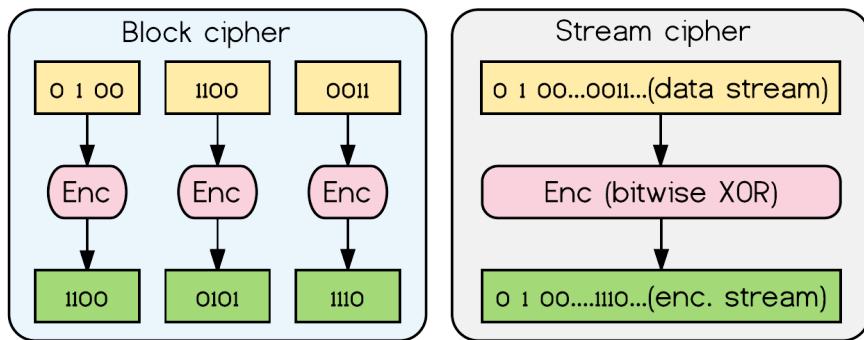


Abbildung 4.2.: Unterschied in der Arbeitsweise zwischen Block- und Stromchiffre. Die Blockchiffre verschlüsselt die Daten blockweise, eine Stromchiffre hingegen verschlüsselt den Datenstrom »on the fly«.

4.2.1.3. Betriebsarten der Blockverschlüsselung

Die Betriebsart beschreibt auf welche Art und Weise die Blöcke verschlüsselt werden. Dies ist insofern wichtig, da sich durch die Betriebsart die Eigenschaften und somit der Einsatzzweck ändern kann. Folgend zwei Betriebsarten zu besserem Verständnis:

Electronic Code Book Mode (ECB): Bei dieser Betriebsart werden die Klartextblöcke unabhängig voneinander verschlüsselt. Dies hat den Nachteil, dass gleiche Klartextblöcke immer gleiche Geheimtextblöcke, bei Verwendung des gleichen Schlüssels, ergeben. Abb. 4.3 zeigt eine Schwäche dieses Verfahrens.

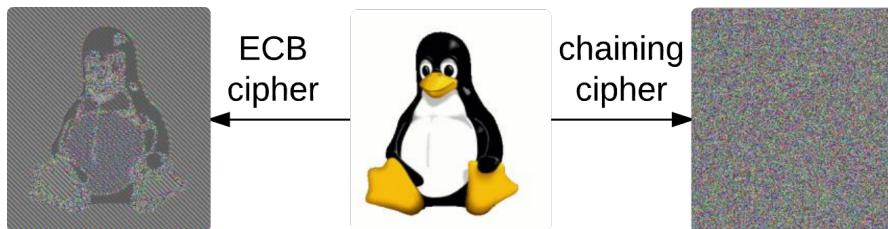


Abbildung 4.3.: Bild zur graphischen Verdeutlichung des ECB–Modus im Vergleich zu einem block chaining cipher.⁵

Cipher Feedback Mode (CFB): Beim CFB–Modus fließt, neben dem Schlüssel, der Geheimtextblock vom Vorgänger ein. Durch diese Arbeitsweise haben im Gegensatz zum ECB–Modus gleiche Klartextblöcke unterschiedliche Geheimtextblöcke. Weiterhin wird bei dieser Arbeitsweise aus der Blockverschlüsselung eine Stromverschlüsselung.

Abb. 4.4 zeigt den Unterschied zwischen den beiden genannten Modi.

⁵Bildquelle ECB: https://de.wikipedia.org/w/index.php?title=Electronic_Code_Book_Mode&oldid=159557291

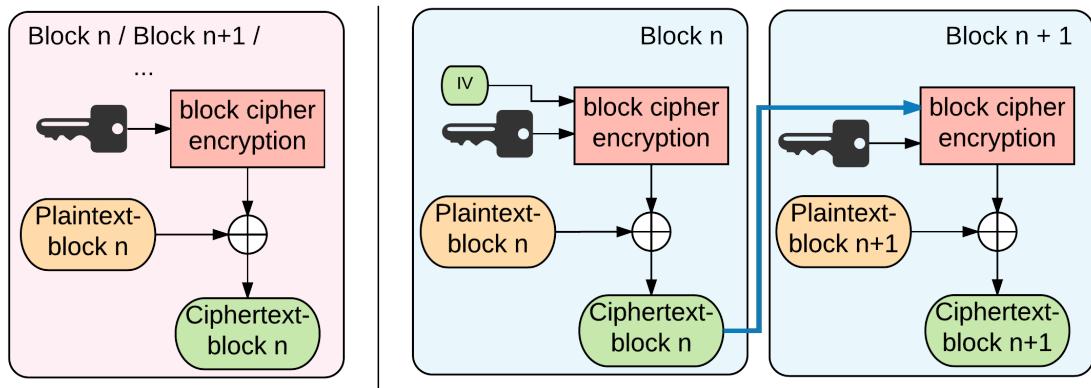


Abbildung 4.4.: ECB-Modus (links): Datenblöcke werden unabhängig voneinander verschlüsselt. CFB-Modus (rechts): Datenblöcke hängen beim Verschlüsseln voneinander ab.

Neben den genannten Betriebsarten gibt es noch weitere, die sich in der Funktionsweise unterscheiden, beziehungsweise für bestimmte Anwendungen konzipiert sind. Je nach Betriebsart ist ein paralleles Ver- und Entschlüsseln oder auch ein wahlfreier Zugriff möglich. Weiterhin variiert auch die Fehleranfälligkeit und Sicherheit. Tabelle 4.1 zeigt gängige Betriebsarten und ihre Eigenschaften.

Tabelle 4.1.: Laut ISO 10116 Standard definierte Betriebsarten für blockorientierte Verschlüsselungsalgorithmen.

Eigenschaft/Betriebsart	ECB	CBC	CFB	CTR	OFB
Verschlüsseln parallelisierbar	ja	nein	nein	ja	nein
Entschlüsseln parallelisierbar	ja	ja	ja	ja	nein
Wahlfreier Zugriff möglich	ja	ja	ja	ja	nein

4.2.1.4. Gängige Algorithmen, Schlüssellängen und Blockgrößen

Der ursprünglich seit Ende der 70er-Jahre verwendete DES (Data Encryption Standard), welcher eine effektive Schlüssellänge von 56 Bit hatte, war Ende der 90er-Jahre nicht mehr ausreichend sicher gegen Brute-Force-Angriffe. In einer öffentlichen Ausschreibung wurde ein Nachfolger, der Advanced Encryption Standard (kurz AES) bestimmt. Gewinner des Wettbewerbs sowie der heutige Quasi-Standard wurde der Rijndael-Algorithmus.

Neben dem bekannten AES (Rijndael)-Algorithmus, gibt es noch weitere Algorithmen, die heutzutage Verwendung finden. Zu den AES-Finalisten gehören weiterhin MARS, RC6, Serpent und der von Bruce Schneier entwickelte Twofish. Alle genannten Algorithmen arbeiten mit einer Blockgröße von 128 Bit und unterstützen jeweils die Schlüssellängen 128 Bit, 192 Bit und 256 Bit.

AES ist die aktuelle Empfehlung vom BSI, vgl. [20] S.22 f. Für weitere Details zum Thema symmetrische Kryptographie vgl. [21], S. 106 ff.

4.2.2. Asymmetrische Verschlüsselungsverfahren

4.2.2.1. Grundlegende Funktionsweise

Im Vergleich zur symmetrischen Verschlüsselung, werden bei der asymmetrischen Verschlüsselung die Daten mit einem unterschiedlichen Schlüssel ver- und entschlüsselt. Der Vorteil zu symmetrischen Verschlüsselung ist, dass die kommunizierenden Parteien *keinen gemeinsamen Schlüssel* kennen müssen.

Um Daten mittels asymmetrischer Verschlüsselung auszutauschen, müssen die beiden Kommunikationspartner *Alice* und *Bob* ein Schlüsselpaar, bestehend aus einem *privaten* und einem *öffentlichen* Schlüssel, erstellen. Anschließend tauschen beide Parteien den *öffentlichen* Schlüssel aus. Der *private* Schlüssel ist geheim und darf nicht weitergegeben werden. Abb. 4.5 zeigt die Funktionsweise bei asymmetrischer Verschlüsselung.

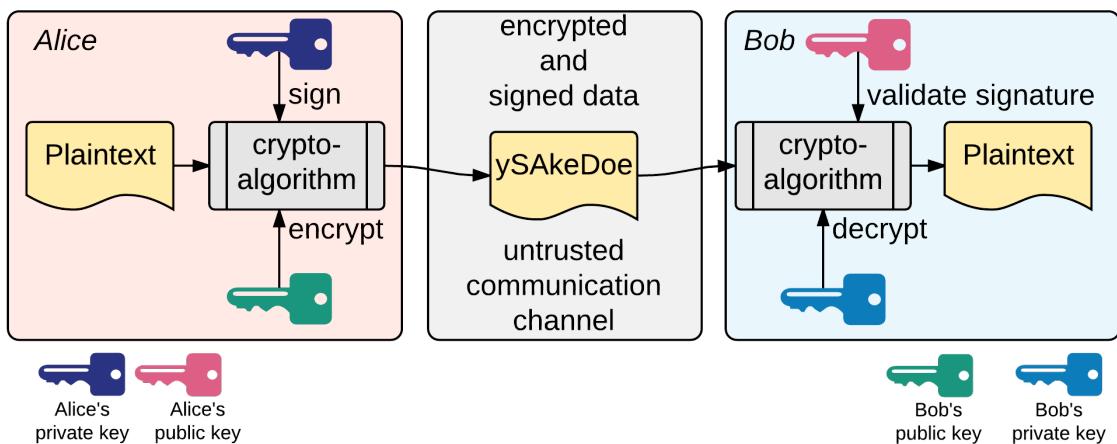


Abbildung 4.5.: Prinzip asymmetrischer Verschlüsselung. Verschlüsselt wird mit dem *öffentlichen* Schlüssel des Empfängers. Der Empfänger entschlüsselt mit seinem *privaten* Schlüssel die Nachricht. Die Signatur erfolgt mit dem *privaten* Schlüssel des Senders, validiert wird diese mit dem *öffentlichen* Schlüssel des Senders.

Im Unterschied zu symmetrischen Verfahren, beruht die asymmetrische Verschlüsselung auf der Basis eines mathematischen Problems, welches eine Einwegfunktion ist. Das heißt, dass die Berechnung in die eine Richtung sehr leicht ist, die Umkehrfunktion jedoch sehr schwierig zu berechnen ist. Die zugrundeliegenden mathematischen Probleme sind das Faktorisierungsproblem (RSA-Verfahren) großer Primzahlen und das diskrete Logarithmusproblem (ElGamal-Verfahren).

4.2.2.2. Gängige Algorithmen, Einsatzzwecke und Schlüssellängen

Zu den gängigen Algorithmen der asymmetrischen Verschlüsselungsverfahren gehören RSA und ElGamal. Beide Verfahren ermöglichen die Ver- und Entschlüsselung von Daten. Zu den gängigen Signaturverfahren gehören die RSA- und ElGamal-Signaturverfahren. Die RSA-Signatur basiert

direkt auf dem RSA–Verfahren, die ElGamal–Signatur auf einer modifizierten Form vom ElGamal, für weitere Details vgl. [21], S. 230 ff. und S. 240 f.

Weiterhin gibt es eine Variante des DSA–Verfahrens⁶, welche Elliptische–Kurven–Kryptographie verwendet, das ECDSA (elliptic curve DSA). Die Verfahren auf elliptischen Kurven haben den Vorteil, dass die Schlüssellängen um Faktor 6–30 kleiner sind, was bei vergleichbarem Sicherheitsniveau Ressourcen sparen kann, obwohl die Operationen auf elliptischen Kurven aufwendiger zu berechnen sind als Operationen in vergleichbar großen endlichen Körpern.

Heutzutage typische Schlüssellängen bei asymmetrischer Verschlüsselung sind 1024 Bit, 2048 Bit und 4096 Bit. Die Schlüssellängen sind nicht direkt mit den symmetrischen Verschlüsselungsverfahren vergleichbar. Tabelle 4.2 zeigt die Schlüssellängen der verschiedenen Verschlüsselungsverfahren im Vergleich zu ihren äquivalenten Vertretern der symmetrischen Verfahren. Die Daten entsprechen der empfohlenen ECRYPTII–Einschätzung⁷.

Tabelle 4.2.: Auf ECRYPTII–Einschätzung basierende effektive Schlüsselgrößen asymmetrischer und symmetrischer Verfahren im direkten Vergleich.

RSA modulus	ElGamal Gruppengröße	Elliptische Kurve	sym. Äquivalent
480	480	96	48
640	640	112	56
816	816	128	64
1248	1248	160	80
2432	2432	224	112
3248	3248	256	128
5312	5312	320	160
7936	7936	384	192
15424	15424	512	256

Für weitere Details zum Thema asymmetrische Kryptographie vgl. [21], S. 150 ff.

4.2.3. Hybride Verschlüsselungsverfahren

Asymmetrische Verschlüsselungsverfahren sind im Vergleich zu symmetrischen Verschlüsselungsverfahren sehr langsam, haben jedoch den Vorteil, dass kein *gemeinsamer* Schlüssel für die verschlüsselte Kommunikation bekannt sein muss. Symmetrische Verfahren hingegen sind sehr effizient, ein Hauptproblem, welches sie jedoch haben ist der Austausch eines *gemeinsamen* Schlüssels zum Ver- und Entschlüsseln.

Bei der hybriden Verschlüsselung macht man sich die Vorteile beider Systeme zu Nutzen. Bevor Alice und Bob kommunizieren können, tauschen sie mittels Public–Key–Kryptographie (asymmetrische Kryptographie) den *gemeinsamen* Schlüssel, welchen sie anschließend für die symmetrische

⁶Digital Signature Algorithm:

https://de.wikipedia.org/w/index.php?title=Digital_Signature_Algorithm&oldid=161702305

⁷ECRYPT II Yearly Report on Algorithms and Key Lengths (2012):

<http://www.ecrypt.eu.org/ecrypt2/documents/D.SPA.20.pdf>

Verschlüsselung verwenden, aus. Für weitere Details zum Thema hybride Verschlüsselungsverfahren vgl. [21], S. 179 ff.

4.3. Diffie–Hellman–Schlüsselaustausch

Aus dem Diffie–Hellman–Schlüsselaustausch (kurz DH) geht das ElGamal–Verschlüsselungsverfahren hervor. DH ist ein Schlüsselaustauschprotokoll, welches es zwei Kommunikationspartnern ermöglicht, einen *gemeinsamen* Schlüssel zu bestimmen, ohne diesen über den potentiell unsicheren Kommunikationskanal austauschen zu müssen.

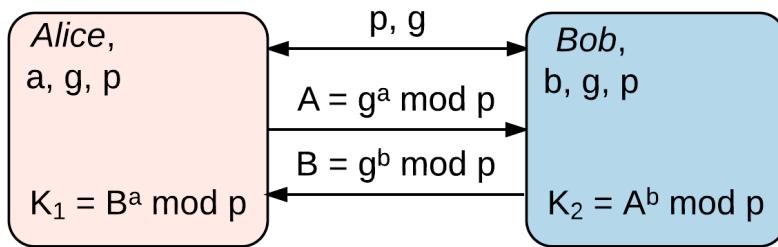


Abbildung 4.6.: Grafische Darstellung, Ablauf des Diffie–Hellman–Schlüsseltausch.

Abb. 4.6 zeigt Ablauf des Diffie–Hellman–Protokolls:

- 1) Alice und Bob einigen sich auf große Primzahl p und natürliche Zahl g , die kleiner ist als p .
- 2) Alice und Bob generieren jeweils eine geheime Zufallszahl a und b .
- 3) Alice berechnet $A = g^a \text{ mod } p$ und schickt A an Bob (dies entspricht im Grunde einem temporären ElGamal Schlüsselpaar: a = privater Schlüssel, g^a = öffentlicher Schlüssel)
- 4) Bob berechnet $B = g^b \text{ mod } p$ und schickt B an Alice.
- 5) Alice erhält B von Bob und berechnet mit a die Zahl $K_1 = B^a \text{ mod } p$.
- 6) Bob berechnet analog $K_2 = A^b \text{ mod } p$.

Beide haben den gleichen Schlüssel berechnet, da gilt:

$$K_1 = B^a = (g^b)^a = (g^a)^b = A^b = K_2$$

Für weitere Details zum Thema Diffie–Hellman vgl. [21], S. 311 ff.

4.4. Hashfunktionen

4.4.1. Kryptographische Hashfunktionen

Hashfunktionen werden in der Informatik verwendet, um eine beliebige endliche Eingabemenge auf einer Prüfsumme (Hashwert) einer bestimmten Länge abzubilden. Prüfsummen können verwendet werden, um beispielsweise die Integrität von Daten zu validieren. Ein Praxisbeispiel wäre die Korrektheit von übertragenen Daten zu validieren, beispielsweise nach dem Download eines Linux–Images.

Kryptographische Hashfunktionen sind spezielle Formen von Hashfunktionen, welche folgende Eigenschaften bieten:

- ▶ Einwegfunktion
- ▶ Schwache Kollisionsresistenz: Praktisch unmöglich zu gegebenen Wert x ein y zu finden, welches den gleichen Hashwert besitzt: $h(x) = h(y), x \neq y$
- ▶ Starke Kollisionsresistenz: Praktisch unmöglich, zwei verschiedene Eingabewerte x und y mit dem gleichen Hashwert zu finden $h(x) = h(y), x \neq y$ zu finden

4.4.2. Message Authentication Codes

Um nicht nur die Integrität der Daten, sondern auch deren Quelle zu validieren, werden sogenannte Message Authentication Codes (kurz MAC) verwendet. MACs sind schlüsselabhängige Hashfunktionen. Neben Hashfunktionen werden auch Blockchiffren verwendet. Abb. 4.7 zeigt die Übertragung von Daten mit einer Keyed-Hash Message Authentication Code (HMAC). Für weitere Details vgl. [21], S. 214 ff.

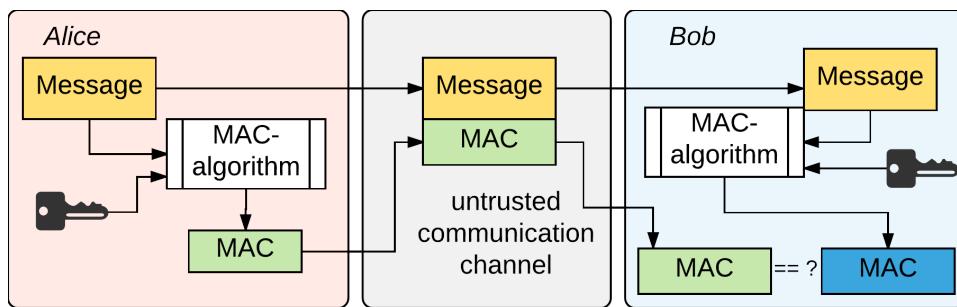


Abbildung 4.7.: Message-Übertragung mit HMAC.

4.5. Authentifizierungsverfahren

Bei den praktischen Authentifizierungsverfahren ist das Passwort immer noch eine sehr weit verbreitete Möglichkeit der Authentifizierung. Passwörter sind eine sehr problematische Möglichkeit der Authentifizierung, weil sie auf gute Entropie⁸ angewiesen sind. Das heißt, dass Passwörter möglichst zufällig sein müssen. Passwörter, die leicht zu erraten sind, sind *de facto* schlechte Authentifizierungsmechanismen.

Die Problematik mit den Passwörtern kennt heutzutage jedes Unternehmen. Sind die Passwort-Richtlinien zu kompliziert, werden die Passwörter oft von Benutzern aufgeschrieben. Gibt es keine Passwort-Richtlinien, dann verwenden Menschen oft schwache Passwörter, oft auch das gleiche einfache Passwort für mehrere Anwendungen.

Die Situation lässt sich jedoch auf recht einfache Art und Weise durch den Einsatz eines zusätzlichen Authentifizierungsfaktors verbessern. Diese Art der Authentifizierung wird Multi-Faktor- oder auch im Speziellen Zwei-Faktor-Authentifizierung genannt. Als zweiter Faktor kann beispielsweise ein biometrisches Merkmal verwendet werden. Eine weitere Form der Zwei-Faktor-Authentifizierung

⁸Entropie: [https://de.wikipedia.org/w/index.php?title=Entropie_\(Informationstheorie\)&oldid=161847818](https://de.wikipedia.org/w/index.php?title=Entropie_(Informationstheorie)&oldid=161847818)

wäre beispielsweise die Chipkarte der Bank. Hierbei wird einerseits die PIN (etwas das man weiß) und die Chipkarte (etwas das man hat) benötigt. Eine erfolgreiche Authentifizierung findet in dem Fall nur bei korrekter PIN unter Verwendung der Chipkarte der Bank statt.

4.6. Keymanagement

Das Keymanagement (Schlüsselverwaltung) ist einer der sensibelsten Bereiche bei der Implementierung eines Systems. Sind die Schlüssel unzureichend geschützt oder die Einsatzweise der Schlüssel fraglich, so kann ein System meist einfach kompromittiert werden. Neben sicherer Verwaltung der Schlüssel, ist auch die Beschränkung auf einen bestimmten Einsatzzweck essentiell.

Security is a process, not a product. — Bruce Schneier

5.1. Beurteilung von Sicherheit

Wie bereits in der Einleitung von Abschnitt 4.1 erwähnt, ist das Entwickeln von sicherer Software kein trivialer Prozess. Auch Systeme, die von Experten entwickelt wurden, können gravierende Sicherheitsmängel aufweisen.

Ein wichtiger Punkt, welcher von Experten oft geraten wird, ist die Verwendung bekannter und bewährter Algorithmen und Protokolle wie beispielsweise AES, RSA/DSA, TLS et cetera. Die Entwicklung neuer kryptographischer Algorithmen und Protokolle sollte nach Möglichkeit vermieden werden. Weiterhin werden Details von kryptographischen Elementen oftmals missverstanden oder es werden für den Einsatz von Sicherheit die falschen Techniken eingesetzt. Beispiele hierfür wären (vgl. [22]):

- ▶ Google Keyczar (timing side channel)¹
- ▶ SSL (session renegotiation)²
- ▶ Amazon AWS signature method (non-collision-free signing)³
- ▶ Flickr API signatures (hash length-extension)⁴
- ▶ Intel HyperThreading (architectural side channel)⁵

Erschwert kommt bei der Auswahl kryptographischer Algorithmen/Protokolle hinzu, dass sich Experten nicht immer einig sind oder es kommt erschwerend hinzu, dass es nicht für jeden Anwendungsfall eine konkrete Empfehlung geben kann. Manchmal muss auch zwischen Sicherheit und Geschwindigkeit abgewogen werden. Ein Beispiel hierfür wäre die Abwägung ob man Betriebsmodi verwendet, die Authentifikation und Verschlüsselung unterstützen, wie beispielsweise GCM, oder doch besser eine Encrypt-than-MAC-Komposition (vgl. auch [23]). Weiterhin macht die Kryptoanalyse Fortschritte und kommt beispielsweise zu neuen Erkenntnissen⁶, dass Primzahlen Hintertüren enthalten können.

Ein weiteres Beispiel welches die Komplexität der Lage darstellt, ist eine aktuelle Warnung vom BSI⁷ bei welcher Sicherheitssoftware aufgrund von gravierenden Sicherheitslücken als Einfallstor für Schadsoftware missbraucht werden kann.

Weiterhin macht es keinen Sinn und ist auch oft wirtschaftlich untragbar alle finanziellen Mittel in die Sicherheit eines Softwareproduktes zu investieren. Laut [21], S. 10 f. ist es in der Regel eine Abwägung zwischen möglichen Risikofaktoren und finanziellem sowie zeitlichem Aufwand. Es macht

¹Keyczar Vulnerability: <https://rdist.root.org/2009/05/28/timing-attack-in-google-keyczar-library/>

²SSL Vulnerability: <https://blog.ivanristic.com/2009/11/ssl-and-tls-authentication-gap-vulnerability-discovered.html>

³AWS Signature Vulnerability: <https://rdist.root.org/2009/05/20/amazon-web-services-signature-vulnerability/>

⁴Flickr Signature Vulnerability: http://netifera.com/research/flickr_api_signature_forgery.pdf

⁵Intel HyperThreading Vulnerability: <http://www.daemonology.net/hyperthreading-considered-harmful/>

⁶Cryptanalysis of 1024-bit trapdoored primes: <http://caramba.inria.fr/hsnfs1024.html>

⁷BSI Warn- und Informationsdienste: <https://www.cert-bund.de/advisoryshort/CB-K16-1797>

so gesehen keinen Sinn, mehrere Millionen Euro in ein Softwareprodukt zu investieren, welches beispielsweise nur Daten im Wert von ein paar tausend Euro schützt. Weiterhin ist es auch wichtig realistische Gefahren zu identifizieren und nicht unnötig Ressourcen in sicherheitstechnische Details zu investieren.

5.2. Angriffsfläche bei »brig«

5.2.1. Allgemein

Ziel dieser Arbeit ist es, praxisrelevante Gefahren/Risiken für ein Software-Produkt und einen Entwicklungsprozess, wie er bei »brig« vorliegt, zu definieren und mögliche Verbesserungskonzepte zu erarbeiten.

Hierbei wird in erster Linie angenommen, dass die meisten Sicherheitsprobleme durch Benutzerfehler oder durch mangelnde Kenntnis/Sicherheitsvorkehrungen zustande kommen. Primär spielen in der Arbeit praxisorientierte Ansätze eine wichtige Rolle, theoretische Sicherheitsmängel sind diesen untergeordnet und werden, auf Grund der hohen Komplexität der Thematik, nur am Rande behandelt.

Zu den theoretischen Problemen gehören im Umfeld dezentraler Softwaresysteme primär Angriffe wie beispielsweise die Sybil-Attacke, bei welcher es für einen Angreifer möglich ist, mit einer großen Anzahl von Pseudonymen einen überproportional großen Einfluss in einem dezentralen Netzwerk zu erlangen. Möglichkeiten, die diesen Angriff entgegenwirken sind in erster Linie zentrale Authentifizierungsinstanzen. Der Angriff wird unter [3], S. 200 ff. genauer erläutert. Das von »brig« verwendete IPFS-Netzwerk verwendet Teile des S/Kademlia-Protokolls, welches es prinzipiell erschwert eine Sybil-Attacke⁸ durchzuführen. Unter [24] wird S/Kademlia bezüglich der Angriffe auf dezentrale Architekturen genauer untersucht.

5.2.2. Praxisorientierte Herausforderungen an die Software

Zu den praxisorientierten Problemen bei der Entwicklung einer Software wie »brig« gehören in erster Linie folgende Punkte, auf welche primär in der Arbeit eingegangen wird:

Passwortmanagement: Menschen sind schlecht darin, gute Passwörter zu vergeben. Erweiterte Technologien ermöglichen es, immer schneller gestohlene Passwörter-Datenbanken zu knacken und in Systeme einzubrechen. *Bruce Schneier* und *Dan Goodin* beschreiben in ihren Artikeln »A Really Good Article on How Easy it Is to Crack Passwords«⁹ und »Why passwords have never been weaker – and crackers have never been stronger«¹⁰ die Problematik detaillierter und geben Empfehlungen. Für weitere Details siehe auch Abschnitt 7.6.

Schlüsselmanagement: Die sichere Kommunikation von kryptographischen Schlüsseln stellt eines der größten Probleme im digitalen Zeitalter dar (21, S. 326 ff.). In jüngster Vergangenheit wurde bei-

⁸IPFS – Content Addressed, Versioned, P2P File System:

<https://blog.acolyer.org/2015/10/05/ipfs-content-addressed-versioned-p2p-file-system/>

⁹A Really Good Article on How Easy it Is to Crack Passwords:

https://www.schneier.com/blog/archives/2013/06/a_really_good_a.html

¹⁰Why passwords have never been weaker – and crackers have never been stronger:

<http://arstechnica.com/security/2012/08/passwords-under-assault/>

spielsweise beim FreeBSD-Projekt mittels gestohlener kryptographischer Schlüssel eingebrochen¹¹. Laut Berichten¹²¹³¹⁴ expandiert der Malware-Markt in dieser Richtung, es wird zunehmend Malware für das Ausspähen kryptographischer Schlüssel entwickelt.

Authentifizierung: Wie weiß der Benutzer, dass sein Kommunikationspartner der ist für den er sich ausgibt? Hier sollen mögliche Konzepte erarbeitet werden, um Angriffe durch fremde Kommunikationspartner identifizieren zu können.

Softwareverteilung: Der Benutzer muss sicherstellen können, dass die aus dem Internet bezogene Open-Source-Software keine Malware ist. Weiterhin ist es wichtig, Software auf einem aktuellen Stand zu halten, um bekannte Sicherheitslücken zu schließen.

Entwicklungsumgebung bei Open-Source-Projekten: Wie können Softwareentwickler das Risiko von unerwünschten Manipulationen am eigenen Projekt minimieren? Wie sichern Softwareentwickler ihren Arbeitsprozess ab?

¹¹ Hackers break into FreeBSD with stolen SSH key: http://www.theregister.co.uk/2012/11/20/freebsd_breach/

¹² Malware & Hackers Collect SSH Keys to Spread Attacks: <https://www.ssh.com/malware/>

¹³ Are Your Private Keys and Digital Certificates a Risk to You?:

<https://www.venafi.com/blog/are-your-private-keys-and-digital-certificates-a-risk-to-you>

¹⁴ Active attacks using stolen SSH keys:

<https://isc.sans.edu/forums/diary/Active+attacks+using+stolen+SSH+keys+UPDATED/4937/>

IPFS (InterPlanetary File System) stellt die Netzwerkbasis für »brig« dar. Da IPFS teilweise andere Ziele als »brig« hat, ist es wichtig, dass die Anforderungen von »brig« durch die IPFS-Basis nicht verletzt werden. Im Folgenden wird IPFS bezüglich bestimmter sicherheitstechnischer Anforderungen genauer beleuchtet, um Diskrepanzen zu den Zielen von »brig« zu identifizieren.

Die IPFS-Codebasis umfasst aktuell ≈ 900.000 LoC (Lines of Code, siehe [Anhang A](#)). Davon gehören ≈ 100.000 LoC direkt dem IPFS-Projekt an, ≈ 800.000 LoC stammen aus Drittanbieter-Bibliotheken.

Im zeitlich begrenzten Umfang der Masterarbeit können nur selektive Mechanismen der Software untersucht werden. Eine genaue Analyse der Quelltext-Basis ist aufgrund der Projektgröße und der begrenzten Zeit nicht möglich.

Es wurde folgende Version aus dem Arch Linux-Repository evaluiert:

```
$ ipfs version
ipfs version 0.4.3
```

6.1. Einleitung IPFS

Das InterPlanetary File System wird als »content-addressable, peer-to-peer hypermedia distribution protocol« definiert. Das Besondere an IPFS ist, dass es ein sogenanntes Content-Addressable-Network (CAN)¹ darstellt. Ein CAN arbeitet mit einer verteilten Hashtabelle (Distributed Hash Table, kurz DHT²), welche als grundlegende Datenstruktur verwendet wird, um die Daten innerhalb eines Peer-to-Peer-Netzwerks zu lokalisieren und zu speichern.

Eine DHT als Datenstruktur bringt in der Theorie laut Wikipedia folgende Eigenschaften mit sich:

- ▶ **Fehlertoleranz:** Das System sollte zuverlässig funktionieren, auch wenn Knoten ausfallen oder das System verlassen.
- ▶ **Lastenverteilung:** Schlüssel werden gleichmäßig auf alle Knoten verteilt.
- ▶ **Robustheit:** Das System sollte korrekt funktionieren können, auch wenn ein Teil (möglicherweise ein Großteil) der Knoten versucht, das System zu stören.
- ▶ **Selbstorganisation:** Es ist keine manuelle Konfiguration nötig.
- ▶ **Skalierbarkeit:** Das System sollte in der Lage sein, auch mit einer großen Anzahl von Knoten funktionsfähig zu bleiben.

¹Content Addressable Network:

https://de.wikipedia.org/w/index.php?title=Content_Addressable_Network&oldid=157598552

²Verteilte Hashtabelle: https://de.wikipedia.org/w/index.php?title=Verteilte_Hashtabelle&oldid=157901191

6.2. IPFS-Basis

Das IPFS-Dateisystem beziehungsweise Protokoll bringt das Kommandozeilenwerkzeug ipfs mit, weiterhin kann es jedoch auch als Software-Bibliothek verwendet werden. Dieses ermöglicht eine rudimentäre Nutzung von IPFS. Beim Initialisieren von IPFS wird ein RSA-Schlüsselpaar generiert. Ein IPFS-Repository kann mit dem Befehl ipfs init initialisiert werden. Dabei wird standardmäßig unter `~/ipfs` ein Repository angelegt.

```
$ ipfs init
initializing ipfs node at /home/qitta/.ipfs
generating 2048-bit RSA keypair...done
peer identity: QmbEg4fJd3oaM9PrpcMHcn6QR2HMdhRXz5YyL5fHnqNAET
to get started, enter:

ipfs cat /ipfs/QmYwAPJzv5CZsnA625s3Xf2nemtYgPpHdWEz79o jWnPbdG/readme
```

Bei der Initialisierung wird eine Peer-ID erzeugt. Anschließend kann der Benutzer die readme-Datei aus dem IPFS-Store betrachten. [Anhang B](#) zeigt weiterhin die aktuelle Sicherheitswarnung der IPFS-Software. Es wird explizit darauf hingewiesen, dass sich IPFS im Alphastadium befindet. Weiterhin gibt es in der Datei »security-notes« Details zur Sicherheit, welche analog zur readme-Datei betrachtet werden kann.

6.3. IPFS-Backend

6.3.1. Speicherung und Datenintegrität

Die Speicherung von Daten mag auf den ersten Blick simpel erscheinen. Betrachtet man jedoch die Rahmenbedingungen, die zu beachten sind, um Daten sicher zu speichern, wird die Thematik komplizierter. Das Hauptproblem an dieser Stelle ist die sogenannte Silent Data Corruption (vgl. [25], [26]), oft auch Bitrot genannt. Der Begriff beschreibt den Umstand, dass Fehler in Daten im Laufe der Zeit auftreten. Für die Fehlerursache können verschiedene Gründe verantwortlich sein, wie beispielsweise:

- ▶ Hardwarefehler bedingt durch Alterungsprozess der Festplatte
- ▶ Fehler in der Festplatten-Firmware
- ▶ Fehler in der Controller-Firmware
- ▶ Fehler in der Software (Kernel, Dateisystem)
- ▶ Schadsoftware

Gängige Dateisysteme wie beispielsweise NTFS³ oder EXT4⁴ können Fehler, verursacht durch Silent Data Corruption, nicht erkennen und den Benutzer vor dieser Fehlerart nicht schützen. Um eine Veränderung der Daten festzustellen, müsste der Benutzer beispielsweise die Daten mit einer kryptographischen Prüfsumme validieren. Entspricht die Prüfsumme beim Lesen der Daten, der gleichen Prüfsumme, welche bei der Speicherung der Daten ermittelt wurde, so sind die Daten mit hoher

³NTFS Dateisystem: <https://en.wikipedia.org/w/index.php?title=NTFS&oldid=743913107>

⁴EXT4 Dateisystem: <https://en.wikipedia.org/w/index.php?title=Ext4&oldid=738311553>

Wahrscheinlichkeit korrekt an den Benutzer zurückgegeben werden. Diese Art der Validierung der Integrität ist jedoch aufgrund des hohen Aufwands nicht praxistauglich.

Dateisysteme wie BTRFS⁵ oder ZFS⁶ validieren die Daten und Metadaten während der Lese- und Schreibvorgänge mittels Prüfsummen. Durch dieses spezielle Feature kann die Verarbeitungskette beim Lesen- und Speichern der Daten bezüglich ihrer Integrität validiert werden. Bei der Benutzung eines RAID-System⁷ können die Daten sogar automatisiert ohne Zutun des Benutzers korrigiert⁸ werden.

Das Speichern der Daten erfolgt bei IPFS (blockweise, in sogenannten chunks) mittels eines Merkle-DAG (directed acyclic graph, gerichteter azyklischer Graph).

IPFS verwendet als Prüfsummen-Format ein eigens entwickeltes Multihash-Format⁹. Abb. 6.1 zeigt das Multihash-Format. Es stellt eine selbstbeschreibende Prüfsumme, welche den Algorithmus, die Länge und die eigentliche Prüfsumme kombiniert. Dieser wird in verschiedenen Varianten encodiert. Beispielsweise Base32 für die interne Namensvergabe der Datenblöcke oder Base58 für die Repräsentation der Peer-ID.

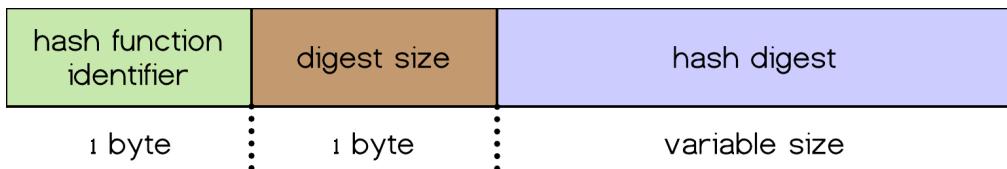


Abbildung 6.1.: Das Multihash-Format.

Das folgende Listing zeigt den internen Aufbau eines IPFS-Repository. Die Daten sind hierbei in .data-Blöcke aufgeteilt und gespeichert. Die Benennung der Datenblöcke basiert auf dem Multihash, die Enkodierung bei Datenblöcken ist Base32.

```
$ tree .ipfs
.ipfs
|--- blocks
|   |--- CIQBE
|   |   |--- CIQBED3K6YA5I3QQWLJOCHWXDRK5EXZQILBCAPEDUJENZ5B5HJ5R3A.data
|   |--- CIQCL
|   |   |--- CIQCLECESM3B720M5DWMSFO6C2EA6KNCI04SFVFDH06JVBYRSJ5G3HQ.data
...
|   |--- CIQPP
|       |--- CIQPPQVFU2X6L6RB67SNEYN4MPR236SNPL50ML2TBA4RIQQPM4FY6VY.data
|--- config
```

⁵Btrfs Dateisystem: <https://en.wikipedia.org/w/index.php?title=Btrfs&oldid=761911810>

⁶ZFS Dateisystem: <https://en.wikipedia.org/w/index.php?title=ZFS&oldid=761908184>

⁷RAID Wikipedia: <https://en.wikipedia.org/w/index.php?title=RAID&oldid=761073220>

⁸FreeBSD/ZFS - self-healing example: <https://www.youtube.com/watch?v=VlFGTtU65Xo>

⁹Github-Multihash: <https://github.com/multiformats/multihash>

```

|--- datastore
|   |--- 000002.ldb
...
|   |--- 000009.log
|   |--- CURRENT
|   |--- LOCK
|   |--- LOG
|   |--- MANIFEST-000010
|--- version

```

17 directories, 25 files

Die Speicherung der Daten in einem Merkle-DAG hat den Vorteil, dass die Daten bei der Speicherung üblicherweise mit einer kryptographischen Prüfsumme abgelegt werden. Durch diesen Umstand kann IPFS eine unerwünschte Veränderung an den Daten feststellen. Das folgende Beispiel zeigt die unerwünschte Modifikation der readme-Datei direkt im Store-Backend und wie die Integritätsprüfung von IPFS die Änderung der Daten erkennt:

```

# Validierung der Integrität der Daten
$ ipfs repo verify
verify complete, all blocks validated.

# Unerwünschte Modifikation der Daten.
$ echo "Trüffelkauz" >> .ipfs(blocks/CIQBED3K6YA[...]JENZ5B5HJ5R3A.data

# Erneute Validierung der Integrität der Daten
$ ipfs repo verify
block QmPZ9gcCEpqKTo6aq61g2nXGUhM4iCL3ewB6LDXZCtioEB \
was corrupt (block in storage has different hash than requested)
Error: verify complete, some blocks were corrupt.

```

Aktuell werden die Daten mittels SHA256 gehasht. Eine Verschlüsselung der Daten findet nicht statt. Es gibt zwar Pläne für die Zukunft einen verschlüsselten Store zu realisieren, aktuell wird jedoch in Feature-Requests¹⁰ die Möglichkeit der manuellen Verschlüsselung beispielsweise mittels OpenSSL/GPG nahegelegt.

6.3.2. Datendeduplizierung

Neben der Möglichkeit der Datenvälidierung, hat die Speicherung der Daten in einem Merkle-DAG auch den Vorteil, Daten effizient deduplizieren zu können. Abb. 6.2 zeigt die Deduplizierung auf Blockebene. Eine vier MiByte große Textdatei wurde hierbei jeweils an verschiedenen Stellen geändert und unter einem neuen Namen gespeichert. Ohne Deduplizierung wird die Datei jedes mal jeweils komplett gespeichert. Der Speicherplatz der dafür benötigt wird würde normalerweise bei 16 MiByte

¹⁰IPFS file encryption request: <https://github.com/ipfs/faq/issues/116>

liegen, da die Daten auf gewöhnlichen Dateisystemen wie beispielsweise EXT4 oder NTFS redundant abgespeichert werden. IPFS teilt diese Datei – in diesem Fall nur beispielhaft – in 1 MiByte große Blöcke auf und speichert nur Datenblöcke, welche dem IPFS-Backend noch nicht bekannt sind (vgl. auch [27], S. 27 f.). Alle bekannten Blöcke werden vom Merkle-DAG nur referenziert. Durch diesen Ansatz reduziert sich im Beispiel der benötigte Speicherplatz auf 6 MiByte.

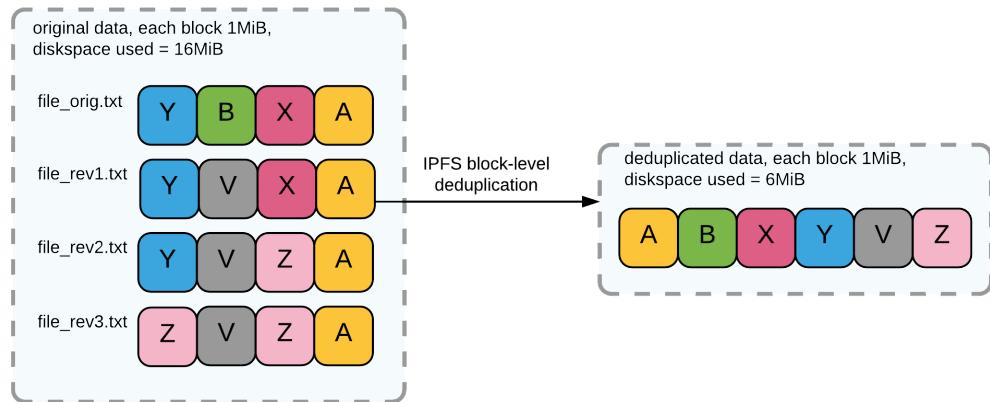


Abbildung 6.2.: IPFS Block-Level Deduplizierung von Daten. Eine 4 MiB große Textdatei wurde viermal kopiert und jeweils an verschiedenen Stellen geändert.

6.3.3. IPFS-Daten und IPFS-Blöcke

Das IPFS-Kommandozeilentool kann mittels des `ipfs add`-Befehl Daten dem IPFS-Netzwerk hinzufügen. Wie bereits erwähnt werden die Daten blockweise abgelegt. Weiterhin existiert auch die Möglichkeit, IPFS auf unterster Ebene Blöcke direkt hinzuzufügen. Hierzu wird das Subkommando `ipfs block` verwendet. Für weitere Details siehe IPFS-Dokumentation¹¹.

6.4. IPFS-ID

6.4.1. Aufbau

Das generierte Schlüsselpaar wird im Klartext auf der Festplatte abgelegt. Der öffentliche Schlüssel kann mit `ipfs id` angeschaut werden, dies liefert folgende Ausgabe (gekürzt):

```
$ ipfs id
{
  "ID": "QmbEg4fJd3oaM9PrpcMHcn6QR2HMdhRXz5YyL5fHnqNAET",
  "PublicKey":
    "CAASpgIwggEiMA0GCSqGSIb3DQEBAQUAA4IBDwAwggEKAoIBAQDGDgKtgJ9FW/EL1qY00hz
    nGGh7dPAszDDpC3fhHcF7rI9iyLp5ei0T9gjfvsj+ULhxKqXHU9qoD7LjiUPUHQKPnND1Yv
    [...]
    7X5gVmtkEB8eDjLkcMNCmqMarAjjSb9Sg5uYI0j1WHPPCrvTVQioZIIHJ6Z7ogWIDpDR5T
    0g55tS4kW7qSJUQebh54v1gyEVd2mSgUf40MBAgMBAAE=",
```

¹¹IPFS-Dokumentation: <https://ipfs.io/docs/commands/>

```

    "Addresses": null,
    "AgentVersion": "go-libp2p/3.3.4",
    "ProtocolVersion": "ipfs/0.1.0"
}

```

Die unter ID gelistete Nummer stellt die Prüfsumme über den öffentlichen Schlüssel als Multihash in Base58-Enkodierung dar. Mit dieser ID lässt sich ein Benutzer beziehungsweise Peer im IPFS-Netzwerk eindeutig identifizieren.

Der private Schlüssel ist neben weiteren Informationen zum ipfs-Repository in der `~/ipfs/config`-Datei zu finden, welche beim Anlegen des Repositories automatisch erstellt wird (gekürzte Ausgabe):

```
$ cat .ipfs/config | grep PrivKey
"PrivKey":
"CAASpwkwggSjAgEAAoIBAQDGDgKtgJ9FW/EL1qY0OhznGGh7dPAszDDpC3fhHcF7rI9iyLp
5ei0T9gjfvsj+ULhxKqXHU9qoD7LjiUPUHQKPnND1YvWzpBIZNUhaiuo107J5MztPvroQ8/
[...]
cWTidFNBdz5IGzj0P0o050K6gadI608TqTvxcLWF4iC/hOMvTUA7W9r1l9dea+YXubchvY
VQMS8YcXyzXoE+DQXzM5TqbZT/jxUS/UUFcs7UKhuEu+E9etcYBgnrcrMoQckQE="
```

Für weitere Details zur Erstellung der Identität sollte der Quelltext¹² zu Rate gezogen werden.

6.4.2. Authentifizierung

Ein Authentifizierungsmechanismus im eigentlichen Sinne existiert bei IPFS nicht. Die Benutzer haben lediglich eine eindeutige globale Peer-ID. Dateien werden nicht direkt von einer bestimmten Peer-ID, sondern aus dem IPFS-Netzwerk bezogen. Schaut man sich beispielsweise mit `ipfs dht findprovs` an, welche Peers die `readme`-Datei anbieten, bekommt man eine Liste verschiedener Teilnehmer:

```
$ ipfs dht findprovs QmYwAPJzv5CZsnA625s3Xf2nemtYgPpHdWEz79ojWnPbdG
Qmadh19kRmg1SufioE8ipNvyESd3fwKqqiMdRdFHBjxseH
[...]
QmPKBHmwWJbin2BuzE3zvua9JxsrsEGm6rn69CgWTEU5wn
```

Ruft man den Befehl auf der Prüfsumme einer persönlichen Datei im lokalen Netzwerk auf, so bekommt man als einzigen anbietenden Knoten die Peer-ID des Rechners im lokalen Netzwerk:

```
$ ipfs dht findprovs QmagF4CPz5LCwSwkwhYwn1uZHhJXoXQJDWeV65fcuTyqP1
QmW2jc7k5Ug987QEkuX6tJUTdZov7io39MDCiKKp2f57mD
```

Eine Art Authentifizierung kann also nur manuell über einen Seitenkanal erfolgen. Ein Benutzer kann also nur überprüfen ob eine Datei mit einer bestimmten Prüfsumme auch auf einem bestimmten System mit der ihm bekannten Peer-ID vorzufinden ist. Hier wäre es denkbar, dass zwei kommunizierende Parteien ihre Peer-ID gegenseitig telefonisch austauschen beziehungsweise bestätigen. Diese

¹²IPFS Schlüsselgenerierung:

<https://github.com/ipfs/go-ipfs/blob/e2ba43c12dd7076357d5627ef02ed56bf2a55c30/repo/config/init.go#L95>

grundlegende Funktionalität der Authentifizierung einer Quelle ist nur bedingt praxistauglich, da eine nicht persönliche Datei durchaus mehrere Provider haben kann.

Einschub: Im Gegensatz dazu haben beispielsweise andere dezentrale Systeme mit einem direkten Kommunikationskanal weitere Möglichkeiten der Authentifizierung. Der Instant-Messaging-Client Pidgin¹³ bietet beispielsweise mit dem OTR-Plugin¹⁴ folgende Möglichkeiten für die Authentifizierung einer gesicherten Verbindung:

- ▶ *Frage und Antwort-Authentifizierung:* Alice stellt Bob eine Frage zu einem gemeinsamen Geheimnis. Beantwortet Bob diese Frage korrekt, so wird er vom System gegenüber Alice authentifiziert – das heißt, der Fingerabdruck (Prüfsumme über eine ID, die Bob eindeutig kennzeichnet) wird in Kombination mit dem Benutzernamen von Bob vom System als valide klassifiziert und abgespeichert.
- ▶ *Shared-Secret-Authentifizierung:* Alice weist Bob an, das gemeinsam bekannte Geheimnis in einem entsprechenden Programmdialogfenster einzutragen. Alice trägt das gemeinsame Geheimnis ebenso in einem Programmdialogfenster ein. Bei Übereinstimmung des gemeinsamen Geheimnisses wird Bob gegenüber Alice vom System authentifiziert – analog zur Frage und Antwort-Authentifizierung wird der Fingerabdruck vom System als valide klassifiziert und abgespeichert.
- ▶ *Manuelle Verifizierung vom Fingerabdruck:* Alice verifiziert den ihr vom System angezeigten Fingerabdruck (Prüfsumme über eine eindeutige ID) von Bob. Dazu kann sie beispielsweise Bob anweisen, ihr über einen Seitenkanal (beispielsweise Telefon) die Korrektheit des Fingerabdruck zu bestätigen.

Die genannten Verfahren erlauben eine initiale Authentifizierung zwischen den Kommunikationspartnern. Bei zukünftiger Kommunikation wird jeweils die ID der Benutzer mit der bei der initialen Authentifizierung gespeicherten ID verglichen.

6.5. IPFS-Netzwerk

Das IPFS-Netzwerk arbeitet mit einer DHT. Standardmäßig sind nach der Installation eine Reihe von sogenannten Bootstrap-Nodes eingetragen, welche einen initialen »Einstiegsplatz« bieten (gekürzt):

```
$ ipfs bootstrap list
/ip4/104.131.131.82/tcp/4001/ipfs/QmaCpDMGvV2BGHeYERUEnRQAwe3N8SzbUtfsmvsqQLuvuJ
[...]
/ip4/178.62.61.185/tcp/4001/ipfs/QmSoLMeWqB7YGVLJN3pNLQpmmEk35v6wYtsMGLzSr5QBU3
```

Alle Änderungen wie beispielsweise das Hinzufügen von Daten, werden so in dem durch alle Nodes aufgespannten Netzwerk allen Teilnehmern bekannt gemacht. Da alle Teilnehmer im IPFS-Netzwerk gleichberechtigt sind, wird lediglich die Prüfsumme einer Datei benötigt, um an die Daten zu gelangen.

Aktuell verwendet IPFS als Transport-Verschlüsselung ein selbst entwickeltes Protokoll namens

¹³Instant-Messaging-Client Pidgin:

[https://de.wikipedia.org/w/index.php?title=Pidgin_\(Instant_Messenger\)&oldid=155942615](https://de.wikipedia.org/w/index.php?title=Pidgin_(Instant_Messenger)&oldid=155942615)

¹⁴Off-the-Record: https://en.wikipedia.org/w/index.php?title=Off-the-Record_Messaging&oldid=741588882

Secio¹⁵, welches laut Entwickleraussagen ([Anhang C](#)) auf einem TLS1.2-Modi basiert. Es ist geplant in Zukunft auf TLS1.3 zu migrieren.

6.6. Zusammenfassung IPFS-Evaluation

Aus Datenhaltungs- und Netzwerksicht stellt IPFS zum aktuellen Zeitpunkt eine attraktive Basis für die Entwicklung des »brig«-Prototypen dar. Dabei sind insbesondere folgende Features interessant:

- ▶ Funktionsweise auf Basis eines Content Addressable Network
- ▶ Netzwerkarchitektur basierend auf Public-Key-Infrastruktur
- ▶ Möglichkeit der Deduplizierung von Daten (aufgrund der Speicherung in einem Merkle-DAG)
- ▶ Möglichkeit der Validierung der Integrität der Daten (SHA256-basiert)

Aus Sicht der Sicherheit muss IPFS um folgende Funktionalitäten erweitert werden:

- ▶ Verschlüsselte Lagerung der Daten und kryptographischen Schlüssel
- ▶ Mechanismus zur Authentifizierung von Kommunikationspartnern
- ▶ Standardisiertes Protokoll zum verschlüsselten Austausch von Metadaten
- ▶ Schlüsselmanagement (Verwaltung und Sicherung von Schlüsseln, Identitäten)

¹⁵GitHub-IPFS-Secio-Verschlüsselung: <https://github.com/libp2p/go-libp2p-secio>

Evaluation von »brig«

Parallel zu dieser Arbeit wird der »brig«-Prototyp entwickelt. Das Ziel dieses Kapitels ist es, die bisherige Arbeit aus Sicht der Sicherheit erneut zu evaluieren und bisher gemachte Fehler zu identifizieren. Für weitere allgemeine Details zur Architektur von »brig«, siehe die Arbeit von Herrn Pahl [27]. Für die Evaluation wird die Softwareversion brig v0.1.0 verwendet:

```
$ brig --version
brig version v0.1.0-alpha+0d4b404 [buildtime: 2016-10-10T10:05:10+0000]
```

7.1. Einleitung »brig«

Das Ziel ist es, mit »brig« ein dezentrales Dateisynchronisationswerkzeug zu entwickeln, welches eine gute Balance zwischen Sicherheit und Usability bietet. Die Entwicklung eines gut funktionierenden dezentralen Protokolls/Dateisystems ist nicht trivial.

In Abschnitt 2.1.6.1 wurden bereits verschiedene dezentrale Protokolle genannt. Diese sind jedoch hauptsächlich für den generellen Dateiaustausch ausgelegt. Um die in Kapitel 3 aufgeführten Anforderungen zu realisieren, müssen die genannten Protokolle beziehungsweise das Verhalten des Peer-to-Peer-Netzwerks an die gesetzten Anforderungen angepasst werden. Als Basis für die Implementierung eines Prototypen standen die beiden Protokolle BitTorrent und IPFS in der engeren Auswahl. Aufgrund der unter Abschnitt 6.6 genannten Funktionalitäten wurde IPFS als Basis bevorzugt.

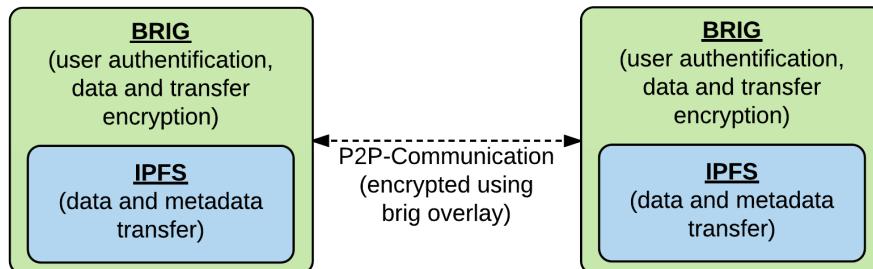


Abbildung 7.1.: »brig« als Overlay-Netzwerk für IPFS

Abb. 7.1 zeigt die Funktionsweise von »brig« als sogenanntes Overlay-Netzwerk. »brig« wird verwendet, um die in Abschnitt 6.6 fehlenden Eigenschaften des IPFS-Protokolls zu ergänzen.

7.2. Datenverschlüsselung

Standardmäßig werden die Daten bei IPFS unverschlüsselt gespeichert. Weiterhin basiert die aktuelle Transportverschlüsselung der Daten auf einem nicht standardisierten Protokoll.

7.2.1. Datenverschlüsselungsschicht

Um die gesetzten Anforderungen (Vertraulichkeit von Daten, [Abschnitt 3.3](#)) zu erreichen, muss »brig« die Funktionalität von IPFS so erweitern, dass die Authentizität und Vertraulichkeit der Daten bei lokaler Speicherung aber auch bei der Übertragung gewährleistet ist. Für diesen Einsatzzweck wurde eine Verschlüsselungsschicht vor dem IPFS-Backend [Abb. 7.2](#) eingeführt.

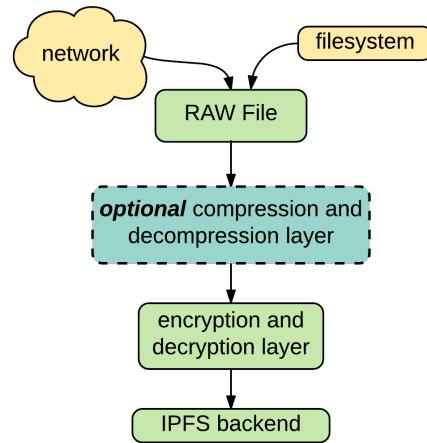


Abbildung 7.2.: »brig« Verschlüsselungsschicht mit Datenverschlüsselung mit Authentizität.

Die Verschlüsselungsschicht ist so ausgelegt, dass die verwendete Verschlüsselungstechnik austauschbar ist. Für den anfänglichen Prototypen wurden zwei Verfahren (AES, ChaCha20) implementiert. Die Entscheidung bei der Standardeinstellung ist neben dem weit etablierten AES-Standard auf ChaCha20 gefallen. Dieses recht neue Verfahren bietet Geschwindigkeitsvorteile¹² auf schwächerer und mobiler Hardware, insbesondere bei Hardware ohne kryptographischer Beschleunigung.

[Abb. 7.3](#) zeigt das Container-Format, welches für »brig« entwickelt wurde, um diese Anforderungen zu erreichen.

¹Do the ChaCha: better mobile performance with cryptography:

<https://blog.cloudflare.com/do-the-chacha-better-mobile-performance-with-cryptography/>

²AES-NI SSL Performance: https://calomel.org/aesni_ssl_performance.html

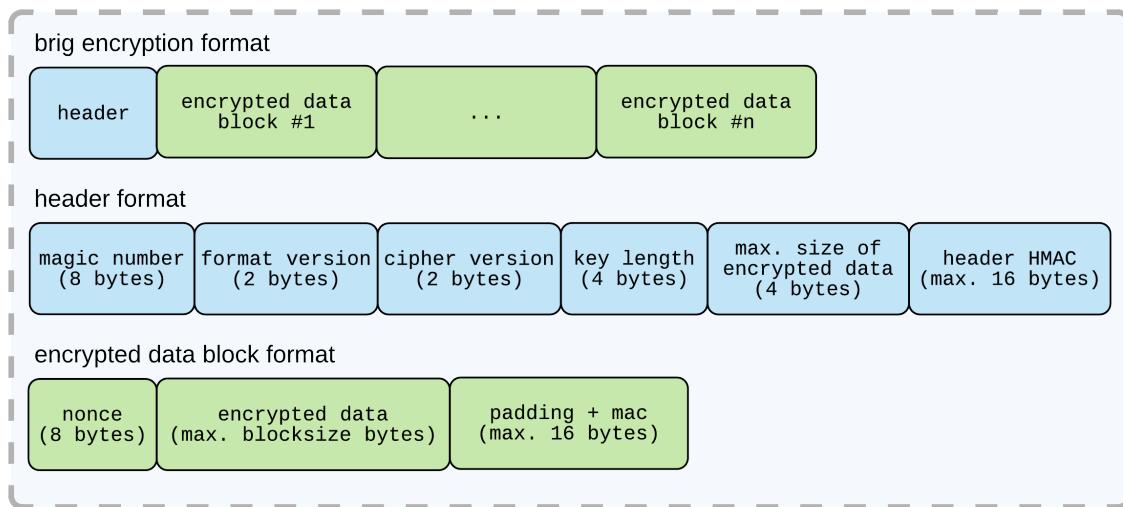


Abbildung 7.3.: »brig«-Container-Format für Datenverschlüsselung mit Authentizität.

Das Container-Format wurde so angelegt, um wahlfreien Zugriff auf Daten zu ermöglichen und den Verschlüsselungsalgorithmus austauschbar zu machen. Falls Schwächen bei einem bestimmten Algorithmus auftauchen sollten, kann die Vertraulichkeit der Daten durch den Wechsel auf einen weiterhin sicheren Algorithmus gewährleistet werden.

7.2.2. Verwendete Algorithmen

Die aktuelle Softwareversion³ beherrscht die AEAD-Betriebsmodi⁴:

- ▶ AES-GCM [28] mit 256 Bit Schlüssellänge
- ▶ ChaCha20/Poly1305 mit 256 Bit Schlüssellänge (externe Bibliothek⁵)

Der AEAD-Betriebsmodus wurde gewählt, weil er den Vorteil hat, dass er neben der Vertraulichkeit auch die Authentizität und die Integrität sicherstellt.

7.2.3. Geschwindigkeitsevaluation

Die bisherigen »brig«-Benchmarks unter [27], S. 96 ff. haben die Geschwindigkeit von »brig« mit IPFS verglichen. Hierbei ist auffällig gewesen, dass die Geschwindigkeit bei Verschlüsselung vergleichsweise stark eingebrochen ist.

Um Verschlüsselungsoperationen zu beschleunigen, gibt es neben der Wahl verschiedener Blockchiffren, auch die Möglichkeit einer hardwarebasierten Beschleunigung. Moderne Prozessoren bieten eine Beschleunigung mit dem sogenannten AES-NI-Befehlserweiterungssatz. Diese müssen jedoch vom Betriebssystem und der jeweiligen Anwendung unterstützt werden.

³ Aktuell von »brig« unterstützte symmetrische Verschlüsselungsverfahren:

<https://github.com/disorganizer/brig/blob/fa9bb634b4b83aaabaa967ac523123ce67aa217d/store/encrypt/format.go>

⁴ Authenticated encryption: https://en.wikipedia.org/w/index.php?title=Authenticated_encryption&oldid=760384391

⁵ ChaCha20/Poly1305-Bibliothek: <https://github.com/codahale/chacha20poly1305>

Von CloudFlare gibt es einen Go-Fork⁶, welcher AES-NI-Erweiterungen unterstützt und somit eine erhöhte Performance auf bestimmten Systemen bieten sollte.

Der AES-NI-Befehlserweiterungssatz war lange Zeit aufgrund von Lizenzproblemen nicht in Go implementiert. Nach eingehender Recherche scheinen jedoch bereits die Patches von CloudFlare mittlerweile in Go Einzug⁷⁸ gefunden zu haben.

7.2.4. Testumgebung

Im Folgenden soll die Verschlüsselungsschicht separiert betrachtet werden, um genauere Aussagen über die Ressourcennutzung machen zu können. Weiterhin soll untersucht werden, wie sich die Verschlüsselungsschicht optimieren lässt beziehungsweise ob die getroffenen Parameter bezüglich der Blockgröße weiterer Optimierung bedürfen.

Weiterhin sollen verschiedene Architekturen in den Benchmark einbezogen werden, damit die Nutzung der Algorithmen und Ressourcen besser klassifiziert werden kann.

Um das Verhalten auf verschiedenen Klassen von Rechnern testen zu können, wurden zwei »Systemklassen« in die Geschwindigkeitsanalyse mit einbezogen. Tabelle 7.1 zeigt die getesteten Prozessorarchitekturen, die schwächeren Systeme sollen hierbei ersatzweise für mobile Plattformen als Referenzwert dienen. Für genauere Informationen zur jeweiligen CPU siehe Anhang D. Weiterhin wurde mit dem dd-Kommandozeilen-Werkzeug ein Lese- und Schreibtest durchgeführt, um die Geschwindigkeit der Festplatte (in der Regel langsamste Komponente) zu identifizieren. Der dd-Benchmark dient lediglich als grober Orientierungswert für die jeweiligen Systeme.

Tabelle 7.1.: Evaluierte Testsysteme mit und ohne AES-NI-Befehlserweiterungssatz.

	Intel i5 3320M	AMD Phenom X4	Intel Atom N270	Raspberry Pi Zero
Architektur	x86_64	x86_64	x86	ARM
Betriebsmodus	64-bit	64-bit	32-bit	32-bit
Arbeitsspeicher	16 GB	8 GB	1.5 GB	512 MB
Taktfrequenz	3.30 GHz (max)	3.2 GHz	1.6 GHz	1 GHz
AES-NI	ja	nein	nein	nein
Kernel	4.8.12	4.8.12	4.4.25	4.4.30
dd-Lesen	470.7 MB/s	409.2 MB/s	55.6 MB/s	21.8 MB/s
dd-Schreiben	181.7 MB/s	146.4 MB/s	10.7 MB/s	10.1 MB/s

Der Benchmark soll die maximal mögliche Geschwindigkeit des jeweiligen Systems beim Ver- und Entschlüsseln evaluieren. Daher wird der Benchmark vollständig in einer RAM-Disk durchgeführt. Abb. 7.4 zeigt grafisch den Aufbau der Testumgebung.

⁶CloudFlare Go-Crypto-Fork: <https://blog.cloudflare.com/go-crypto-bridging-the-performance-gap/>

⁷Go AES-NI-Patch-Merge: <https://go-review.googlesource.com/#/c/10484/>

⁸Go ECDSA-P256-Patch-Merge: <https://go-review.googlesource.com/#/c/8968/>

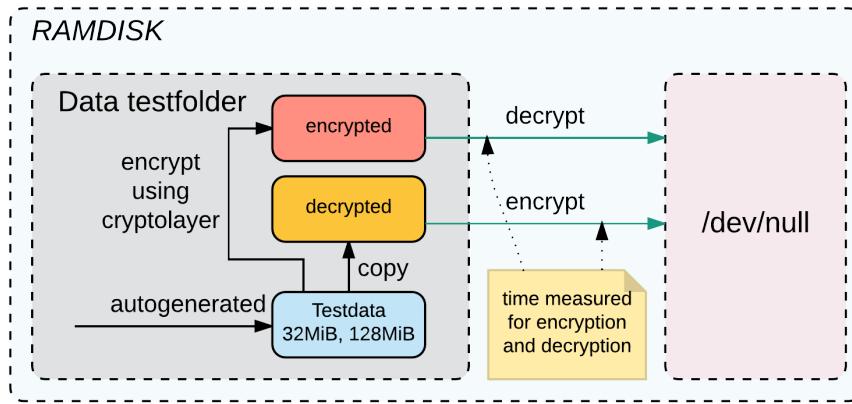


Abbildung 7.4.: Testablauf in der RAM-Disk bei der Erhebung der Messdaten.

Beim Erheben der Daten wurde wie folgt vorgegangen:

- ▶ Swap-Datei wurde mit `swapoff -a` deaktiviert, um ein Auslagern der Daten aus dem RAM auf die Festplatte zu verhindern.
- ▶ Eine RAM-Disk wurde mittels Skript angelegt.
- ▶ Verschlüsselte und entschlüsselte Daten werden mittels Skript erstellt.
- ▶ Daten werden verschlüsselt (Schreibvorgang) nach `/dev/null` geschrieben, Daten werden entschlüsselt (Lesevorgang) nach `/dev/null` geschrieben.
- ▶ Zeitmessung erfolgt jeweils beim Ver-/Entschlüsseln nach `/dev/null`.
- ▶ Die Messpunkte bilden den Mittelwert aus fünf Durchläufen, um statistische Ausreißer zu minimieren. Hier wurde bewusst der Mittelwert anstatt des maximal möglichen Werts genommen, um eine möglichst praxisnahe Abbildung zu erhalten.

Die Benchmark-Skripte sind unter [Anhang E](#) zu finden.

In der Praxis wird in der Regel beim Ver- und Entschlüsseln die Festplatte oder die Netzwerkanbindung der limitierende Faktor sein. Ob dies bei der Verschlüsselungsschicht jedoch pauschal, auch auf schwächeren Systemen der Fall ist, ist unklar.

7.2.5. Benchmarks

7.2.5.1. Einfluss der Blockgröße beim Ver- und Entschlüsseln

Die Verschlüsselungsschicht arbeitet aktuell mit einer Blockgröße von 64 KiByte. Diese Blockgröße wurde mehr oder weniger für den ersten Prototypen willkürlich festgelegt. Die Blockgröße entspricht hierbei dem internen Verschlüsselungspuffer, welcher zum Ver- und Entschlüsseln im Speicher allokiert wird.

Die beiden Auswertungen [Abb. 7.5](#) (Lesegeschwindigkeit) und [Abb. 7.6](#) (Schreibgeschwindigkeit) zeigen, welchen Einfluss die Wahl der Blockgröße auf die Geschwindigkeit hat. Unter Tabelle [Tabelle 7.2](#) (Lesegeschwindigkeit) und Tabelle [Tabelle 7.3](#) (Schreibgeschwindigkeit) sind jeweils die effizientesten Blockgrößen der Systeme zu entnehmen. Das Verschlüsselungsmodul wurde hierfür

mit der aktuellen Go Version 1.7.1 kompiliert.

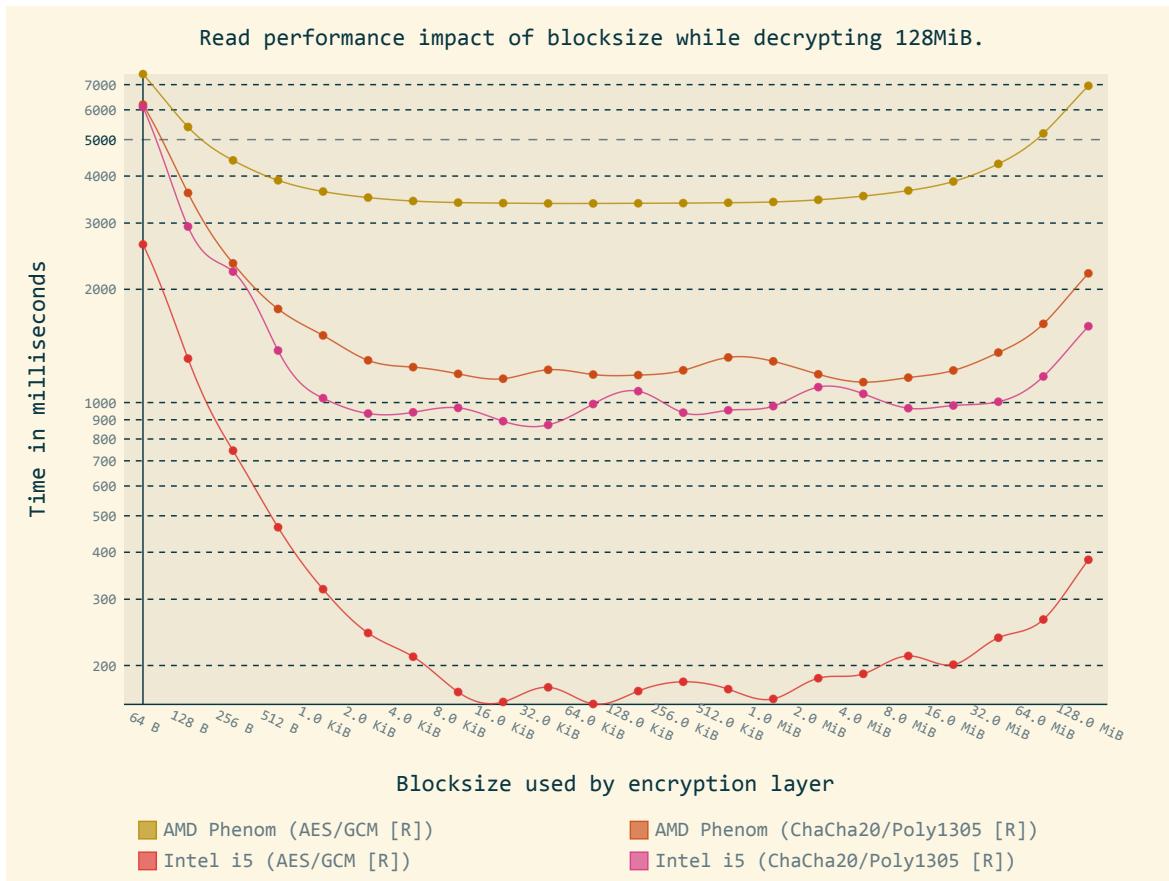


Abbildung 7.5.: Lesegeschwindigkeit des Kryptographielayers bei der Benutzung verschiedener Blockgrößen.

Tabelle 7.2.: Zeigt die effizientesten Blockgrößen beim Entschlüsseln. Der erste Wert entspricht der Zeit in Millisekunden, der zweite Wert der Geschwindigkeit in MiByte/s beim Lesen einer 128 MiByte großen Datei.

Blockgröße	32.0 KiByte	64.0 KiByte	4.0 MiByte
AMD (ChaCha20)	1222; 104.7	1187; 107.8	1133; 113.0
Intel (AES)	175; 731.4	158; 810.1	190; 673.7
Intel (ChaCha20)	872; 146.8	991; 129.2	1055; 121.3
AMD (AES)	3383; 37.8	3383; 37.8	3539; 36.2

Hierbei wurden die Systeme mit einer Datei der Größe von 128 MiByte getestet. Diese Datei wurde jeweils komplett mehrmals in der RAM-DISK mit verschiedenen Blockgrößen verschlüsselt und wieder entschlüsselt.

Beim Entschlüsseln Abb. 7.5 ist erkennbar, dass die Geschwindigkeit bei beiden Algorithmen unterhalb 4 KiByte einbricht. Im Mittelfeld ist die Geschwindigkeit stabil, ab einer Blockgröße von oberhalb 32

MiByte scheint die Geschwindigkeit wieder zu degenerieren.

Ähnlich wie beim Entschlüsseln, zeigt die Verschlüsselung einen vergleichbaren Verlauf. Hier ist die Geschwindigkeit unterhalb 4 KiByte Blockgröße rückläufig. Der obere Grenzwert für die Blockgröße ist beim Verschlüsseln weniger gut erkennbar. Hier bricht die Geschwindigkeit verglichen mit dem Entschlüsseln nur beim Intel-System mit AES-Algorithmus ab ungefähr 32 MiByte ein.

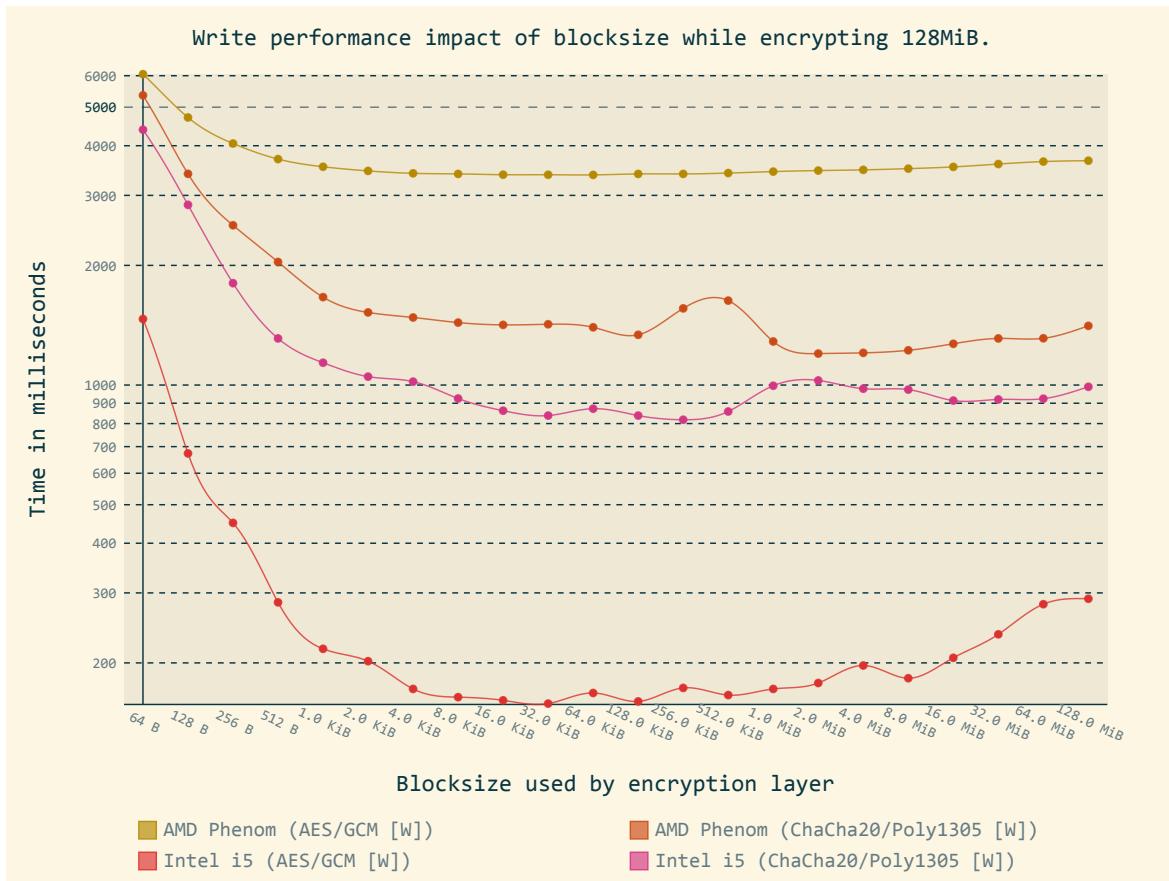


Abbildung 7.6.: Schreibgeschwindigkeit des Kryptographielayers bei der Benutzung verschiedener Blockgrößen.

Tabelle 7.3.: Zeigt die effizientesten Blockgrößen beim Verschlüsseln. Der erste Wert entspricht der Zeit in Millisekunden, der zweite Wert der Geschwindigkeit in MiByte/s beim Schreiben einer 128 MiByte großen Datei.

Blockgröße	32.0 KiByte	64.0 KiByte	256.0 KiByte	2.0 MiByte
AMD (AES)	3381; 37.9	3378; 37.9	3397; 37.7	3464; 37.0
AMD (ChaCha20)	1422; 90.0	1397; 91.6	1559; 82.1	1200; 106.7
Intel (AES)	158; 810.1	168; 761.9	173; 739.9	178; 719.1
Intel (ChaCha20)	838; 152.7	872; 146.8	818; 156.5	1027; 124.6

Über die Faktoren bei den großen Blockgrößen kann nur gemutmaßt werden, dass hier der Geschwin-

digkeitseinbruch mit dem Speichermanagement/Speicherallokierung zusammenhängen könnte.

7.2.5.2. Einfluss des AES-NI-Befehlserweiterungssatzes beim Ver- und Entschlüsseln

Abb. 7.7 zeigt den Geschwindigkeitszuwachs, der durch die Nutzung des AES-NI-Befehlserweiterungssatzes zustande kommt. Hier wurde die Verschlüsselungsschicht mit verschiedenen Go-Versionen kompiliert, um zu sehen, wie stark sich die Geschwindigkeit ab Go Version 1.6 (Merge des Cloudflare-AES-NI-Patches⁹) verändert hat.

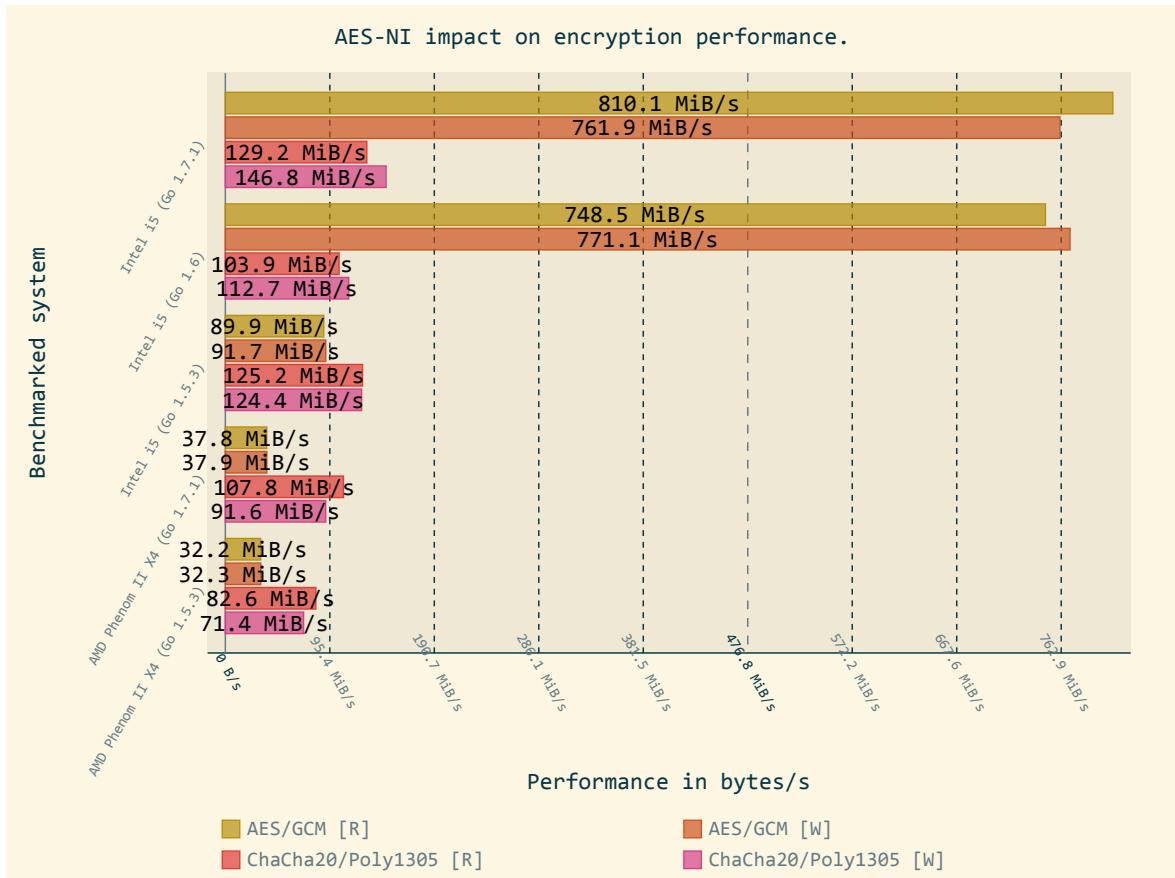


Abbildung 7.7.: Geschwindigkeitszuwachs durch AES-NI.

Weiterhin wurde das AMD-System, welches kein AES-NI unterstützt zum Vergleich mit in die Auswertung aufgenommen. Hier zeigt sich ein großer Unterschied beim AES/GCM-Verfahren zwischen den beiden Systemen wenn man die Go-Version 1.5.3 mit der Version 1.7.1 vergleicht. Die ChaCha20/Poly1305-Implementierungen weisen einen Geschwindigkeitszuwachs von ~20–30% auf, die AES/GCM-Implementierung hingegen hat beim AMD-System jedoch einen Geschwindigkeitszuwachs von nur ~15%, das Intel-System hingegen kann seine Geschwindigkeit jedoch aufgrund der funktionierenden AES-NI-Beschleunigung um ~750% (!) steigern. Weiterhin ist auffällig, dass der ChaCha20/Poly1305-Algorithmus auf dem Intel-System mit Go-Version 1.6 langsamer ist als die Go-Version 1.5.3 Variante. Diese Regression ist reproduzierbar, die Ursache hierfür ist unklar.

⁹Go AES-NI-Patch-Merge: <https://go-review.googlesource.com/#/c/10484/>

7.2.5.3. Schwächere Systeme

Neben leistungsfähigeren Systemen soll auch die Geschwindigkeit von schwächeren Systemen, wie beispielsweise dem weit verbreiteten Raspberry Pi evaluiert werden.

Für das Erstellen der Benchmark-Binary für die schwächeren Systeme wurde die Go-Cross-Compiler-Funktionalität verwendet. Hierbei wurden die Binaries für die beiden Systeme mit folgenden Parametern kompiliert:

- ▶ Raspberry Pi: GOARM=6 GOARCH=arm GOOS=linux go build main.go
- ▶ Intel Atom SSE2: GOARCH=386 GO386=sse2 go build main.go
- ▶ Intel Atom FPU-387: GOARCH=386 GO386=387 go build main.go

Ein kurzer Testdurchlauf vor dem eigentlichen Benchmark hat gezeigt, dass die beiden Systeme (Intel Atom, Raspberry Pi) im Vergleich zum Intel i5/AMD Phenom so langsam sind, dass eine Durchführung des Benchmarks mit den gleichen Parametern in keiner akzeptablen Zeit durchgeführt werden kann. Aus diesem Grund wurde die Anzahl der Durchläufe auf drei, und die zu verarbeitende Dateigröße auf 32 MiByte reduziert.

Abb. 7.8 zeigt den Einfluss der Blockgröße beim Lesen/Entschlüsseln auf den schwächeren Systemen. In der Tabelle 7.4 sind jeweils die effizientesten Blockgrößen beim Entschlüsseln zu entnehmen. Hier zeigt sich ein vergleichbares Verhalten wie bei den stärkeren Systemen. Unterhalb 4 KiByte bricht die Lesegeschwindigkeit stark ein. Verglichen zu den stärkeren Systemen bricht hier die Lesegeschwindigkeit bereits ab einer Blockgröße von ungefähr 2 MiByte ein.

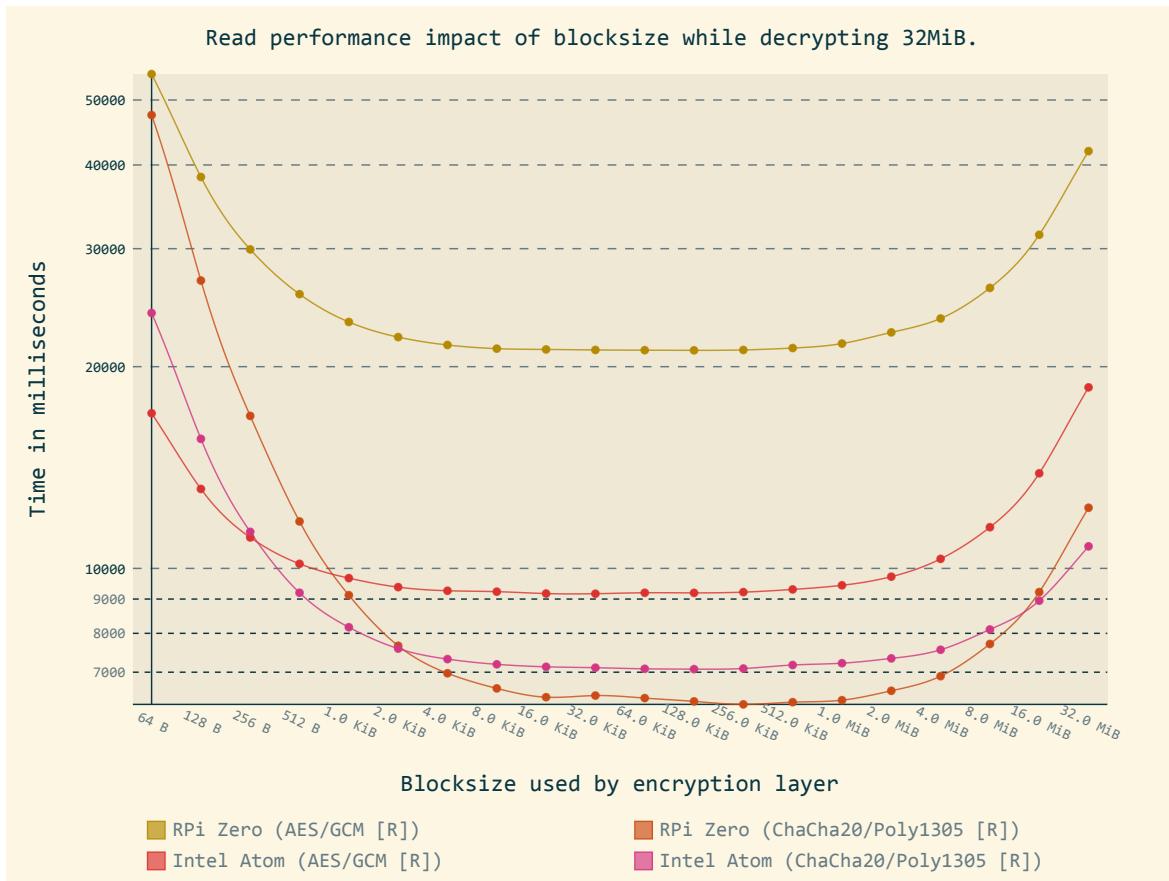


Abbildung 7.8.: Lesegeschwindigkeit der Verschlüsselungsschicht auf schwächeren Systemen bei der Benutzung verschiedener Blockgrößen.

Tabelle 7.4.: Zeigt die effizientesten Blockgrößen beim Entschlüsseln. Der erste Wert entspricht der Zeit in Millisekunden, der zweite Wert der Geschwindigkeit in MiByte/s beim Lesen einer 32 MiByte großen Datei.

Blockgröße	32.0 KiByte	128.0 KiByte	256.0 KiByte
Intel Atom (AES)	9167; 3.5	9193; 3.5	9217; 3.5
Intel Atom (ChaCha20)	7109; 4.5	7074; 4.5	7089; 4.5
RPi Zero (AES)	21183; 1.5	21155; 1.5	21183; 1.5
RPi Zero (ChaCha20)	6460; 5.0	6332; 5.1	6267; 5.1

Abb. 7.9 zeigt den Einfluss der Blockgröße beim Schreiben/Verschlüsseln auf schwächeren Systemen. Unter Tabelle **Tabelle 7.5** sind jeweils die effizientesten Blockgrößen beim Verschlüsseln auf den schwächeren Systemen zu finden. Hier bricht die Geschwindigkeit, wie bei den restlichen getesteten System, unterhalb der 4 KiByte stark ein. Eine obere Blockgröße, bei der die Geschwindigkeit einbricht, ist bei den schwächeren Systemen weniger gut erkennbar.

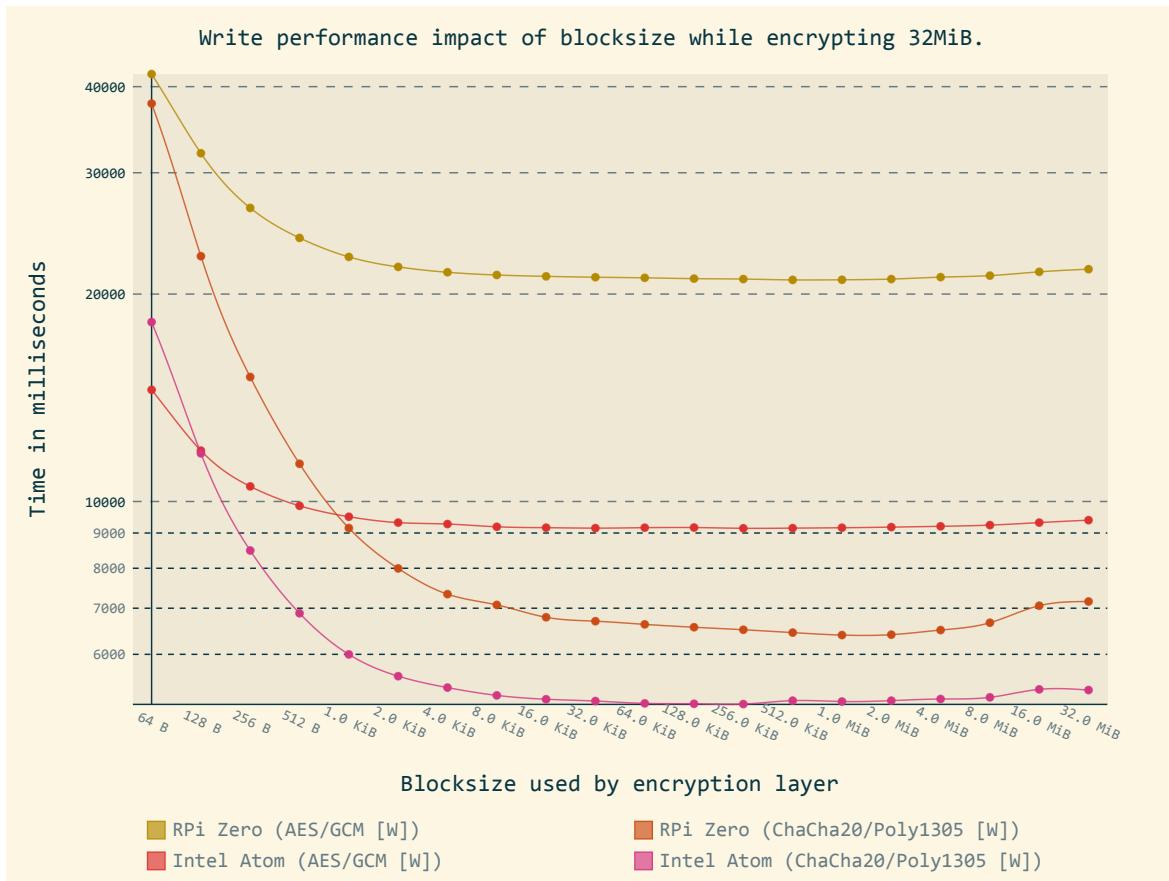


Abbildung 7.9.: Schreibgeschwindigkeit der Verschlüsselungsschicht auf schwächeren Systemen bei der Benutzung verschiedener Blockgrößen.

Tabelle 7.5.: Zeigt die effizientesten Blockgrößen beim Verschlüsseln. Der erste Wert entspricht der Zeit in Millisekunden, der zweite Wert der Geschwindigkeit in MiByte/s beim Schreiben einer 32 MiByte großen Datei.

Blockgröße	256.0 KiByte	512.0 KiByte	1.0 MiByte
Intel Atom (AES)	9143; 3.5	9148; 3.5	9160; 3.5
Intel Atom (ChaCha20)	5081; 6.3	5140; 6.2	5125; 6.2
RPi Zero (AES)	21029; 1.5	20969; 1.5	20978; 1.5
RPi Zero (ChaCha20)	6514; 4.9	6452; 5.0	6399; 5.0

Trotz des reduzierten Benchmarkaufwandes lagen die Zeiten für die Durchführung des Benchmarks bei 1 Stunde 30 Minuten (Raspberry Pi) und 40 Minuten (Intel Atom). **Abb. 7.10** zeigt den Einbruch

der Geschwindigkeit beim Ver- und Entschlüsseln auf den schwächeren Systemen. Der Wert Base zeigt hier die Lese- und Schreibgeschwindigkeit der Systeme ohne Verschlüsselungsschicht. Auf beiden Systemen ist ein starker Geschwindigkeitseinbruch unter Verwendung von Verschlüsselung zu verzeichnen. Betrachtet man die tatsächliche Geschwindigkeit beim Lesen und Schreiben auf die Festplatte mit dem dd-Kommandozeilen-Werkzeug (Tabelle 7.1), so fällt weiterhin auf, dass bei den beiden Systemen die Verschlüsselung und nicht die langsame Festplatte der limitierende Faktor ist. Auffällig ist auch, dass das Intel-Atom-System ohne SSE2-Optimierungen schneller ist.

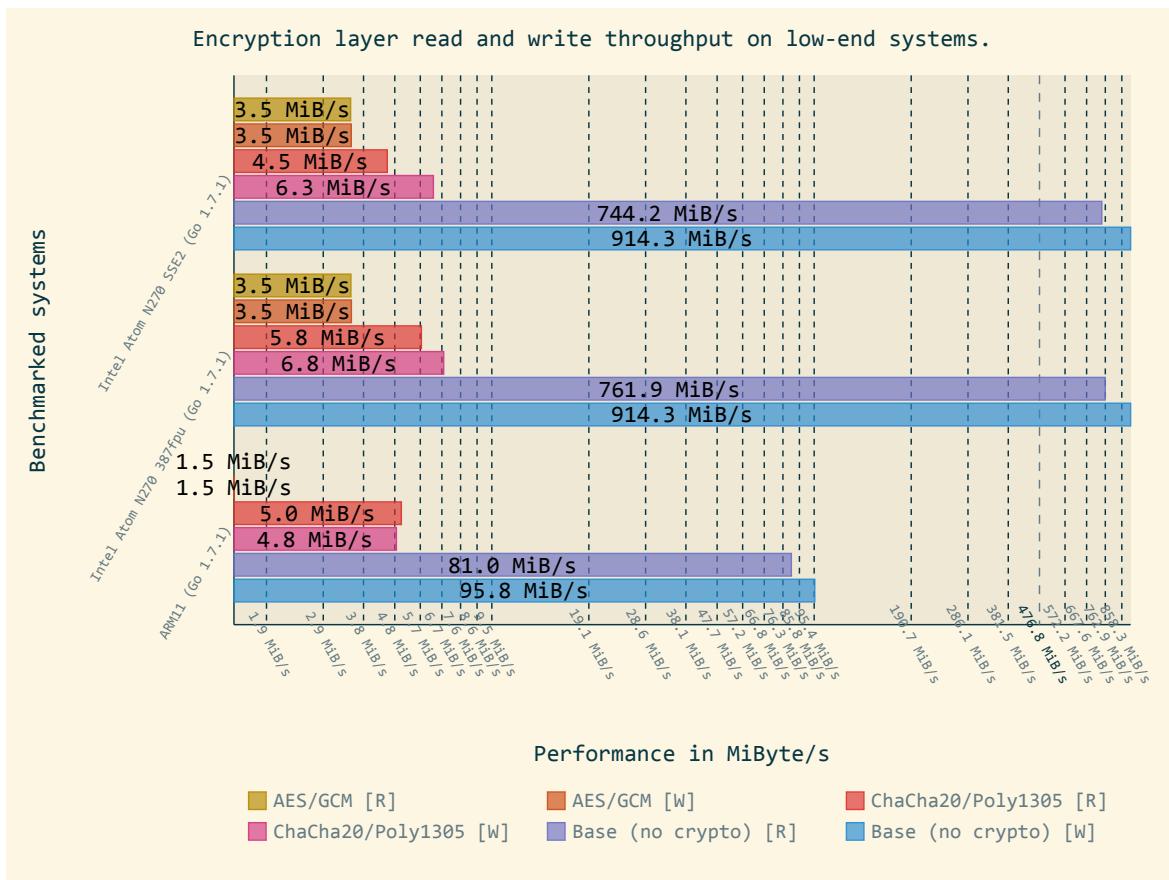


Abbildung 7.10.: Geschwindigkeitseinbruch unter Verwendung von Verschlüsselung auf schwächeren Systemen.

Weiterhin ist in der Grafik ersichtlich, dass der Chacha20/Poly1305-Algorithmus bei diesen schwächeren Systemen, verglichen mit AES/GCM, bessere Ver- und Entschlüsselungsgeschwindigkeiten liefert.

7.2.5.4. Schlüsselgenerierung

Aktuell wird für jede Datei ein Schlüssel zufällig generiert. Dieser wird in den Metadaten abgelegt. Durch das zufällige Generieren eines Schlüssels wird bei zwei unterschiedlichen Kommunikationspartnern für die gleiche Datei ein unterschiedlicher Schlüssel erstellt. Ein großer Nachteil, der aktuell dadurch zum Tragen kommt, ist, dass die IPFS-Deduplizierung nur noch stark eingeschränkt funktioniert.

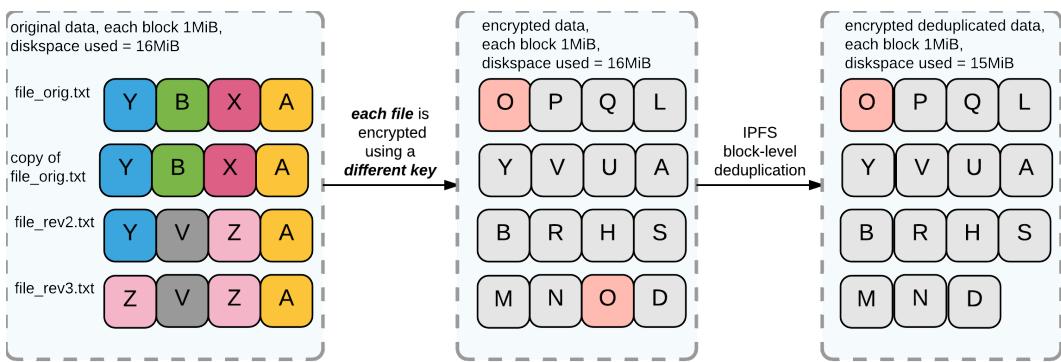


Abbildung 7.11.: Für die gleiche Datei werden aktuell unterschiedliche Schlüssel generiert. Das hat zur Folge, dass die Deduplizierungsfunktionalität von IPFS weitestgehend nicht mehr funktioniert.

Abb. 7.11 zeigt den aktuellen Ansatz. Durch den zufällig generierten Schlüssel haben die verschlüsselten Dateien und Datenblöcke – auch bei gleichem Input – einen unterschiedlichen Output. Dies hat zur Folge, dass IPFS die Daten nicht mehr sinnvoll deduplizieren kann, wie unter (siehe Abschnitt 7.2.5.4) abgebildet. Eine möglicher Ansatz dieses Problem zu umgehen oder abzumildern ist Convergent Encryption, siehe Abschnitt 8.1. Der in der Abbildung gezeigte Fall, dass es im geringen Maße trotzdem zu einer Deduplizierung kommen kann (Block O), ist sehr unwahrscheinlich.

Beim Benchmark-Tool wurde beim Verschlüsseln eine Geschwindigkeits-Anomalie bei der Verwendung verschiedener Dateigrößen festgestellt. Nach eingehender Recherche wurde hierfür die Ableitung des Schlüssels mittels scrypt¹⁰ als Ursache identifiziert.

Abb. 7.12 zeigt den Verlauf des Overheads bei verschiedenen Dateigrößen verglichen mit zwei weiteren Verfahren. Die scrypt-Schlüsselableitungsfunktion fällt bei kleinen Dateien stark ins Gewicht. »brig« verwendet aktuell einen zufällig generierten Schlüssel (in Abb. 7.12 Dev Random generated key).

¹⁰Scrypt: <https://de.wikipedia.org/w/index.php?title=Scrypt&oldid=158788414>

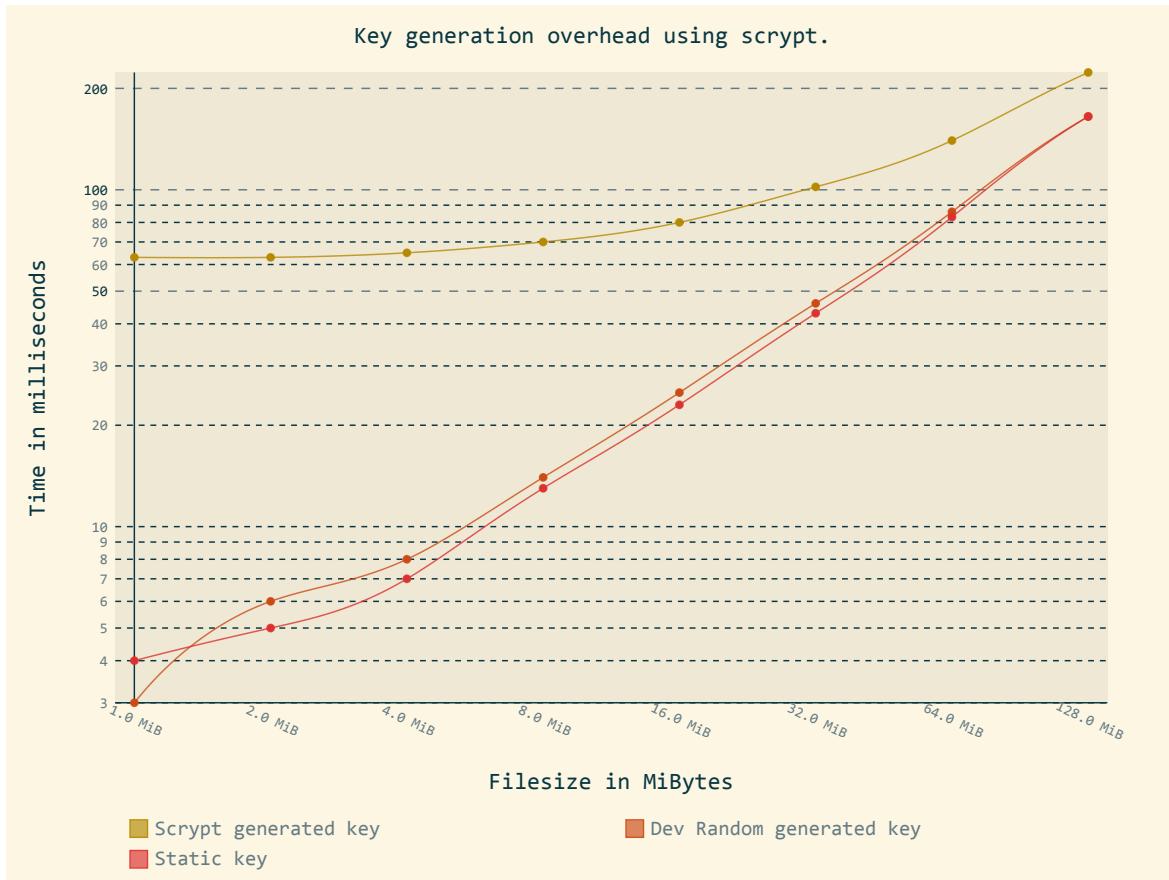


Abbildung 7.12.: Geschwindigkeitseinbruch verursacht durch Schlüsselableitung mittels der scrypt-Schlüsselableitungsfunktion.

7.2.5.5. Zusammenfassung Geschwindigkeitsevaluation

Zusammengefasst kann gesagt werden, dass sich die Blockgröße beim Verschlüsseln und Entschlüsseln im Bereich von 32 KiByte bis wenigen MiByte bewegen sollte. Blockgrößen außerhalb dieses Bereiches haben laut den Benchmarkergebnissen einen negativen Einfluss auf die Geschwindigkeit der Verschlüsselungsschicht. Weiterhin scheint der ChaCha20/Poly1305–Algorithmus für schwächere Systeme besser als AES/GCM geeignet zu sein.

7.3. Metadatenverschlüsselung

Neben den Nutzdaten, die von IPFS verwaltet werden, werden weiterhin die sogenannten Stores verschlüsselt. Diese beinhalten den Metadatenstand der jeweiligen Synchronisationspartner.

Folgend ist die Struktur eines neu initialisierten »brig«–Repositories (vgl. auch [27], S. 73 ff.) abgebildet:

```
$ tree -al alice
alice
|-- .brig
    |-- index # Stores der Synchronisationspartner
```

```

|   |-- alice@ipfsnetwork.de.locked
|-- config
|-- ipfs
|   |-- config
|   |...
|   |-- version
|-- master.key.locked
|-- remotes.yml.locked # Bekannte Synchronisationspartner
|-- shadow

```

Die Dateien mit der Endung `locked` sind durch »brig« verschlüsselt. Als »Einstiegspunkt« für den Zugriff auf das Repository fungiert aktuell eine Passwort-Abfrage. Das Passwort ist samt zufällig generiertem Salt als SHA-3-Repräsentation in der `shadow`-Datei¹¹ gespeichert.

Die verschlüsselte Remotes-Datei beinhaltet den Benutzernamen mit dazugehöriger Peer-ID und einen Zeitstempel (letzter Onlinestatus) für die jeweils bekannten (authentifizierten) Synchronisationspartner.

Einschätzung: Das IPFS-Repository, sowie das Schlüsselpaar von IPFS ist aktuell unverschlüsselt. Dies würde diverse Modifikationen am Repository erlauben wie beispielsweise die Manipulation der Peer-ID von IPFS. Der `master.key` hat aktuell keine Verwendung.

7.4. »brig«-Identifier

Da IPFS an sich keinen Authentifizierungsmechanismus bietet, muss dieser von »brig« bereitgestellt werden. Im IPFS-Netzwerk haben die Peers durch die Prüfsumme über den öffentlichen Schlüssel eine eindeutige Kennung. Diese Prüfsumme ist aufgrund des Aufbaus und der Länge als menschenlesbare Kennung nicht geeignet. Aus diesem Grund wurde ein »brig«-Identifier (»brig«-ID) eingeführt.

Die »brig«-ID repräsentiert den Benutzer mit einem Benutzernamen im »brig«-Netzwerk nach außen. Der Aufbau dieses Namens ist an die Semantik des XMPP-Standard¹² angelehnt und mit dem Präfix `brig#user:` versehen. Die eigentliche »brig«-ID entspricht hier der Anlehnung an den XMPP-Standard, der Präfix wird intern von »brig« angehängt, um die Benutzer des »brig«-Netzwerks identifizieren zu können.

Folgendes Beispiel mit dem Multihash-Tool vom IPFS-Projekt zeigt die Generierung einer User-Hash aus dem Benutzernamen.

```
$ echo 'brig#user:alice@university.cc' | multihash -
QmYRofJVzXGsL4njv5BW7HNhCkpLCiCjQvrqesbm7TWUCe
```

Diese Definition ermöglicht es Organisationen ihre Mitarbeiter und deren Ressourcen im »brig«-Netzwerk abzubilden. Weiterhin hat es den Vorteil, dass eine E-Mail-Adresse auch einen korrekten Benutzernamen darstellen würde.

¹¹Quellcode: <https://github.com/disorganizer/brig/blob/fa9bb634b4b83aaabaa967ac523123ce67aa217d/repo/shadow.go>

¹²Jabber-ID: https://de.wikipedia.org/w/index.php?title=Jabber_Identifier&oldid=147048396

Um diese Funktionalität bereitzustellen, wird ein »Trick« angewendet, bei welchem die Zeichenkette des Nutzernamens als Block dem IPFS–Netzwerk bekannt gemacht wird (vgl. [27], S. 62). Dieser Block selbst ist nicht eindeutig und könnte auch von einem Angreifer selbst erstellt worden sein. Um eine Eindeutigkeit herzustellen, wird der Benutzername direkt an die eindeutige, öffentliche ID (siehe Abschnitt 6.4) geknüpft.

Folgende Daten werden kombiniert, um einen Benutzer eindeutig zu identifizieren:

- ▶ **Peer-ID:** Prüfsumme über den öffentlichen IPFS–Schlüssel.
- ▶ **User-Hash:** Prüfsumme über die »brig«-ID, welche einen vom Benutzer gewählten Identifier darstellt.

Abb. 7.13 zeigt das Auffinden von einem Benutzer im IPFS–Netzwerk. Für weitere Details zur Softwarearchitektur und Funktionsweise siehe auch [27].

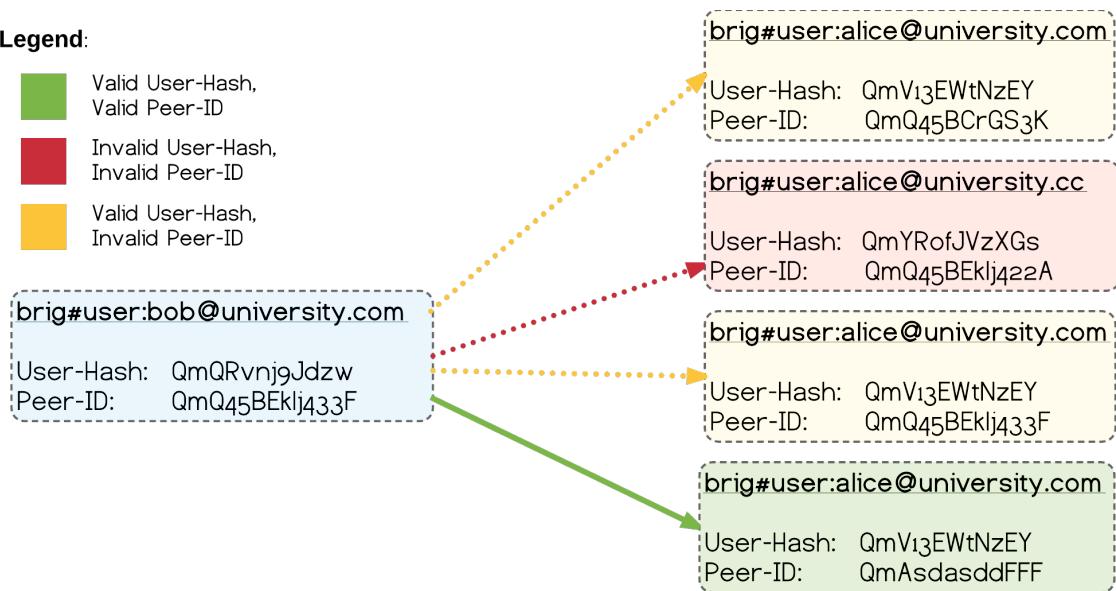


Abbildung 7.13.: User–Lookup mittels »brig«-ID (hier bestehend aus gekürzter Peer-ID + User-Hash). Nur bei Übereinstimmung vom Peer–Fingerprint und Benutzernamen–Fingerprint wird der Benutzer als valide erkannt.

7.5. Authentifizierung

Eine Schwierigkeit, die sich im Voraus stellt, ist die sichere Authentifizierung. Mit der »brig«-ID ist es aufgrund des Multihash vom öffentlichen IPFS–Schlüssel möglich, den Benutzer eindeutig zu identifizieren. Bei der aktuellen »brig«–Version muss der Fingerabdruck beim Hinzufügen des Synchronisationspartners manuell hinzugefügt werden. Dies setzt voraus, dass die Synchronisationspartner ihre IPFS–Fingerabdrücke vorab austauschen. Anschließend können beide Synchronisationspartner ihre Repositories synchronisieren.

```
# *Alice* fügt *Bob* (Bob's brig-ID:bob@jabber.nullcat.de/desktop,
# Bob's IPFS-Fingerabdruck: QmbR6tDXRCgpRwWZhGG3qLfJMKrLcrgk2q)
# als Synchronisationspartner hinzu
brig remote add bob@jabber.nullcat.de/desktop QmbR6tDXRCgpRwWZhGG3qLfJMKrLcrgk2q
```

Analog dazu muss auch *Alice* von *Bob* als Synchronisationspartner hinzugefügt werden.

7.6. Repository-Zugriff

Um Zugriff auf das »brig«-Repository zu erhalten, muss sich der Benutzer »brig« gegenüber mit einem Passwort authentifizieren. Schlechte Passwörter sind ein großes Problem im Informationszeitalter, da von ihnen die Sicherheit eines gesamten Systems abhängen kann. Menschen sind schlecht darin, die Entropie¹³ von Passwörtern richtig einzuschätzen. »brig« verwendet für die Bestimmung der Passwort-Entropie die zxcvbn-Bibliothek, welche von Dropbox entwickelt wurde. Laut einer Studie [29], S. 11 wird die Bibliothek, im Vergleich zu den getesteten Konkurrenten, in ihrer Funktionsweise als akkurate bezeichnet. Eine Schwäche, welche bei den Entropie-Schätzungsgeräten auftritt, ist, dass diese ohne Basis eines Wörterbuchs arbeiten und somit bei Zeichenketten Brute-Force als schnellsten Angriff annehmen. Weiterhin ist die Schätzung einer sicheren Entropie schwierig, da diese stark vom Angriffszenario abhängt. Die Schätzspanne für ein sicheres Passwort liegt zwischen 29 Bit (Online-Passwörter) – 128 Bit oder mehr (Sicherung kryptographischer Schlüssel)¹⁴.

Wie bereits unter Punkt Passwortmanagement, Abschnitt 5.1 erwähnt, ist der Einsatz von Passwörtern problematisch. In den Berichten von *Bruce Schneier* und *Dan Goodin* wird erwähnt, dass heutzutage mit modernen Methoden und moderner Hardware auch Passwörter, die bisher von vielen Benutzern als hinreichend sicher angesehen waren, nicht mehr verwendet werden sollten. Dazu gehören insbesondere Passwörter, bei welchen Buchstaben durch Sonderzeichen oder Zahlen ausgetauscht wurden. Tabelle Tabelle 7.6 listet einen Teil der Passwörter, die laut *Bruce Schneier* während der sogenannten cracking session »gecrackt« wurden. Diese Passwörter sind definitiv als unsicher anzusehen. Die Tabelle zeigt die geschätzte Entropie und Crackzeit (hier ist leider die genaue Hardware nicht bekannt) der genannten Passwörter. In die Cygnius-Schätzungen¹⁵ fließen neben der zxcvbn-Bibliothek noch weitere Prüfungen (Länge, Kleinbuchstaben, Großbuchstaben, Zahlen, Sonderzeichen), welche eine zusätzliche Passwortakzeptanz-Aussage machen, ein. Die bennish-Plattform¹⁶ arbeitet ebenso mit der zxcvbn-Bibliothek.

Tabelle 7.6.: Geschätzte Passwort-Entropie und Crackdauer von unsicheren Passwörtern.

Passwort	bennish(Entropie/Zeit)	Cygnius(Entropie/Zeit/Akzept.)
k1araj0hns0n,	19.868/instant	21.372/4 minutes/no
Sh1a+labe0uf,	63.847/centuries	53.613/centuries/yes
Apr!l221973,	31.154/3 days	27.835/5 hours/no

¹³Password-Entropy: https://en.wikipedia.org/w/index.php?title=Password_strength&oldid=761771461#Entropy_as_a_measure_of_password_strength

¹⁴Randomness Requirements for Security: <https://tools.ietf.org/html/rfc4086#section-8.2>

¹⁵Cygnius Password Strength Test: <https://apps.cygnius.net/passtest/>

¹⁶Password Strength Checker: <https://www.bennish.net/password-strength-checker/>

Passwort	bennish(Entropie/Zeit)	Cygnius(Entropie/Zeit/Akzept.)
Qbesancon321,	45.326/70 years	41.625/7 years/yes
DG091101%,	44.116/31 years	34.936/21 days/no
@Yourmom69,	35.049/22 days	35.904/3 months/no
tmdmmj17,	43.413/20 years	41.921/8 years/no
iloveyousomuch,	24.256/18 minutes	27.902/5 hours
Philippians4:6+7,	60.662/centuries	39.915/3 years/no
qeadzcwrsfxv1331.	83.148/centuries	78.795/centuries/no

Betrachtet man die Tabelle, so würde sie dem Benutzer nach Aussagen der Entropie-Schätzwerkzeuge ein falsches Sicherheitsgefühl vermitteln. Eine Empfehlung an dieser Stelle wäre ein zufällig generiertes Passwort wie beispielsweise »iyLGBu<tmr"6!w-s.1fT« und die Verwendung eines Passwort-Manager¹⁷.

Einschätzung: Bei der aktuellen Authentifizierung gegenüber dem Repository ist ein (schlechtes) Passwort oder die erzwungene Komplexität (Benutzer schreiben komplexe Passwörter auf Post-it's auf) eine Schwachstelle. Weiterhin ist auch problematisch, dass der geheime Schlüssel von IPFS nicht verschlüsselt abgelegt ist. Dieser Umstand ermöglicht beispielsweise einen Identitätsdiebstahl.

¹⁷The secret to online safety - Lies, random characters, and a password manager: <http://arstechnica.com/information-technology/2013/06/the-secret-to-online-safety-lies-random-characters-and-a-password-manager/>

7.7. Aufbau einer verschlüsselten Verbindung

Abb. 7.14 zeigt den Ablauf beim Aufbau einer verschlüsselten Verbindung zwischen zwei Synchronisationspartnern.

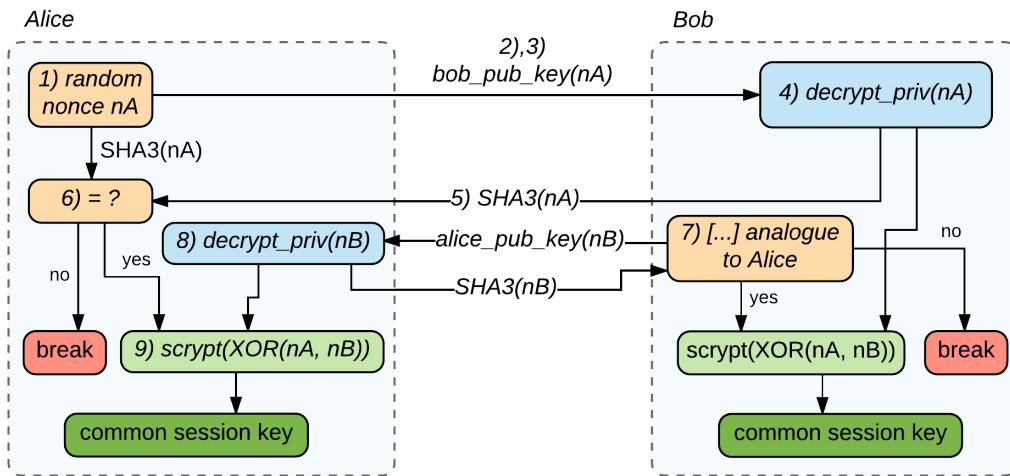


Abbildung 7.14.: Vereinfachte Ansicht bei der Aushandlung eines Sitzungsschlüssel beim Verbindungsauftbau.

Ablauf aus der Sicht von Alice:

1. Alice generiert eine zufälligeNonce nA .
2. Alice verschlüsselt nA mit dem öffentlichen Schlüssel von Bob.
3. Alice schickt verschlüsselteNonce nA an Bob ($\text{bob_pub_key}(nA)$).
4. Bob entschlüsselt dieNonce nA von Alice mit seinem privaten Schlüssel.
5. Bob generiert eine Prüfsumme derNonce nA und schickt diese an Alice.
6. Alice vergleicht die von Bob erhaltene Prüfsumme $\text{SHA3}(nA)$ mit ihrer eigenen. Stimmt diese überein, so kann der Vorgang fortgesetzt werden, ansonsten wird dieser abgebrochen.
7. Analog zu Alice hat Bob auf die gleiche Art und Weise seineNonce nB mit Alice ausgetauscht.
8. Alice entschlüsselt dieNonce nB von Bob und lässt diese in die Schlüsselgenerierung einfließen.
9. Der gemeinsame Schlüssel ist nun eine XOR-Verknüpfung der beiden Noncen nA und nB . Um den Schlüssel zu verstärken wird zusätzlich die Schlüsselableitungsfunktion scrypt (vgl. [30]) angewendet.

Einschätzung: Der Aufbau der verschlüsselten Verbindung setzt eine vorherige Authentifizierung des jeweiligen Synchronisationspartner voraus. Wäre dieser nicht authentifiziert, so wäre in diesem Fall ein Man-in-the-Middle-Angriff (kurz MITM-Angriff)¹⁸ denkbar. Weiterhin wäre bei der aktuellen Implementierung ein Replay-Angriff wahrscheinlich möglich, da die freshness derNonce nicht gegeben ist (vgl. [21], S. 259 ff.). Eine zusätzliche freshness sollte hier das Problem beheben.

Die aktuelle Softwareversion bietet hier keinen Automatismus und auch keinen Authentifizierungsmechanismus, wie er beim Pidgin-Messenger mit OTR-Verschlüsselung vorhanden ist.

¹⁸Man-in-the-Middle-Angriff:

<https://de.wikipedia.org/w/index.php?title=Man-in-the-Middle-Angriff&oldid=161582213>

7.8. Entwicklung und Entwicklungsumgebung

7.8.1. Sichere Softwareentwicklung

Bei Software, welche das Augenmerk auf einen gewissen Sicherheitsstandard legt, sollte neben dem korrekten Einsatz von kryptographischen Elementen bereits der Entwicklungsprozess gewissen Mindestanforderungen genügen. Welche dies im konkreten Fall sind, hängt stark vom Projekt, dem Aufwand und dem Einsatzzweck ab.

Da ein Hauptaugenmerk von »brig« der sichere und dezentrale Austausch von Daten ist und es zusätzlich als Open-Source-Projekt angelegt ist, ist ein detaillierter Überblick zur Entwicklung beziehungsweise der Projektbeteiligten wichtig.

7.8.2. »brig«-Quellcode-Repository

Aktuell wird »brig« mit der Quelltext-Versionsverwaltungssoftware git verwaltet. Da es sich um ein Open-Source-Projekt handelt, ist das Repository öffentlich auf der GitHub-Plattform zu finden.

Einschätzung: Quelltexte/Releases sind bisher nicht signiert. Updates am Repository werden aktuell über den von den Entwicklern hinterlegten SSH-Schlüssel gepusht beziehungsweise gepullt. Der Zugriff auf die GitHub-Plattform erfolgt aktuell zum Teil über eine Kombination aus Benutzername und Passwort. Hier würde sich die Aktivierung der von GitHub angebotenen Zwei-Faktor-Authentifizierung aus Gründen der Sicherheit anbieten.

Problematisch ist aktuell der Umstand, dass die Urheber des Quelltextes nicht direkt authentifiziert werden können. Durch diesen Umstand wird es Personen (Angreifern) einfacher gemacht, sich als Entwickler unter einer falschen Identität auszugeben.

7.8.3. Update-Mechanismus

Im aktuellen Stadium kann »brig« über das Beziehen des Quelltextes von GitHub installiert werden. Ein Update-Mechanismus existiert nicht, die Validierung der Integrität des Quelltextes ist aktuell nur auf Basis der git SHA-1-Integrität möglich.

Einschätzung: Aktuell ist es für den Benutzer nicht möglich, den Ursprung der heruntergeladenen Software zu validieren.

Auf Basis der Evaluation von »brig« sollen nun mögliche Konzepte vorgestellt werden, um die in der Evaluation identifizierten Schwächen abzumildern beziehungsweise zu beheben.

8.1. Datenverschlüsselung

Aktuell verwendet die Implementierung der Datenverschlüsselungsschicht 8 Bytes große Noncen. Die MAC ist inklusive Padding 16 Bytes groß (siehe [Abschnitt 7.2.1](#)). Hier muss laut BSI-Richtlinie nachgebessert werden. Die empfohlene Noncen-Größe ist mit 96 Bit angegeben (vgl. [20], S. 24).

Wie unter [Abschnitt 6.4](#) zu sehen, sind die IPFS-Schlüssel aktuell in der Konfigurationsdatei von IPFS im Klartext hinterlegt. »brig« verschlüsselt diese Datei zum aktuellen Zeitpunkt nicht. Hier wäre eine Verschlüsselung mit einem Repository-Key möglich, welcher wiederum durch einen Hauptschlüssel geschützt werden sollte (siehe [Abschnitt 8.2](#)). Eine weitere Überlegung wäre, das gesamte Repository mittels eines externen Hauptschlüssels zu verschlüsseln.

Wie unter [Abschnitt 7.2.5.4](#) erläutert, wird aktuell für jede Datei ein zufälliger Schlüssel generiert. Mit diesem Ansatz wird die Deduplizierungsfunktionalität von IPFS weitestgehend nutzlos gemacht.

Ein Ansatz dieses Problem zu umgehen, ist die sogenannte Convergent Encryption. Diese Technik wird beispielsweise von Cloud-Speicher-Anbietern verwendet, um verschlüsselte Daten deduplizieren zu können, ohne dabei auf den eigentlichen Inhalt zugreifen zu müssen (vgl. [31]).

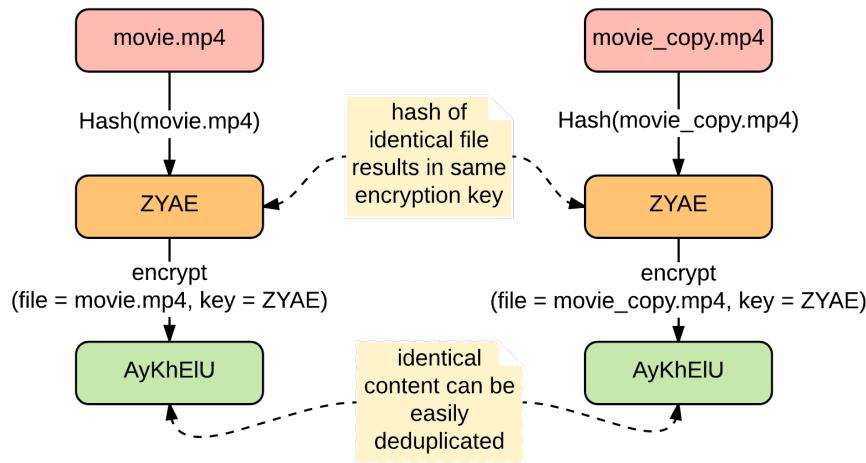


Abbildung 8.1.: Bei Convergent Encryption resultiert aus gleichem Klartext der gleiche Geheimtext da der Schlüssel zur Verschlüsselung vom Klartext selbst abgeleitet wird.

Wie in Abb. 8.1 zu sehen, wird hierbei beispielsweise der Schlüssel zum Verschlüsseln einer Datei von dieser selbst mittels einer kryptographischen Hashfunktion abgeleitet.

Dieses Verfahren lässt sich jedoch bei der aktuellen Architektur (separate Verschlüsselungsschicht) nur eingeschränkt realisieren, da die Prüfsumme der Daten erst nach dem Hinzufügen zum IPFS bekannt ist. Um die Daten zu verschlüsseln, müssten diese vor dem Hinzufügen komplett gehasht werden. Dies würde bedeuten, dass man die Daten insgesamt zweimal einlesen müsste (1. Prüfsumme generieren, 2. brig stage), was bei vielen und/oder großen Dateien ineffizient wäre.

Ein Kompromiss wäre beispielsweise anstatt der kompletten Prüfsumme über die ganze Datei, nur die Prüfsumme über einen Teil (beispielsweise 1024 Bytes vom Anfang der Datei) der Datei zu bilden und zusätzlich die Dateigröße mit in die Berechnung des Schlüssels einfließen zu lassen. Dies hätte den Nachteil, dass man auch viele unterschiedliche Dateien mit dem gleichen Schlüssel verschlüsseln würde, da mehrere unterschiedliche Dateien mit einer gewissen Wahrscheinlichkeit fälschlicherweise die gleiche Prüfsumme generieren würden.

Ein weiteres Problem der Convergent Encryption ist, dass dieses Verfahren für den Confirmation of a File-Angriff anfällig ist. Das heißt, dass es einem Angreifer möglich ist, durch das Verschlüsseln eigener Dateien darauf zu schließen, was beispielsweise ein anderer Benutzer in seinem Repository gespeichert hat.

8.2. Keymanagement

8.2.1. Sicherung und Bindung der kryptographischen Schlüssel an eine Identität

Das asymmetrische Schlüsselpaar von IPFS ist standardmäßig in keiner Weise gesichert und muss daher besonders geschützt werden, da dieses Schlüsselpaar die Identität eines Individuums oder einer Institution darstellt. Beim Diebstahl des Schlüssels (Malware, Laptop-Verlust/Diebstahl) kann die jeweilige Identität nicht mehr als vertrauenswürdig betrachtet werden.

Die IPFS-Identität ist eng mit dem IPFS-Netzwerk verwoben. Da das Softwaredesign und auch die Sicherheitskomponenten sich aktuell in keinem finalen Stadium befinden, ist es für »brig« sinnvoll, eine eigene Schlüsselhierarchie umzusetzen, welche die Komponenten von IPFS schützt. So haben auch zukünftige Änderungen an IPFS selbst keinen oder nur wenig Einfluss auf das Sicherheitskonzept von »brig«.

Abb. 8.2 zeigt ein Konzept, bei welchem ein externes und vom Benutzer kontrolliertes Schlüsselpaar als Hauptschlüssel für die Absicherung des IPFS-Repository dient.

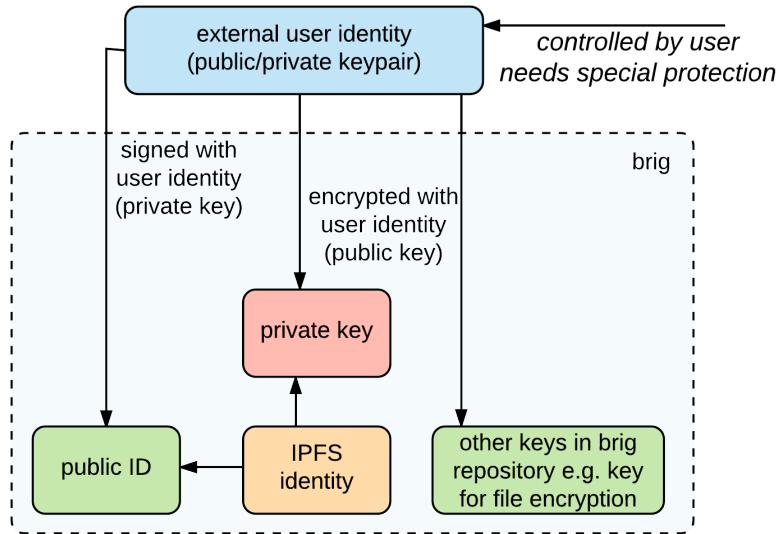


Abbildung 8.2.: Externes asymmetrisches Schlüsselpaar dient als Hauptschlüssel zur Sicherung der ungesicherten IPFS-Daten. Das Signieren der öffentlichen IPFS-ID ermöglicht eine zusätzliche Schicht der Authentifizierung.

Diese externe Identität muss dabei besonders vor Diebstahl geschützt werden, um einen Missbrauch zu vermeiden. Auf üblichen Endverbrauchergeräten ist der Passwortschutz dieses Schlüsselpaares ein absolutes Minimalkriterium. Weiterhin könnte »brig« den Ansatz fahren und die kryptographischen Schlüssel (Konfigurationsdatei von IPFS) selbst nur »on demand« im Speicher beispielsweise über ein VFS (Virtual Filesystem)¹ IPFS bereitstellen. Abb. 8.3 zeigt ein Konzept bei welchem »brig« über einen VFS-Adapter die Konfigurationsdateien und somit auch die kryptographischen Schlüssel IPFS bereitstellt.

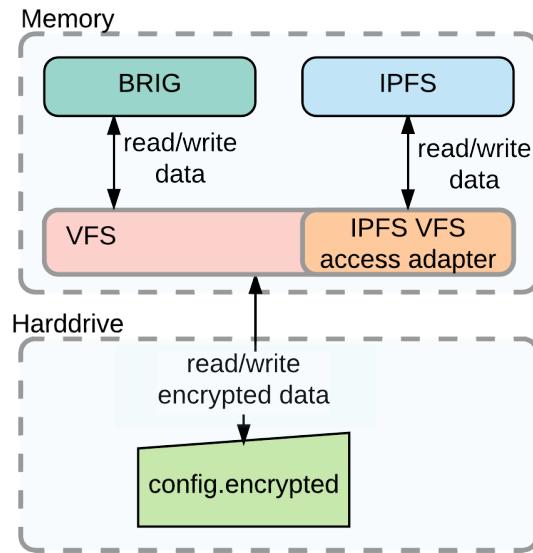


Abbildung 8.3.: »brig« stellt mittels VFS eine Zugriffsschnittstelle für IPFS dar.

¹Virtual file system: https://en.wikipedia.org/w/index.php?title=Virtual_file_system&oldid=758011055

Dabei wird beim Starten des »brig«-Daemon die verschlüsselte Datei im Arbeitsspeicher entschlüsselt und anschließend IPFS über einen Zugriffsadapter bereitgestellt. Dabei wird der komplette Zugriff über das VFS von »brig« verwaltet.

8.2.2. GnuPG als Basis für »externe Identität«

8.2.2.1. Einleitung

Für die Erstellung einer externen Identität (Abschnitt 8.2.1) kann beispielsweise GnuPG verwendet werden. GnuPG ist eine freie Implementierung des OpenPGP-Standards (RFC4880²). Die Implementierung ist heutzutage auf den gängigen Plattformen (Windows, MacOS, Linux, BSD) vorhanden. Die Implementierung für Windows (Gpg4win³) wurde vom Bundesamt für Sicherheit in der Informationstechnik in Auftrag gegeben. Neben dem Einsatz der sicheren E-Mail-Kommunikation, wird GnuPG heute unter vielen unixoiden Betriebssystemen zur vertrauenswürdigen Paketverwaltung verwendet. Distributionen wie beispielsweise Debian⁴, OpenSuse⁵, Arch Linux⁶ und weitere verwenden GnuPG zum Signieren von Paketen.

8.2.2.2. Grundlagen

Das theoretische Prinzip der asymmetrischen Verschlüsselung ist unter Abschnitt 4.2.2 anschaulich erläutert. In der Praxis ergeben sich jedoch nennenswerte Unterschiede, welche direkten Einfluss auf die Sicherheit des Verfahrens haben können.

Von GnuPG werden aktuell die folgenden drei Versionen gepflegt:

- ▶ GnuPG modern 2.1 – neuer Entwicklungszweig mit erweiterten Funktionalitäten und neuen kryptographischen Algorithmen beispielsweise für elliptische Kurven.
- ▶ GnuPG stable 2.0 – modularisierte Version von GnuPG (Support bis 2017-12-31).
- ▶ GnuPG classic – obsolete nicht modulare Version, diese wird aus Kompatibilitätsgründen zu alten Systemen gepflegt.

Für die hier vorgestellten Konzepte wird GnuPG modern in der version 2.1 als Kommandozeilen-Werkzeug verwendet, dieses ist unter Linux über die Kommandozeile mit gpg/gpg2 verwendbar:

```
$ gpg2 --version
gpg (GnuPG) 2.1.16
libgcrypt 1.7.3
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <https://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
```

²RFC4880: <https://www.ietf.org/rfc/rfc4880.txt>

³Gpg4win: <https://de.wikipedia.org/w/index.php?title=Gpg4win&oldid=159789331>

⁴Archive Signing Keys: <https://ftp-master.debian.org/keys.html>

⁵RPM – der Paket-Manager:

http://www.mpiiks-dresden.mpg.de/~mueller/docs/suse10.3/opensuse-manual_de/manual/cha.rpm.html

⁶Pacman/Package Signing: https://wiki.archlinux.org/index.php/Pacman/Package_signing

```
Home: /home/qitta/.gnupg
```

Supported algorithms:

Pubkey: RSA, ELG, DSA, ECDH, ECDSA, EDDSA

Cipher: IDEA, 3DES, CAST5, BLOWFISH, AES, AES192, AES256, TWOFISH, CAMELLIA128, CAMELLIA192, CAMELLIA256

Hash: SHA1, RIPEMD160, SHA256, SHA384, SHA512, SHA224

Compression: Uncompressed, ZIP, ZLIB, BZIP2

Für den GnuPG–Neuling gibt es verschiedene Frontends⁷ und Anwendungen, welche als GnuPG–Aufsatz verwendet werden können.

Beim Erstellen eines Schlüsselpaares wird bei GnuPG standardmäßig ein Hauptschlüssel- und ein Unterschlüsselpaar angelegt. Dies hat einerseits historische Gründe (aufgrund von Patenten konnten frühere Versionen von GnuPG kein RSA/RSA Schlüsselpaar zum Signieren und Ver- und Entschlüsseln anlegen, es wurde standardmäßig ein DSA Schlüssel zum Signieren und ein ElGamal Schlüssel zum Ver- und Entschlüsseln angelegt), andererseits ermöglicht es GnuPG, Schlüssel mit unterschiedlichem Schutzbedarf anders zu behandeln.

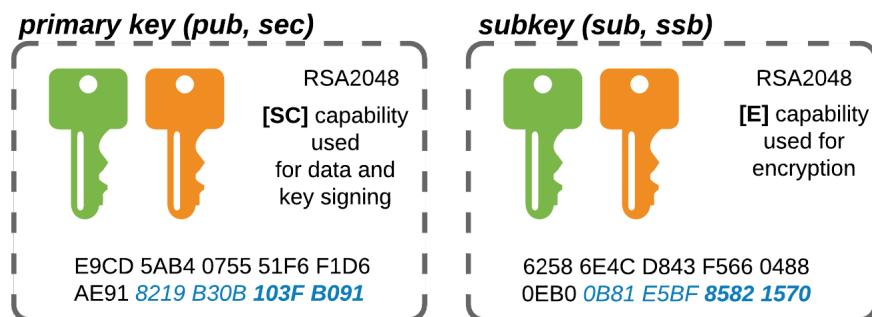


Abbildung 8.4.: GnuPG-Schlüsselpaar RSA/RSA bestehend aus einem Haupt- und Unterschlüsselpaar. Beide Schlüssel haben unterschiedliche Fähigkeiten und bestehen jeweils aus einem öffentlichen und einem privaten Schlüssel. In Hexadezimal ist jeweils der Fingerprint eines Schlüssels dargestellt. Der Fingerprint ist 20 Bytes groß. Die »Long-Key-ID« entspricht den letzten 8 Bytes, die »Short-Key-ID« entspricht den letzten 4 Bytes.

Der Schlüsselbund besteht bei Anlage eines neuen Schlüssels mit `gpg2 --gen-key` aus einem Hauptschlüssel- und einem Unterschlüsselpaar. Beide Schlüsselpaare bestehen jeweils aus einem öffentlichen und einem privaten Schlüssel (siehe Abb. 8.4).

Der Hauptschlüssel dient standardmäßig zum Signieren von Daten und Zertifizieren von Schlüsseln. Der Unterschlüssel wird für das Verschlüsseln von Daten erstellt⁸.

Der folgende Kommandozeilenauszug zeigt beispielhaft einen mit dem oben genannten Befehl angelegten Schlüssel, welcher mit den Daten der Abbildung übereinstimmt.

⁷GnuPG-Frontends: https://www.gnupg.org/related_software/frontends.html

⁸Debian Wiki Subkeys: <https://wiki.debian.org/Subkeys>

```
$ gpg2 --list-keys --fingerprint --fingerprint christoph@nullcat.de
pub    rsa2048 2013-02-09 [SC] [expires: 2017-01-31]
      E9CD 5AB4 0755 51F6 F1D6 AE91 8219 B30B 103F B091
uid          [ultimate] Christoph Piechula <christoph@nullcat.de>
sub    rsa2048 2013-02-09 [E] [expires: 2017-01-31]
      6258 6E4C D843 F566 0488 0EB0 0B81 E5BF 8582 1570
```

Hier ist in der ersten Zeile der öffentliche Teil des Hauptschlüssels (pub = public key) zu sehen. Es handelt sich um einen RSA-Schlüssel mit 2048 Bit. Im unteren Bereich ist der Unterschlüssel (sub = public subkey) zu sehen, hierbei handelt es sich ebenso wie beim Hauptschlüssel um einen RSA-Schlüssel mit 2048 Bit. Die Fähigkeit eines Schlüssels wird durch die Flags in eckigen Klammern angezeigt (Hauptschlüssel [SC], Unterschlüssel [E]).

Schlüssel können unter GnuPG folgende Fähigkeiten besitzen (vgl. `man gpg2`, Unterabschnitt `show--usage`):

- ▶ C (certification): Dieser Schlüssel ist zum Signieren von anderen/neuen Schlüsseln fähig.
- ▶ S (signing): Dieser Schlüssel ist zum Signieren von Daten fähig.
- ▶ E (encryption): Dieser Schlüssel ist zum Ver- und Entschlüsseln fähig.
- ▶ A (authentication): Dieser Schlüssel ist zum Authentifizieren fähig.

Einen besonderen Schutzbedarf hat an dieser Stelle der Hauptschlüssel, da dieser in der Lage ist, neu erstellte Schlüssel und Unterschlüssel zu signieren ([C]). Fällt einem Angreifer dieser Schlüssel in die Hände, so wäre er in der Lage im Namen des Benutzers neue Schlüssel zu erstellen und Schlüssel von anderen Teilnehmern zu signieren.

8.2.2.3. Offline Hauptschlüssel

Die privaten Schlüssel sind bei GnuPG mit einer Passphrase geschützt. Zusätzlich bietet GnuPG für den Schutz des privaten Hauptschlüssels eine Funktionalität namens Offline-Hauptschlüssel⁹ (Offline Master Key). Diese Funktionalität ermöglicht dem Benutzer den privaten Teil des Hauptschlüssels zu exportieren und beispielsweise auf einem sicheren externen Datenträger zu speichern.

Dieser Schlüssel wird zum Zertifizieren/Signieren anderer Schlüssel verwendet und wird nicht für den täglichen Gebrauch benötigt. Für die sichere externe Speicherung kann beispielsweise ein Verfahren wie Shamir's Secret Sharing (vgl. [21], S. 337 f. und [32]) verwendet werden. Bei diesem Verfahren wird ein Geheimnis auf mehrere Instanzen aufgeteilt, zur Rekonstruktion des Geheimnisses ist jedoch nur eine gewisse Teilmenge nötig. Das Shamir's Secret Sharing-Verfahren wird von libgfshare¹⁰ implementiert. Diese Bibliothek bietet mit den beiden Kommandozeilenwerkzeugen `gfsplit` und `gfcombine` eine einfache Möglichkeit, den privaten Schlüssel auf mehrere Instanzen aufzuteilen. Im Standardfall wird der Schlüssel auf fünf Dateien aufgeteilt, von welchen mindestens drei benötigt werden, um den Schlüssel wieder zu rekonstruieren.

⁹Offline-Hauptschlüssel:

https://de.wikipedia.org/w/index.php?title=GNU_Privacy_Guard&oldid=159842195#Offline-Hauptschl.C3.BCssel

¹⁰GitHub libgfshare: <https://github.com/jcushman/libgfshare>

Die folgenden Kommandozeilenauszüge zeigen die Funktionsweise im Erfolgs- und Fehlerfall. Gezeigt wird die Prüfsumme des zu sichernden privaten Schlüssels und die Aufteilung mittels gfsplit:

```
$ sha256sum private.key
d90dc1dbb96387ef25995ada677c59f909a9249eafcb32fc7a4f5eae91c82b42  private.key

$ gfsplit private.key && ls
private.key  private.key.022  private.key.064  \
private.key.076  private.key.153  private.key.250
```

Das Zusammensetzen mit genügend (recovered1.key) und ungenügend (recovered2.key) vielen Teileheimnissen und die anschließende Validierung über die Prüfsumme:

```
$ gfcombine -o recovered1.key private.key.022 private.key.064
$ gfcombine -o recovered2.key private.key.022 private.key.064 private.key.153
$ sha256sum recovered* private.key
6ea5094f26ae1b02067c5b96755313650da78ded64496634c3cc49777df79de6  recovered1.key
d90dc1dbb96387ef25995ada677c59f909a9249eafcb32fc7a4f5eae91c82b42  recovered2.key
d90dc1dbb96387ef25995ada677c59f909a9249eafcb32fc7a4f5eae91c82b42  private.key
```

Eine Möglichkeit, den privaten Schlüssel analog zu sichern, bietet die Applikation Paperkey¹¹. Paperkey extrahiert nur die benötigten Daten zur Sicherung des privaten Schlüssels und bringt diese in eine gut druckbare Form.

Die Offline-Hauptschlüssel-Funktionalität ist eine zusätzliche Funktionalität von GnuPG und *nicht* Teil des RFC4880-Standards.

8.2.2.4. Unterschlüssel und Key Separation

Eine weitere Maßnahme und Best Practise im Bereich der Kryptographie ist die sogenannte Key Separation (vgl. [21], S. 359 ff.). Das heißt, dass kryptographische Schlüssel an einen bestimmten Zweck gebunden sein sollen. Einen Schlüssel für mehrere verschiedene Zwecke zu verwenden, ist sicherheitstechnisch bedenklich.

Obwohl es mit GnuPG möglich ist, ein Schlüsselpaar zu erstellen, welches zum Signieren, Zertifizieren und Ver- und Entschlüsseln verwendet werden kann (Flags [SCE]), ist dies aus Gründen der Sicherheit nicht empfehlenswert. Beim Anlegen eines neuen Schlüssels wird standardmäßig bereits ein Schlüsselpaar bestehend aus Haupt- und Unterschlüssel angelegt. In der Standardkonfiguration würde der Hauptschlüssel aufgrund seiner Flags neben dem Signieren von Schlüsseln [C] auch für das Signieren von Daten [S] Verwendung finden. Dieser Umstand würde auch verhindern, dass der Benutzer beim Einsatz der Offline-Hauptschlüssel-Funktionalität Daten signieren kann.

Die Umsetzung einer Key Separation kann mit GnuPG beim Anlegen (`gpg2 --full-gen-key --expert`) oder nachträglich (`gpg2 --edit-key <keyid>`) realisiert werden.

¹¹Paperkey Homepage: <http://www.jabberwocky.com/software/paperkey/>

Abb. 8.5 zeigt die Möglichkeit der Anlage von Unterschlüsseln für den regulären Gebrauch. Eine weitere Empfehlung an dieser Stelle wäre es, die Unterschlüssel zusätzlich auf eine Smartcard auszulagern (siehe Abschnitt 8.4.9.3).

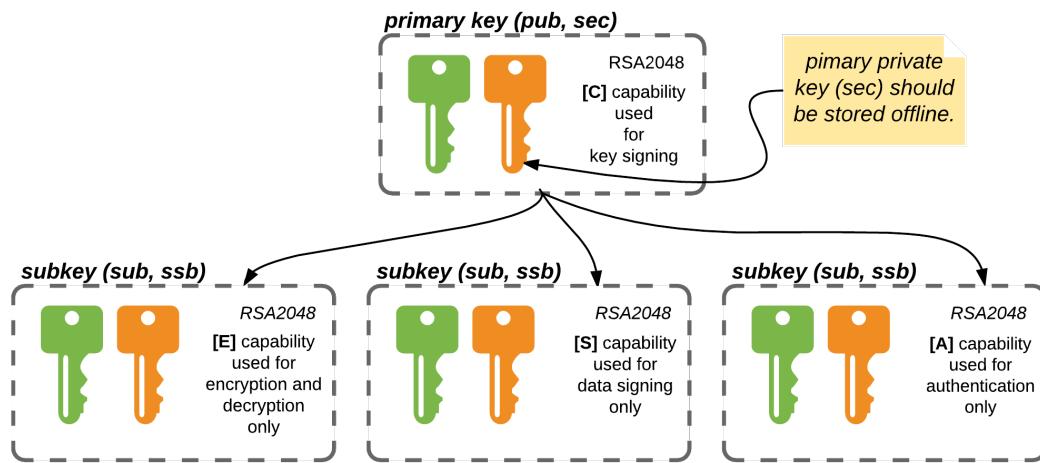


Abbildung 8.5.: GPG-Schlüsselbund mit Unterschlüsseln für den regulären Einsatz. Jeder Unterschlüssel ist an einen bestimmten Einsatzzweck gebunden.

8.2.2.5. GPG-Agent

GnuPG hat einen gpg-agent. Dieser übernimmt das Management der vom Benutzer eingegebenen Passphrasen und kann diese für eine gewisse Zeit speichern und bei Bedarf abfragen. Weiterhin bietet der Agent seit Version 2.0.x die Möglichkeit, auf Smartcards zuzugreifen. Zusätzlich ist es seit Version 2 möglich, GPG-Schlüssel für die SSH-Authentifizierung zu verwenden.

8.2.2.6. Weiteres

Bei der Evaluation einer sinnvollen Schlüsselverwaltung ist aufgefallen, dass die Passwortabfrage beim Generieren des GnuPG-Schlüssels eine für den Benutzer fragliche Rückmeldung bezüglich der Passwortqualität liefert.

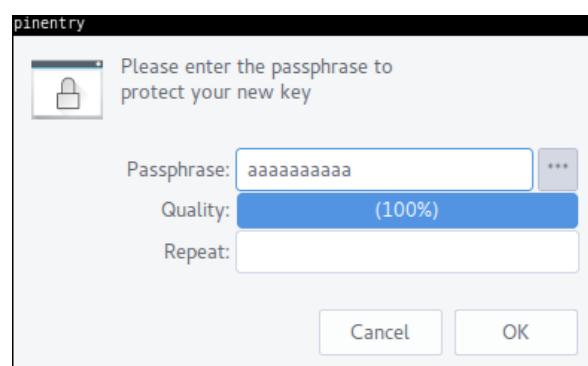


Abbildung 8.6.: Fragwürdige Entropieschätzung im GnuPG-Pinentry Dialog.

Hier wird in der aktuellen GnuPG-Version 2.1.16 unter Arch Linux anscheinend jedes Passwort mit einer Zeichenlänge von 10 Zeichen als *Qualität 100%* definiert (siehe beispielhaft Abb. 8.6). Dieser Ansatz ist bei einer Sicherheitssoftware wie GnuPG, welche wichtige kryptographische Schlüssel schützen muss fragwürdig, da ein Passwort dieser Komplexität als definitiv unsicher angesehen werden sollte (siehe auch Abschnitt 7.6). Weiterhin vermittelt dieser Dialog dem Benutzer ein Gefühl von falscher Sicherheit und erzieht ihn zu schlechten Gewohnheiten.

8.3. Authentifizierungskonzept

8.3.1. Authentifizierungskonzept mit IPFS-Bordmitteln

Unter Abschnitt 7.5 wurde die aktuelle Situation evaluiert. Zum aktuellen Zeitpunkt hat »brig« keinen Authentifizierungsmechanismus. Die kommunizierenden Parteien müssen ihre »brig«-Fingerabdrücke gegenseitig validieren und auf dieser Basis manuell ihrer Liste aus Synchronisationspartnern hinzufügen. Neben der Möglichkeit, den Fingerabdruck über einen Seitenkanal (Telefonat, E-Mail) auszutauschen, sollen nun benutzerfreundlichere Konzepte vorgestellt werden.

8.3.1.1. Manuelle Authentifizierung über QR-Code

Eine sinnvolle Erweiterung an dieser Stelle wäre die Einführung eines QR-Codes welcher die Identität eines Synchronisationspartners eindeutig bestimmt. Auf Visitenkarten gedruckte QR-Codes lassen den Benutzer beispielsweise seinen Synchronisationspartner mit wenig Aufwand über eine Smartphone-Applikation verifizieren. Bei Anwendung einer externen Identität welche für das Signieren der »brig«-ID verwendet werden kann – wie unter Abschnitt 8.2 vorgeschlagen – würde der Datensatz zur Verifikation wie folgt aussehen

- ▶ IPFS-ID: QmbR6tDXRCgpRwWZhGG3qLfJMKrLcrgk2qv5BW7HNhCkpL
- ▶ GPG-Key-ID (16 Byte): D3B2 790F BAC0 7EAC, falls keine GPG-Key-ID existiert: 0000 0000 0000 0000

und könnte beispielsweise in folgender Form realisiert werden:

- ▶ QmbR6tDXRCgpRwWZhGG3qLfJMKrLcrgk2qv5BW7HNhCkpL | D3B2790FBAC07EAC

Abb. 8.7 zeigt den definierten Datensatz als QR-Code (Quelltext siehe Anhang L).



Abbildung 8.7.: »brig« QR-Code um einen Synchronisationspartner auf einfache Art und Weise zu authentifizieren.

Es sollte bei der GPG-Key-ID darauf geachtet werden, dass hier mindestens 16 Byte des Fingerprints verwendet werden, da die 8 Byte Repräsentation Angriffsfläche¹² bietet.

8.3.1.2. Authentifizierung über Frage–Antwort–Dialog

Da IPFS bereits ein Public/Private–Schlüsselpaar mitbringt, würde sich im einfachsten Falle nach dem ersten Verbindungsauflauf die Möglichkeit bieten, seinen Synchronisationspartner anhand eines gemeinsamen Geheimnisses oder anhand eines Frage–Antwort–Dialogs zu verifizieren. Abb. 8.8 zeigt den Ablauf einer Authentifizierung des Synchronisationspartners mittels Frage–Antwort–Dialog, welcher in folgenden Schritten abläuft:

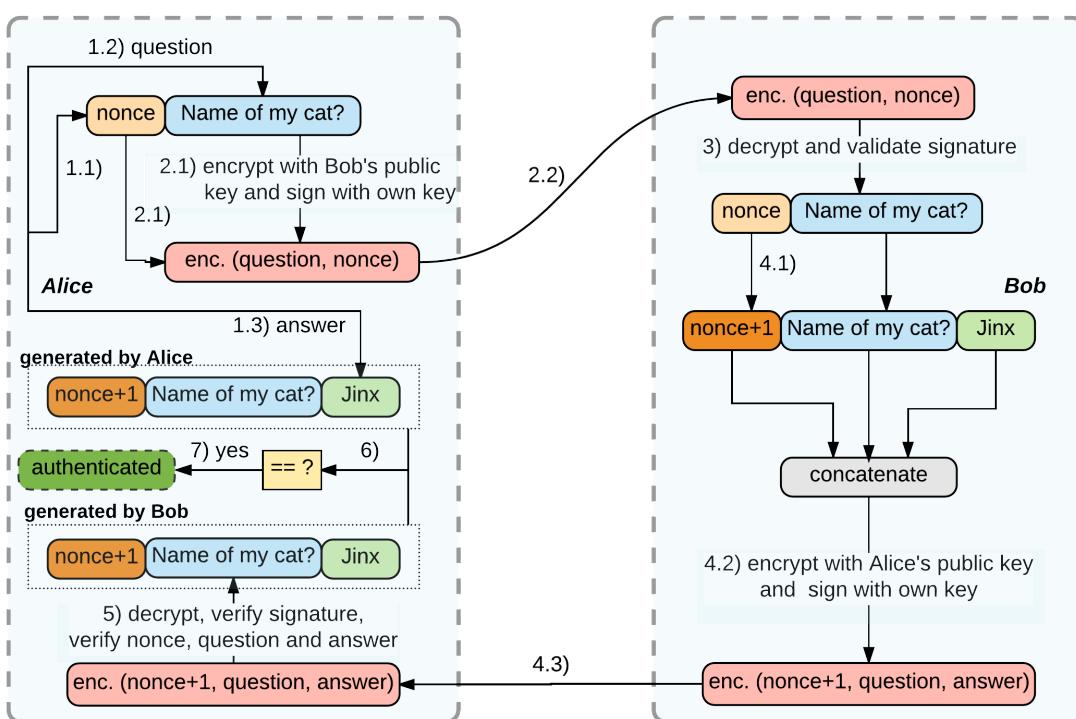


Abbildung 8.8.: Frage–Antwort–Authentifizierung. Alice stellt Bob eine persönliche Frage auf die Bob die Antwort weiß.

1. Alice generiert eine zufällige Nonce, Frage und Antwort.
2. Alice verschlüsselt Nonce + Antwort, signiert diese mit ihrem privaten Schlüssel und schickt diese an Bob.
3. Bob prüft die Signatur, ist diese ungültig, wird abgebrochen, bei Gültigkeit entschlüsselt Bob die Nachricht.
4. Bob inkrementiert die Nonce von Alice um eins und erstellt ein Antwortpaket bestehend aus Nonce, Frage und Antwort und schickt dieses verschlüsselt an Alice.
5. Alice prüft die Signatur, ist diese ungültig wird abgebrochen, bei Gültigkeit entschlüsselt Alice die Nachricht.

¹²Evil32-Schwachstelle: <https://evil32.com/>

6. Alice prüft ob Nonce um eins inkrementiert wurde, ob die Frage zur gestellten Frage passt und ob die Antwort die erwartete Antwort ist.
7. Stimmen alle drei Parameter überein, dann wird Bob von Alice als vertrauenswürdig eingestuft und Alice kann somit Bob's ID als vertrauenswürdig hinterlegen.

Um Alice gegenüber Bob zu verifizieren, muss das Protokoll von Bob aus initialisiert werden.

Die Anforderungen des Protokolls richten sich hierbei nach den Prinzipien (vgl. [21], S. 295 ff.):

- ▶ Nachrichtenauthentifizierung, durch Signatur bereitgestellt.
- ▶ Gültigkeit (engl. freshness), durch Nonce bereitgestellt.
- ▶ Bezug zur korrekten Anfrage. Frage wird in der Antwort mitgesendet.

Weiterhin wäre diese Art der Authentifizierung auch unter Verwendung des Socialist Millionaire-Protokolls möglich, siehe folgender Abschnitt.

8.3.1.3. Authentifizierung über gemeinsames Geheimnis (Socialist Millionaire-Protokoll)

Eine weitere Möglichkeit der Authentifizierung des Synchronisationspartners wäre, wie beim OTR-Plugin des Pidgin-Messengers, das Teilen eines gemeinsamen Geheimnisses. Das Off-the-Record-Messaging-Protokoll verwendet hierbei das Socialist Millionaire-Protokoll. Das Protokoll erlaubt es Alice und Bob ein gemeinsam geglaubtes Geheimnis x (Alice) und y (Bob) auf Gleichheit zu prüfen ohne es austauschen zu müssen. Weiterhin ist das Protokoll nicht für einen Man-in-the-Middle-Angriff anfällig. Beim Off-the-Record-Messaging-Protokoll¹³ werden alle Operationen modulo einer bestimmten 1536 Bit Primzahl genommen, g_1 ist hier das erzeugende Element dieser Gruppe. Abb. 8.9 zeigt den grundlegenden Ablauf des Socialist Millionaire-Protokoll.

¹³Socialist Millionaire Protocol (SMP): <https://otr.cypherpunks.ca/Protocol-v3-4.0.0.html>

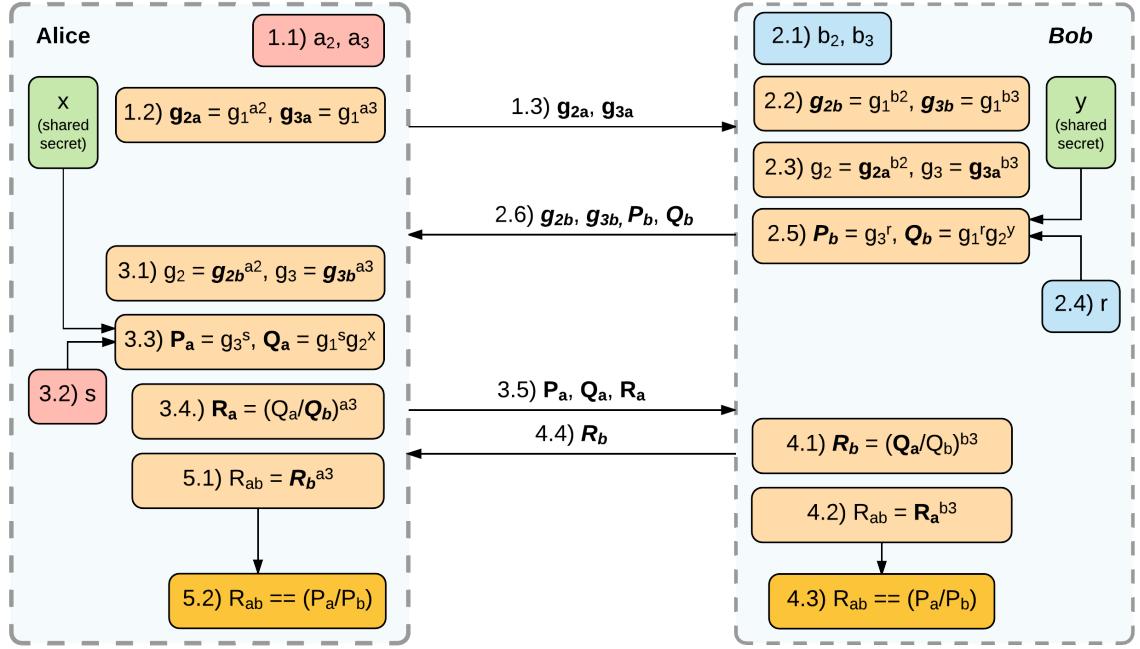


Abbildung 8.9.: Authentifizierung über ein gemeinsames Geheimnis unter Verwendung des Socialist Millionaire–Protokoll.

1. Alice generiert zwei zufällige Exponenten a_2 und a_3 . Anschließend berechnet sie g_{2a} und g_{3a} und sendet diese an Bob.
2. Bob generiert zwei zufällige Exponenten b_2 und b_3 . Anschließend berechnet Bob g_{2b} , g_{3b} , g_2 und g_3 . Bob wählt einen zufälligen Exponenten r . Bob berechnet P_b und Q_b (hier fließt das gemeinsame Geheimnis y von Bob ein) und sendet g_{2b} , g_{3b} , P_b und Q_b an Alice.
3. Alice berechnet g_2 , g_3 , wählt einen zufälligen Exponenten s , berechnet P_a , Q_a (hier fließt das gemeinsame Geheimnis x von Alice ein), R_a und sendet P_a , Q_a und R_a an Bob.
4. Bob berechnet R_b , R_{ab} und prüft ob $R_{ab} == (P_a/P_b)$. Anschließend sendet er R_b an Alice.
5. Alice berechnet R_{ab} und prüft ob $R_{ab} == (P_a/P_b)$.

Konnten Alice und Bob jeweils $R_{ab} = (P_a/P_b)$ als korrekt validieren, so haben sie sich gegenseitig authentifiziert. Für weitere Details zum Protokoll vgl. [33] und [34].

Für eine Implementierung in »brig« kann hier beispielsweise die Go-Standardsbibliothek¹⁴ oder andere Implementierungen¹⁵ als Referenz hergenommen werden.

¹⁴Go-Standardsbibliothek Off-The-Record-Protokoll: <https://godoc.org/golang.org/x/crypto/otr>

¹⁵Socialist Millionaire-Implementierung auf GitHub: <https://github.com/cowlicks/socialist-millionaire-go>

8.3.2. Authentifizierungskonzept auf Basis des Web of Trust

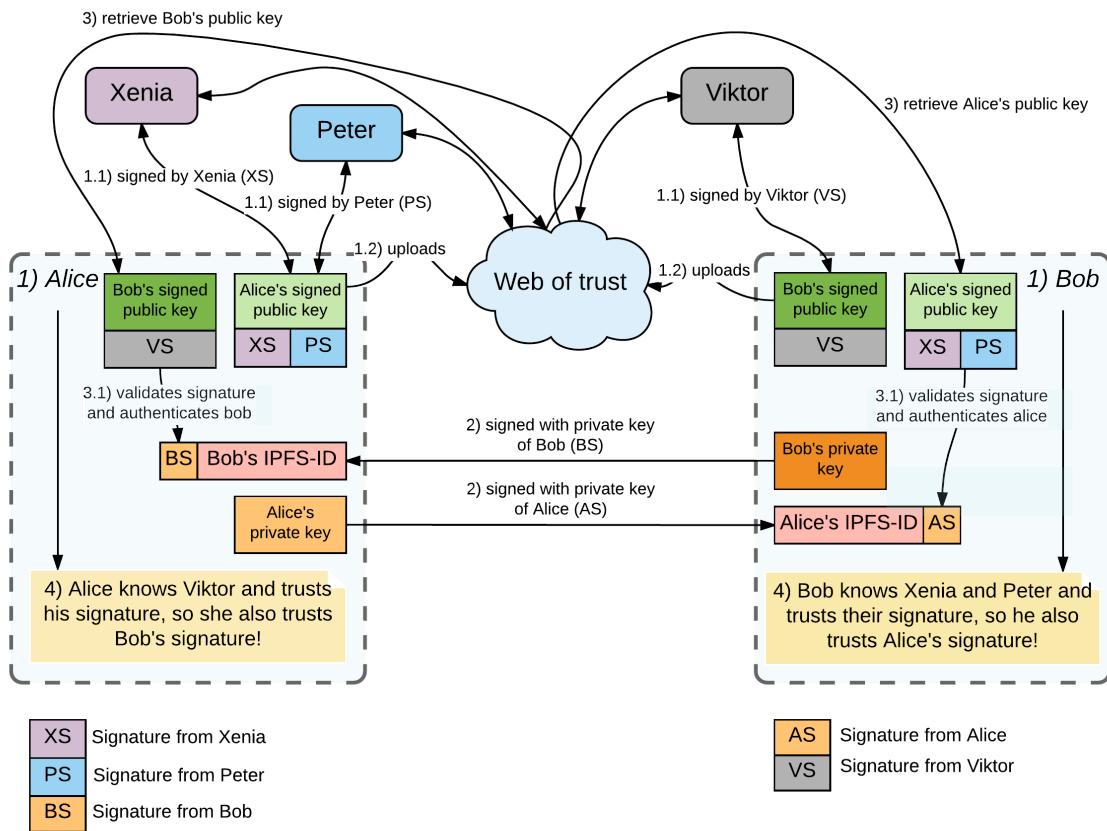


Abbildung 8.10.: Authentifizierung auf Basis des Web of Trust.

Eine weitere Möglichkeit einer Authentifizierung wäre beispielsweise auf Basis des Web of Trust denkbar. Dieses beschreibt einen typischen dezentralen PKI-Ansatz, welcher aktuell mittels der GnuPG-Software umgesetzt wird. Die IPFS-ID kann hierbei mit dem privaten Schlüssel signiert werden und über das Web of Trust-Overlay-Network von den jeweiligen Benutzern validiert werden. Abb. 8.10 stellt das Konzept grafisch dar.

Die Authentifizierung von Kommunikationspartnern ist für den Benutzer keine triviale Aufgabe. Die in Abb. 8.10 dargestellte Situation stellt jedoch eine ideale Sichtweise des Web of Trust-Vertrauensmodells dar. Das Vertrauen zwischen den verschiedenen Parteien des Web of Trust ist nicht generell übertragbar, da es lediglich auf Empfehlungen einzelner Individuen basiert (vgl. [35]). »A Probabilistic Trust Model for GnuPG« stellt eine interessante wahrscheinlichkeitstheoretische Erweiterung des klassischen Vertrauensmodells dar (vgl. [36]).

Kurze Erläuterung:

1. Alice und Bob sind Teilnehmer des Web of Trust, ihre öffentlichen Schlüssel sind von weiteren Personen (Freunden) signiert.
2. Alice und Bob signieren ihre IPFS-ID vor dem Austausch mit dem jeweiligen Synchronisations-

partner.

3. Alice und Bob beschaffen sich den öffentlichen Schlüssel des Synchronisationspartners aus dem Web of Trust, um damit die Signatur der IPFS-ID zu prüfen.
4. Da die öffentlichen Schlüssel der jeweiligen Parteien bereits von anderen vertrauenswürdigen Parteien unterschrieben sind, akzeptieren beide Synchronisationspartner die Signatur und somit die IPFS-ID.

Dieses Konzept ist um so vertrauenswürdiger, je mehr vertrauenswürdige Parteien einen öffentlichen Schlüssel unterschreiben. Durch zusätzliche Instanzen, wie beispielsweise die Zertifizierungsstelle CAcert¹⁶ und die c't Krypto-Kampagne¹⁷, kann das Vertrauen in die Identität einer Partei weiter erhöht werden.

Wissenschaftliche Untersuchungen haben weiterhin ergeben, dass ein Großteil der Web of Trust-Teilnehmer zum sogenannten Strong Set gehören. Diese Teilmenge repräsentiert Benutzer/Schlüssel welche durch gegenseitige Bestätigung vollständig miteinander verbunden sind. Projekte wie die c't Krypto-Kampagne oder auch das Debian-Projekt sollen hierzu einen deutlichen Beitrag geleistet haben (vgl. [37] und [38]).

8.4. Smartcards und RSA-Token als 2F-Authentifizierung

8.4.1. Allgemein

Wie bereits erwähnt, ist die Authentifizierung über ein Passwort oft der Schwachpunkt eines zu sichernden Systems. Ist das Passwort oder die Passwort-Richtlinien zu komplex, so neigen Benutzer oft dazu, die Passwörter aufzuschreiben. Ist die Komplexität beziehungsweise Entropie zu niedrig, so ist es mit modernen Methoden vergleichsweise einfach, das Passwort zu berechnen (siehe Abschnitt 5.2).

Ein weiterer Schwachpunkt, der oft ausgenutzt wird, ist die unsichere Speicherung von kryptographischen Schlüsseln. Passwörter sowie kryptographische Schlüssel können bei handelsüblichen Endanwendersystemen, wie beispielsweise PC oder Smartphone relativ einfach mitgeloggt beziehungsweise entwendet werden. Neben dem FreeBSD-Projekt, welches dem Diebstahl von kryptographischen Schlüsseln zum Opfer fiel, gibt es laut Berichten zunehmend Schadsoftware welche explizit für diesen Einsatzzweck konzipiert wurde (siehe Abschnitt 5.2).

Um hier die Sicherheit zu steigern, wird von Sicherheitsexperten oft zur Zwei-Faktor-Authentifizierung beziehungsweise zur hardwarebasierten Speicherung kryptographischer Schlüssel (persönliche Identität, RSA-Schlüsselpaar) geraten (vgl. [21], S. 350 ff.).

8.4.2. OpenPGP Smartcard

Für die Speicherung von kryptographischen Schlüsseln eignen sich beispielsweise Chipkarten, welche die Speicherung kryptographischer Schlüssel ermöglichen.

¹⁶CAcert: <https://de.wikipedia.org/w/index.php?title=CAcert&oldid=161320822>

¹⁷Krypto-Kampagne: <https://www.heise.de/security/dienste/Krypto-Kampagne-2111.html>

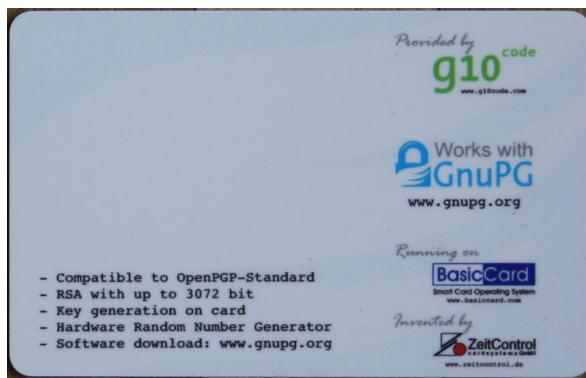


Abbildung 8.11.: Von g10 code vertriebene Smartcard für den Einsatz mit GnuPG¹⁸.

Abb. 8.11 zeigt die OpenPGP-Card Chipkarte¹⁹ von ZeitControl, welche über den Anbieter g10 code vertrieben wird. Der Anbieter der Smartcard ist gleichzeitig der Entwickler hinter dem GnuPG-Projekt. Auf der OpenPGP-Chipkarte Version 2.0 lassen sich drei RSA-Schlüssel (Signieren, Ver-/Entschlüsseln, Authentifizieren) mit jeweils 2048 Bit speichern. Der Vorteil bei der Smartcard ist, dass die kryptographischen Schlüssel die Karte selbst nie verlassen. Alle kryptographischen Operationen werden auf der Karte selbst durchgeführt. Dieser Ansatz schafft eine sichere Aufbewahrung der kryptographischen Schlüssel.

Die Problematik bei Smartcards ist jedoch, dass man zusätzlich ein Lesegerät benötigt. Dieser Umstand schränkt die Benutzung stark ein und ist deswegen weniger für den privaten Einsatzzweck geeignet.

8.4.3. Zwei-Faktor-Authentifizierung

Bei der Zwei-Faktor-Authentifizierung gibt es verschiedene Varianten, welche in der Regel ein Passwort mit einem weiteren Element wie einer Bankkarte oder einem Hardware-Token wie beispielsweise der RSA SecureID²⁰ verknüpfen.

Ein Problem hierbei ist wieder die Umsetzung im privaten Bereich.

Eine relativ »neue« Möglichkeit bieten die Hardware-Tokens von Yubico²¹ (siehe Abb. 8.12) und Nitrokey²². Diese Hardware-Tokens haben zudem den Vorteil, dass sie die Funktionalität einer Smartcard und eines Hardware-Tokens für Zwei-Faktor-Authentifizierung vereinen.

¹⁸Bildquelle: <https://g10code.com/graphics/newcard-b.jpg>

¹⁹Homepage g10 code: <https://g10code.com/p-card.html>

²⁰RSA-SecureID: <https://de.wikipedia.org/w/index.php?title=SecurID&oldid=159759136>

²¹Yubico: <https://www.yubico.com>

²²Nitrokey: <https://www.nitrokey.com/>



Abbildung 8.12.: YubiKey NEO²³ mit USB-Kontaktschnittstelle und Push-Button, welcher bei Berührung reagiert.

Das Besondere bei diesen Hardware-Komponenten ist, dass sie sich über die USB-Schnittstelle als HID (Human-Interface-Device²⁴) ausgeben und somit keine weitere Zusatzhardware wie beispielsweise ein Lesegerät benötigt wird. Weiterhin müssen keine zusätzlichen Treiber, beispielsweise für ein Lesegerät, installiert werden.

Bei beiden Herstellern gibt es die Hardware-Token in verschiedenen Ausführungen. Bekannte Institutionen, welche den YubiKey verwenden sind beispielsweise die Universität von Auckland²⁵, das CERN²⁶ oder auch das Massachusetts Institute of Technology²⁷.

Für die Entwicklung von »brig« wurden YubiKey NEO-Hardware-Token – aufgrund der umfangreichen Programmier-API – des Herstellers Yubico beschafft. Alle weiteren Ausführungen und Demonstrationen beziehen sich auf dieses Modell.

8.4.4. YubiKey NEO Einleitung

Der YubiKey NEO hat folgende Funktionalitäten beziehungsweise Eigenschaften:

- ▶ Yubico OTP, One-Time-Password-Verfahren des Herstellers. Standardmäßig kann jeder Yubi-Key gegen den YubiCloud-Authentifizierungsdienst mittels One-Time-Password authentifiziert werden.
- ▶ OATH-Kompatibilität (HMAC-Based-OTP- und Time-Based-OTP-Verfahren, für weitere Details vgl. [39], S. 904 f.)
- ▶ Challenge-Response-Verfahren (HMAC-SHA1, Yubico OTP)
- ▶ FIDO U2F (Universal Second Factor)
- ▶ Statische Passwörter

²³Bildquelle: <https://www.yubico.com/wp-content/uploads/2015/04/YubiKey-NEO-1000-2016-444x444.png>

²⁴Human Interface Device: https://en.wikipedia.org/w/index.php?title=Human_interface_device&oldid=746909537

²⁵Auckland University YubiKey-Benutzeranweisung:

<https://www.auckland.ac.nz/en/about/the-university/how-university-works/policy-and-administration/computing/use/twostepverification.html>

²⁶CERN YubiKey-Benutzeranweisung: <https://security.web.cern.ch/security/recommendations/en/2FA.shtml>

²⁷Massachusetts Institute of Technology YubiKey-Benutzeranweisung:

<http://kb.mit.edu/confluence/pages/viewpage.action?pageId=154177462>

Smartcard-Funktionalität:

- ▶ PIV (Personal Identity Verification) Standard²⁸
- ▶ OpenPGP-Smartcard-Standard

Weitere Eigenschaften sind im Datenblatt²⁹ des YubiKey NEO zu finden.

Der YubiKey NEO bietet mit zwei Konfigurationsslots (siehe GUI-Screenshot Abb. 8.13) die Möglichkeit, mehrere Verfahren gleichzeitig nutzen zu können. Eine beispielhafte Konfiguration wäre den ersten Konfigurationsslot mit einem statischen Passwort und den zweiten mit einem One-Time-Password zu belegen. Slot 1 lässt sich mit einem kurzen Drücken (0,3–1,5 Sekunden) ansprechen, Slot 2 mit einem längeren Drücken (2,5–5 Sekunden). Für weitere Details siehe »The YubiKey Manual«³⁰.

Die grundlegende Konfiguration des YubiKey ist mit dem YubiKey Personalization Tool möglich.

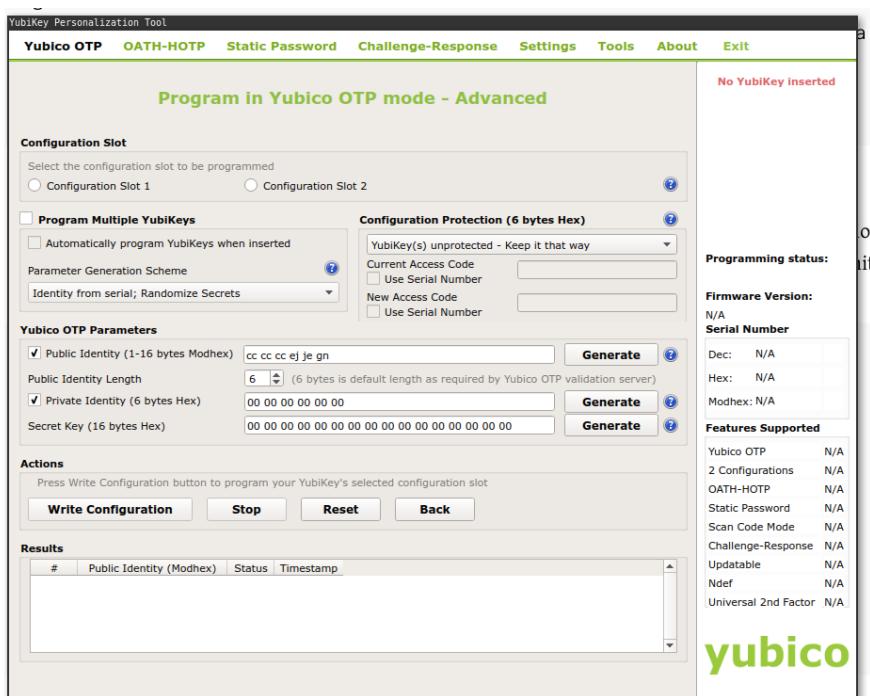


Abbildung 8.13.: GUI des YubiKey Personalization Tool. Das Konfigurationswerkzeug ist eine QT-Anwendung, diese wird von den gängigen Betriebssystemen (Linux, MacOs, Windows) unterstützt.

8.4.5. Yubico OTP Zwei-Faktor-Authentifizierung

Der YubiKey ist im Auslieferungszustand so konfiguriert, dass er sich gegenüber der YubiCloud mittels Yubico OTP authentifizieren kann. Abb. 8.15 zeigt den Ablauf des Authentifizierungsprozesses. Das One-Time-Password ist insgesamt 44 Zeichen lang und besteht dabei aus zwei Teilkomponenten. Die ersten 12 Zeichen repräsentieren eine statische öffentliche ID mit welcher sich die YubiKey Hardware identifizieren lässt. Die verbleibenden Zeichen repräsentieren den dynamisch generierten Teil des One-Time-Password.

²⁸PIV-Standard: <http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.201-2.pdf>

²⁹YubiKey NEO: https://www.yubico.com/wp-content/uploads/2016/02/Yubico_YubiKeyNEO_ProductSheet.pdf

³⁰The YubiKey Manual https://www.yubico.com/wp-content/uploads/2015/03/YubiKeyManual_v3.4.pdf

Abb. 8.14 zeigt ein vollständiges valides One-Time-Password.

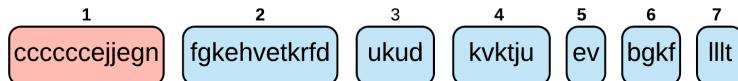


Abbildung 8.14.: Yubico OTP Aufbau

1. Öffentliche ID (6 Bytes), statischer Teil des One-Time-Password.
2. Geheime ID (6 Bytes), dynamisch erzeugt, wird von der validierenden Instanz geprüft.
3. Interner Zähler (2 Bytes), ein nichtflüchtiger Zähler, der beim »power up/reset« um eins inkrementiert wird.
4. Zeitstempel (3 Bytes)
5. Sitzungszähler (1 Byte), beim »power up« wird dieser Zähler auf null gesetzt und bei jedem One-Time-Password inkrementiert.
6. Zufällig generierteNonce (2 Bytes), wird vom internen Random-Number-Generator erstellt, um weitere Entropie hinzuzufügen.
7. CRC16 Prüfsumme (2 Bytes).

Der dynamische Teil besteht aus mehreren verschiedenen Einzelkomponenten, die eine zufällige Nonce, Sitzungsschlüssel und Zähler beinhalten. Weiterhin fließt ein AES-Schlüssel in die Generierung des One-Time-Password ein. Es ist für einen Angreifer somit nicht möglich, die eigentlichen Daten auszuwerten.

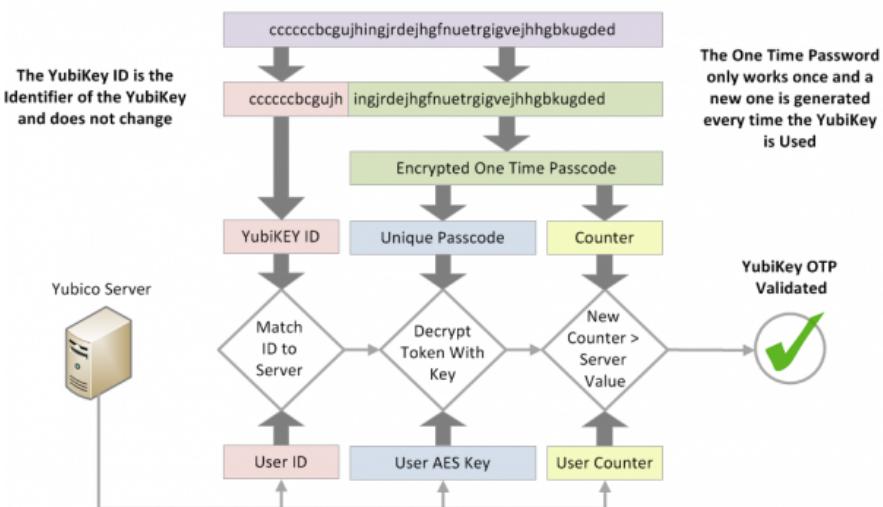


Abbildung 8.15.: Yubico OTP-Authentifizierungsprozess an der YubiCloud³¹.

Das One-Time-Password ist nur einmal gültig. Zur Validierung werden am Server Sitzung und Zähler jeweils mit den zuvor gespeicherten Daten überprüft. Stimmen diese nicht – da beispielsweise der aktuelle Zähler kleiner ist, als der zuletzt gespeicherte – so wird das One-Time-Password nicht akzeptiert.

³¹Bildquelle OTPs Explained: https://developers.yubico.com/OTP/OTPs_Explained.html

Für das Testen der korrekten Funktionalität stellt Yubico eine Demoseite für OTP³² und U2F³³ bereit. Über diese lässt sich ein One-Time-Password an die YubiCloud schicken und somit die korrekte Funktionsweise eines YubiKey validieren. Abb. 8.16 zeigt die Authentifizierungsantwort der YubiCloud.

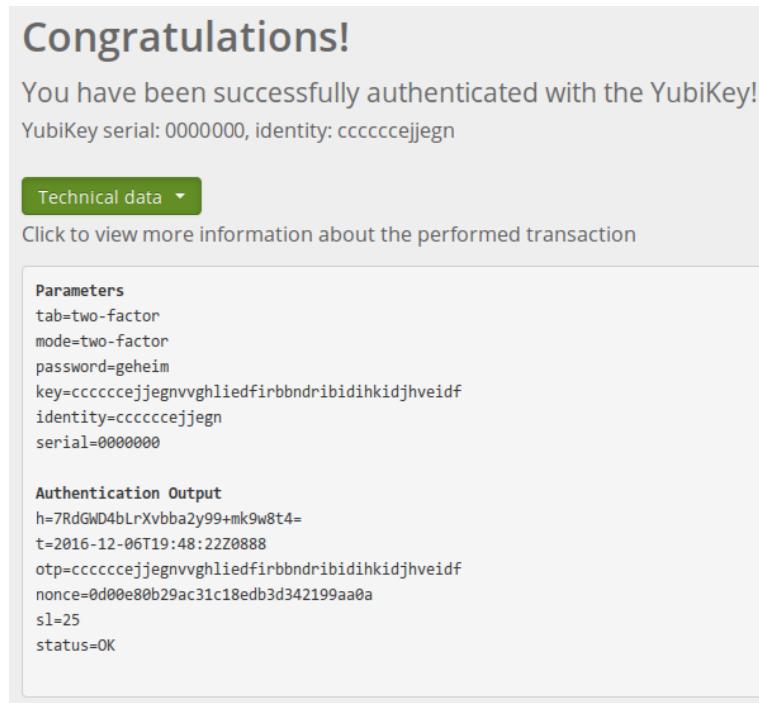


Abbildung 8.16.: YubiCloud Response bei Zwei-Faktor-Authentifizierung. Seriennummer des YubiKeys wurde retuschiert.

Die Demoseite bietet hier neben dem einfachen Authentifizierungstest, bei welchem nur das One-Time-Password validiert wird, auch noch die Möglichkeit, einen einfachen Zwei-Faktor-Authentifizierungstest und einen Zwei-Faktor-Authentifizierungstest mit Passwort durchzuführen.

8.4.6. Konzept zur Zwei-Faktor-Authentifizierung von »brig« mit der YubiCloud

Für die Proof-of-Concept-Implementierung der Zwei-Faktor-Authentifizierung wird die yubigo-Bibliothek³⁴ verwendet.

Für den Einsatz unter »brig« wird ein API-Key und ein Secret-Key von Yubico benötigt. Dieser wird für die Authentifizierung der Bibliothek gegenüber dem Yubico-Dienst verwendet. Die Beantragung erfolgt online³⁵ und erfordert einen YubiKey. Die minimale Implementierung in Anhang M zeigt einen voll funktionsfähigen Authentifizierungs-Client, welcher einen YubiKey am YubiCloud-Dienst authentifiziert.

³²Yubico OTP-Demopage: <https://demo.yubico.com>

³³Yubico U2F-Demopage: <https://demo.yubico.com/u2f>

³⁴Yubigo Dokumentation: <https://godoc.org/github.com/GeertJohan/yubigo>

³⁵Yubico API-Key beantragen: <https://upgrade.yubico.com/getapikey/>

Abb. 8.17 zeigt schematisch den Zwei-Faktor-Authentifizierungs-Vorgang mit einem YubiKey über die YubiCloud.

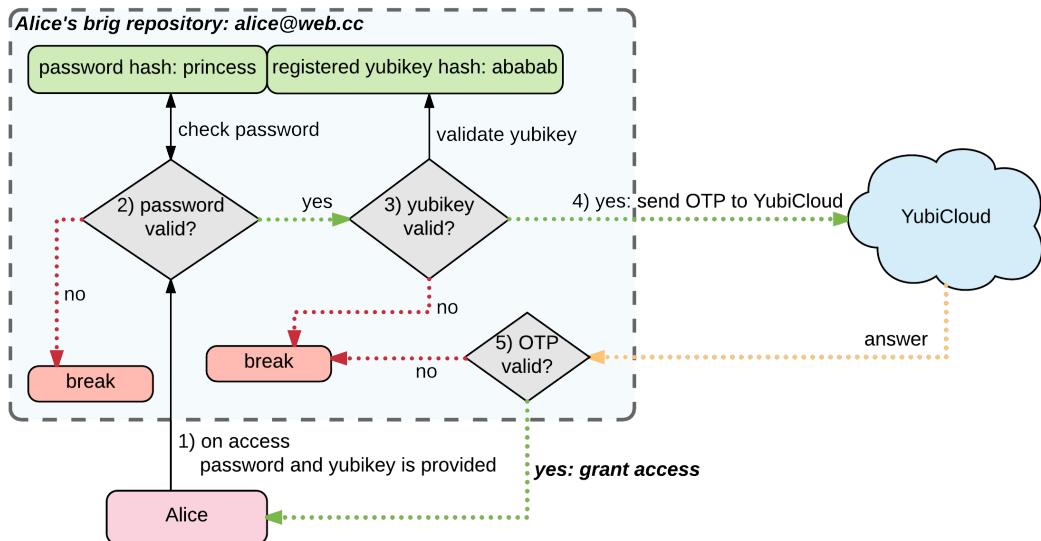


Abbildung 8.17.: Schematische Darstellung der Zwei-Faktor-Authentifizierung gegenüber einem »brig«-Repository.

1. Alice startet mit ihrem Passwort und YubiKey einen Loginvorgang.
2. »brig« prüft das Passwort von Alice.
3. »brig« prüft anhand der Public-ID, ob der YubiKey von Alice dem Repository bekannt ist.
4. »brig« lässt das One-Time-Password des YubiKey von der YubiCloud validieren.
5. Ist das One-Time-Password korrekt, so bekommt Alice Zugriff auf das Repository.

Eine essentiell wichtige Komponente an dieser Stelle ist der Zusammenhang zwischen dem Repository und dem YubiKey. Der YubiKey muss beim Initialisieren dem System genauso wie das Passwort bekannt gemacht werden. Passiert dies nicht, so könnte sich jeder Benutzer mit einem gültigen YubiKey gegenüber »brig« authentifizieren. Der YubiKey kann bei der Initialisierung durch ein One-Time-Password gegenüber »brig« bekannt gemacht werden.

Beim Ausführen des Code-Snippets (Anhang M) wird das Passwort als erster Parameter übergeben. Der YubiKey-OTP-Schlüssel ist der zweite übergebene Parameter. Die Proof-of-Concept-Implementierung hat aktuell einen YubiKey registriert und das Passwort *Katzenbaum* als valide anerkannt.

Authentifizierung mit korrektem Passwort und YubiKey:

```
$ ./twofac katzenbaum ccccccelefliufignnbjllkhrfeklifntfknicfbilced
[yubico: OK, brig? : OK, password: OK]
```

Authentifizierung mit falschem Passwort und korrektem YubiKey:

```
$ ./twofac elchwald ccccccelefliujnnbjllkhrfeklifntfknicfbilced
[yubico: OK, brig : OK, password: X]
```

Authentifizierung mit falschem Passwort und falschem YubiKey:

```
$ ./twofac elchwald ccccccejjegnjrvnbbthinvbvrbbjerljknbeteluugut
[yubico: OK, brig : X, password: X]
```

Authentifizierung mit falschem Passwort und dem zuletzt wiederholten One-Time-Password des falschen YubiKey:

```
$ ./twofac elchwald ccccccejjegnjrvnbbthinvbvrbbjerljknbeteluugut
2016/12/12 22:41:30 The OTP is valid, but has been used before. \
If you receive this error, you might be the victim of a man-in-the-middle attack.
[yubico: X, brig : X, password: X]
```

8.4.7. Konzept mit eigener Serverinfrastruktur

8.4.7.1. Allgemein

Neben der Möglichkeit, das YubiKey One-Time-Password gegen die YubiCloud validieren zu lassen, gibt es auch die Möglichkeit, eine eigene Infrastruktur für die Validierung bereitzustellen³⁶.

Dies ist in erster Linie für Unternehmen interessant, da keine Abhängigkeit zu einem externen Dienst besteht. Weiterhin bekommt das Unternehmen dadurch mehr Kontrolle über den Verwendungszweck und Einsatz und kann den YubiKey feingranularer als Sicherheitstoken nicht nur für »brig«, sondern die gesamte Unternehmensinfrastruktur nutzen.

8.4.7.2. Einrichtung

Als Vorbereitung muss der YubiKey mit einer neuen Identität programmiert werden. Für die Programmierung wird das YubiKey Personalization Tool verwendet. Hier kann unter dem Menüpunkt *Yubico OTP/Quick* eine neue Identität autogeneriert werden. Die hier erstellte Public-ID, sowie der AES-Schlüssel müssen anschließend dem Validierungsserver bekannt gemacht werden. Zum Testen wird folgend ein in Go³⁷ geschriebener Validierungsserver verwendet.

Für die Registrierung einer neuen Identität für die YubiCloud stellt Yubico eine Seite³⁸ bereit, über welche der AES-Schlüssel an die Yubico Validierungsserver geschickt werden kann.

Registrierung einer neuen Applikation am eigenen Validierungsserver:

```
$ ./yubikey-server -app "brig"
app created, id: 1 key: Q0w7RkvWxL/lvCynBh+TYiuhZKg=
```

Registrierung eines YubiKey unter dem Benutzernamen Christoph und Übergabe der Public- und Secret-ID des YubiKeys:

```
$ ./yubikey-server -name "Christoph" -pub "vvrfglutrrgk" \
-secret "619d71e138697797f7af68924e8ecd68"
```

³⁶Validation Servers: https://developers.yubico.com/Software_Projects/Yubico OTP/YubiCloud_Validation_Servers/

³⁷Go YubiKey Server: <https://github.com/stumpyfr/yubikey-server>

³⁸Yubico AES-Key-Upload: <https://upload.yubico.com/>

```
creation of the key: OK
```

Server auf localhost starten:

```
$ ./yubikey-server -s
2016/12/08 19:28:20 Listening on: 127.0.0.1:4242...
```

8.4.7.3. Testen des Validierungsservers

Der folgende Konsolenauszug zeigt die Validierung am lokalen Server. Für den Zugriff wird das Kommandozeilen-Tool cURL³⁹ verwendet. Die URL für die Anfrage ist wie folgt aufgebaut:

```
http://<ip>:<port>/wsapi/2.0/verify?otp=<otp>&id=<app id>&nonce=<nonce>
```

Auf der Kommandozeile werden die Platzhalter durch ein valides One-Time-Password, eine valide Applikations-ID und eine zufällige Nonce ersetzt. Der Server läuft standardmäßig auf localhost, Port 4242:

```
# Validierung des YubiKey OTP
$ curl "http://localhost:4242/wsapi/2.0/verify \
?otp=vvrglutrgrgkkddjfnkjlituvfglnbkfghlnjvnkflj&id=1&nonce=test42"
nonce=test42 otp=vvrglutrgrgkkddjfnkjlituvfglnbkfghlnjvnkflj
name=Christoph
status=OK
t=2016-12-08T19:29:20+01:00
h=7DJyK6NZ0IeCcs9lHcH+K8RFaYY=
```

Beim wiederholten Einspielen des gleichen OTP verhält sich der eigene Validierungsserver genauso wie die YubiCloud und gibt die erwartete Fehlermeldung REPLAYED OTP aus.

```
# Wiederholtes OTP
$ curl "http://localhost:4242/wsapi/2.0/verify \
?otp=vvrglutrgrgkkddjfnkjlituvfglnbkfghlnjvnkflj&id=1&nonce=test42"
nonce=test42
otp=vvrglutrgrgkkddjfnkjlituvfglnbkfghlnjvnkflj
name=
status=REPLAYED OTP
t=2016-12-08T19:35:18+01:00
h=Jq0407mZWS4Us/J/n2jCtbSnRFk=
```

8.4.7.4. Sicherheit

Beim Betreiben eines eigenen Validierungsservers muss besonderer Wert auf die Sicherheit gelegt werden, da der Server die AES-Schlüssel der registrierten YubiKeys enthält.

³⁹cURL Homepage: <https://curl.haxx.se/>

Es ist für Unternehmen empfehlenswert, den Validierungsserver nicht direkt am Netz, sondern über einen Reverse-Proxy zu betreiben. Neben der GO-Implementierung haben andere Validierungsserver auf Python-⁴⁰ und C-Basis^{41,42} in der Vergangenheit kritische Sicherheitslücken aufgewiesen.

Abb. 8.18 zeigt einen Ansatz bei welchem der Validierungsserver hinter einem Reverse-Proxy betrieben wird. Alle One-Time-Password-Anfragen werden über einen normalen Webserver entgegengenommen und an den YubiKey-Validierungsserver weitergeleitet.

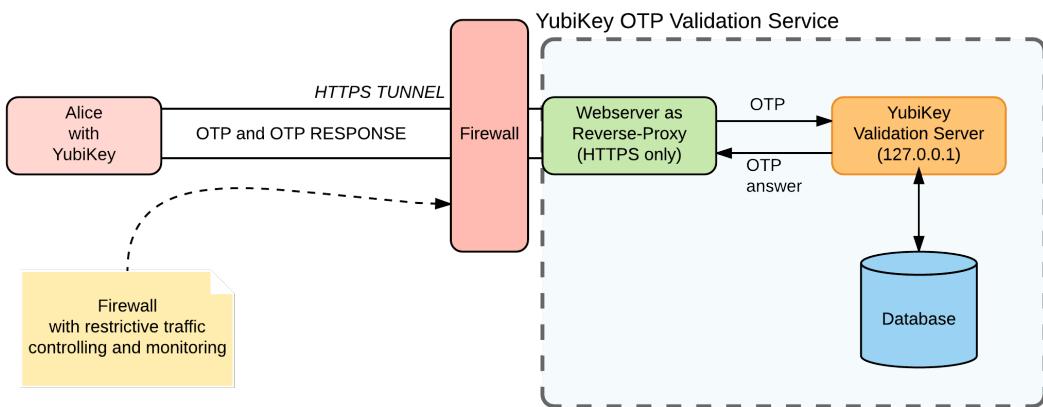


Abbildung 8.18.: Validierungsserver, welcher über einen Reverse-Proxy angesprochen wird.

Der Vorteil an dieser Stelle ist, dass je nach Organisation und Unternehmensgröße ein anderer Validierungsserver im Hintergrund eingesetzt werden kann. Es ist zusätzlich weiterhin davon auszugehen, dass kritische Sicherheitslücken in weit verbreiteten Webservleten wie NGINX⁴³ oder Apache⁴⁴ schneller gefunden und behoben werden, wie bei einem spezifischen Server-Produkt. Weiterhin kann der Webserver feingranularer konfiguriert werden und Sicherheitsfeatures, wie beispielsweise X-XSS-Protection oder HTTP Strict Transport Security (HSTS)⁴⁵ aktiviert werden. Wichtig an dieser Stelle ist auch der ausschließliche Einsatz von HTTPS mit einem validen Zertifikat als Transportprotokoll, um mögliche Man-in-the-Middle-Angriffe zu verhindern.

Da der Webserver besonders gut gesichert sein muss, würde sich an dieser Stelle neben der architektonischen Trennung auch die Umsetzung einer internen sicherheitsorientierten Systemarchitektur anbieten, um das Risiko bestimmter Exploit-Arten zu minimieren. Unter Linux, beispielsweise mittels grsecurity, SELinux (vgl. [40]) oder mittels Jails bei einem BSD-Derivat wie FreeBSD.

Für den Einsatz bei gemeinnützigen Organisationen oder auch öffentlichen Institutionen wie einer Hochschule, würde sich zusätzlich der Einsatz von Letsencrypt⁴⁶ für die Bereitstellung kostenfreier validierter Webserver-Zertifikate eignen.

⁴⁰Yubico-YubiServer SQL Injection Vulnerability: <https://code.google.com/archive/p/yubico-yubiserve/issues/38>

⁴¹YubiServer CVE-2015-0842 SQL Injection Vulnerability: <https://security-tracker.debian.org/tracker/CVE-2015-0842>

⁴²YubiServer CVE-2015-0843 Buffer Overflow Vulnerability: <https://security-tracker.debian.org/tracker/CVE-2015-0843>

⁴³NGINX-Webserver Homepage: <https://nginx.org/en/>

⁴⁴Apache-Webserver Homepage: <https://httpd.apache.org/>

⁴⁵Hardening Your HTTP Security Headers: <https://www.keycdn.com/blog/http-security-headers/>

⁴⁶Let's Encrypt-Homepage: <https://letsencrypt.org/about/>

8.4.8. Einsatz des YubiKey zur Passworthärtung

Unter Abschnitt 8.4.4 wird die Funktionalität »Statische Passwörter« erwähnt. Diese Funktionalität ermöglicht es, auf dem YubiKey NEO ein bis zu 32-Zeichen langes Passwort zu hinterlegen. Der YubiKey arbeitet aus Kompatibilitätsgründen mit einem Modhex-Alphabet⁴⁷. Die Konfiguration kann entweder bequem vom Benutzer mit der yubikey-personalization-gui erfolgen oder unter Linux beispielsweise auch mit dem ykpersonalize-Werkzeug (dieses ist Teil des YubiKey Personalization Tool) mit Hilfe des ModHex-Converters⁴⁸ von Michael Stapelberg:

```
$ $(perl pw-to-yubi.pl MyVeryLongPasswordYouWontGuessToday)
Firmware version 3.4.1 Touch level 1551 Program sequence 6
```

Configuration data to be written to key configuration 2:

```
fixed: m:kcbrrkcjbgbrjvdbbc1kecfbhbb1bd
uid: 15079c12189a
key: h:1211178a18081616971207041c000000
acc_code: h:000000000000
ticket_flags:
config_flags: SHORT_TICKET
extended_flags:
```

```
Commit? (y/n) [n]: y
```

Im Beispiel wurde der zweite Slot des YubiKey NEO mit dem Passwort MyVeryLongPasswordYouWontGuessToday konfiguriert. Beim anschließenden zweimaligem Aktivieren des zweiten Slots, liefert der YubiKey zweimal das gleiche Passwort:

```
$ MyVeryLongPasswordYouWontGuessTodayMyVeryLongPasswordYouWontGuessToday
```

Dieses Feature erlaubt es dem Benutzer durch Merken eines einfachen Passwortes, trotzdem ein sicheres Passwort generieren zu können. Je nach Anwendung kann so das vom YubiKey generierte Passwort mit einem Präfix und/oder Suffix erweitert werden. Beispiel mit Präfix = »YEAH« und Suffix = »GehtDoch!?« (Benutzereingabe + YubiKey + Benutzereingabe):

```
$ YEAHMyVeryLongPasswordYouWontGuessTodayGehtDoch! ?
```

Auch wenn dieses Feature dem Benutzer das Merken von langen Passwörtern erspart, so sollte es trotzdem mit Vorsicht eingesetzt werden, da Standard-Passwörter jederzeit beispielsweise mittels eines Keyloggers aufgezeichnet werden können. Die bessere Alternative ist an dieser Stelle trotzdem eine echte Zwei-Faktor-Authentifizierung, sofern diese von der jeweiligen Applikation beziehungsweise vom Dienstanbieter angeboten wird.

⁴⁷YubiKey Static Password Function:

https://www.yubico.com/wp-content/uploads/2015/11/Yubico_WhitePaper_Static_Password_Function.pdf

⁴⁸GitHub pw-to-yubi.pl: <https://github.com/stapelberg/pw-to-yubi/blob/master/pw-to-yubi.pl>

8.4.9. YubiKey als Smartcard

8.4.9.1. Einleitung

Wie unter Abschnitt 8.4.4 erwähnt, hat der YubiKey die Möglichkeit als Smartcard zu fungieren. Die Chip Card Interface Device (CCID)⁴⁹-Funktionalität ist beim YubiKey NEO ab Werk deaktiviert. Für die Aktivierung kann das Kommandozeilen-Werkzeug `ykpersonalize` verwendet werden. Standardmäßig ist beim YubiKey NEO nur die OTP-Funktionalität aktiviert. In welchem Betriebsmodus sich der YubiKey befindet, kann man beispielsweise nach dem Anstecken über das System/Kernel-Logging mittels `dmesg` herausfinden (gekürzte Ausgabe):

```
$ dmesg | tail -n 2
[324606.823079] input: Yubico YubiKey NEO OTP as /devices/pci[...]
[324606.877786] hid-generic 0003:1050:0110.023F: input,hidraw5: \
USB HID v1.10 Keyboard [Yubico YubiKey NEO OTP] on usb-0000:00:1d.0-1.8.1.3/input0
```

Beim Aktivieren kann man beim YubiKey NEO zwischen insgesamt sieben verschiedenen Modi — Einzelmodi und Kombinations-Modi — wählen⁵⁰:

- ▶ Einzel-Modi: OTP, CCID, U2F
- ▶ Kombinations-Modi: OTP/CCID, OTP/U2F, U2F/CCID, OTP/U2F/CCID

8.4.9.2. Aktivierung des OpenPGP-Applets

Da der YubiKey im Falle von »brig« auch als Authentifizierungs- und Signertoken für die Entwickler dienen soll (siehe Abschnitt 8.5.3), bietet sich der OTP/CCID Kombimodus an. Dieser kann mit dem Kommandozeilenprogramm `ykpersonalize` wie folgt aktiviert werden:

```
$ ykpersonalize -m2
Firmware version 3.4.1 Touch level 1551 Program sequence 3

The USB mode will be set to: 0x2

Commit? (y/n) [n]: y
```

Nach dem erneuten Anstecken des YubiKeys meldet sich dieser am System als Human Interface Device (HID) mit OTP+CCID-Funktionalität an (gekürzte Ausgabe):

```
dmesg | tail -n 2
[324620.832438] input: Yubico YubiKey NEO OTP+CCID as /devices/pci[...]
[324620.888719] hid-generic 0003:1050:0111.0240: input,hidraw5: \
USB HID v1.10 Keyboard [Yubico YubiKey NEO OTP+CCID] on usb-0000:00:1d.0[...]
```

⁴⁹CCID (protocol): [https://en.wikipedia.org/w/index.php?title=CCID_\(protocol\)&oldid=745448227](https://en.wikipedia.org/w/index.php?title=CCID_(protocol)&oldid=745448227)

⁵⁰YubiKey NEO Modes: https://developers.yubico.com/libu2f-host/Mode_switch_YubiKey.html

8.4.9.3. Übertragung kryptographischer Schlüssel auf den YubiKey

Nach der Aktivierung des OpenPGP-Applets kann der YubiKey wie eine Standard-OpenPGP-Smartcard mit GnuPG verwendet werden.

`gpg2 --card-status` zeigt den aktuellen Inhalt des YubiKey OpenPGP-Applets:

```
$ gpg2 --card-status
Reader .....: 0000:0000:0:0
Application ID ....: 00000000000000000000000000000000
Version .....: 2.0
Manufacturer .....: Yubico
Serial number ....: 00000000
Name of cardholder: [not set]
Language prefs ....: [not set]
Sex .....: unspecified
URL of public key : [not set]
Login data .....: [not set]
Signature PIN ....: forced
Key attributes ....: rsa2048 rsa2048 rsa2048
Max. PIN lengths ..: 127 127 127
PIN retry counter : 3 3 3
Signature counter : 0
Signature key ....: [none]
Encryption key....: [none]
Authentication key: [none]
General key info...: [none]
```

Wie die Ausgabe zeigt, gibt es die Möglichkeit, drei verschiedene RSA-Schlüssel (2048 Bit) zum Signieren, Ver-/Entschlüsseln und Authentifizieren zu speichern. Der Authentifizierungsschlüssel, der hier gesetzt werden kann, wird nicht von `gpg2` genutzt, jedoch von anderen PAM-basierten⁵¹ Anwendungen. Für weitere Informationen zum Aufbau des Applets und zu den Funktionalitäten (PIN-Counter et cetera) siehe die GnuPG-Administrationsdokumentation zur Smartcard⁵².

Es gibt zwei Möglichkeiten die Smartcard mit kryptographischen Schlüsseln zu befüllen:

1. Schlüssel direkt auf Smartcard generieren lassen.
2. Schlüssel extern auf PC generieren und auf Smartcard verschieben.

Variante 1:

Die Schlüssel lassen sich direkt mit `gpg2 --card-edit` auf der Smartcard generieren. Hier muss man in den admin-Modus wechseln und kann anschließend mit dem Befehl `generate` die Schlüssel generieren lassen. Anhang F zeigt den kompletten Vorgang. Beim Generieren der Schlüssel wird man von der Anwendung gefragt, ob ein Schlüssel »off card«-Backup gemacht werden soll. Weiterhin

⁵¹PAM: https://de.wikipedia.org/w/index.php?title=Pluggable_Authentication_Modules&oldid=146057418

⁵²Chapter 3. Administrating the Card: <https://www.gnupg.org/howtos/card-howto/en/ch03.html>

werden Revocation-Zertifikate generiert. Früher mussten diese manuell erstellt werden, aktuelle GnuPG-Versionen erstellen diese automatisch.

Auszug aus Anhang F:

```
gpg: Note: backup of card key saved to '/home/qitta/.gnupg/sk_E5A1965037A8E37C.gpg'
gpg: key 932AEBFDD72FE59C marked as ultimately trusted
gpg: revocation certificate stored as '/home/qitta/.gnupg/openpgp-revocs.d/ \
D61CEE19369B9C330A4A482D932AEBFDD72FE59C.rev'
```

Der Hinweis, dass der Schlüssel sk_E5A1965037A8E37C.gpg (sk == secret key) gespeichert wurde, ist an dieser Stelle irreführend. Es wurde hier lediglich ein sogenannter Stub⁵³ erstellt, welcher den eigentlichen privaten Schlüssel *nicht* enthält. Bei Schlüsseln, die auf der Smartcard generiert wurden, gibt es *keine* Möglichkeit des Backups der privaten Schlüssel. Dies hat zur Folge, dass man bei einem Defekt oder Verlust auch die auf der Smartcard erstellte Identität verliert.

Variante 2:

Die zweite Variante ermöglicht es dem Benutzer ein echtes Backup der privaten Schlüssel anzulegen. Ein Schlüsselpaar kann hier mit den Standardbefehlen gpg2 --gen-key angelegt werden. Wird der Expertenmodus nicht verwendet, so legt GnuPG standardmäßig einen Haupt- und einen Unterschlüssel an (siehe Abschnitt 8.2.2.2).

Wie unter Abschnitt 8.2.2.3 erwähnt, ist es sinnvoll für den täglichen Einsatz Unterschlüssel zu generieren, da diese das Keymanagement erheblich erleichtern. Für die Evaluation der Authentifizierung über die Smartcard wird der bereits in Abschnitt 8.2.2.2 gezeigte Entwicklerschlüssel erweitert:

```
$ gpg2 --list-keys --fingerprint --fingerprint \
E9CD5AB4075551F6F1D6AE918219B30B103FB091
pub    rsa2048 2013-02-09 [SC] [expires: 2017-01-31]
      E9CD 5AB4 0755 51F6 F1D6  AE91 8219 B30B 103F B091
uid          [ultimate] Christoph Piechula <christoph@nullcat.de>
sub    rsa2048 2013-02-09 [E] [expires: 2017-01-31]
      6258 6E4C D843 F566 0488  0EB0 0B81 E5BF 8582 1570
```

Die Erweiterung des Schlüssels um einen Unterschlüssel zum Signieren und Unterschlüssel zum Authentifizieren wird in Anhang G gezeigt. Weiterhin wurde der Hauptschlüssel um 10 Jahre Laufzeit erweitert. Der Ver-/Entschlüsselungs-Unterschlüssel wurde um 2 Jahre erweitert (siehe Anhang H). Nach dem Anpassen schaut der für die Smartcard vorbereitete Schlüssel wie folgt aus:

```
$ gpg2 --list-keys --fingerprint --fingerprint \
E9CD5AB4075551F6F1D6AE918219B30B103FB091
pub    rsa2048 2013-02-09 [SC] [expires: 2026-12-09]
      E9CD 5AB4 0755 51F6 F1D6  AE91 8219 B30B 103F B091
uid          [ultimate] Christoph Piechula <christoph@nullcat.de>
sub    rsa2048 2013-02-09 [E] [expires: 2018-12-11]
      6258 6E4C D843 F566 0488  0EB0 0B81 E5BF 8582 1570
```

⁵³gpg(1) - Linux man page: <https://linux.die.net/man/1/gpg>

```

sub rsa2048 2016-12-11 [S] [expires: 2018-12-11]
    7CD8 DB88 FBF8 22E1 3005 66D1 2CC4 F84B E43F 54ED
sub rsa2048 2016-12-11 [A] [expires: 2018-12-11]
    2BC3 8804 4699 B83F DEA0 A323 74B0 50CC 5ED6 4D18

```

Beim Verschieben der Schlüssel auf die Smartcard werden von GnuPG sogenannte Stubs für die privaten Schlüssel erstellt. Deshalb sollte spätestens jetzt ein Backup von den privaten Haupt- und Unterschlüsseln erfolgen. Dies kann am einfachsten über das Kopieren des .gnupg-Konfigurationsordners bewerkstelligt werden. Dieser enthält die `pubring.gpg` und `secring.gpg` Dateien, welche die öffentlichen und privaten Schlüssel enthalten. Eine alternative Methode einen bestimmten Schlüssel zu sichern, zeigt [Anhang I](#). Verfahren zur Offline-Speicherung von Schlüsseln wurden bereits unter [Abschnitt 8.2.2.3](#) behandelt.

Nach dem Verschieben schaut die Ausgabe der privaten Schlüssel im Schlüsselbund wie folgt aus:

```

$ gpg2 --list-secret-keys --fingerprint --fingerprint \
E9CD5AB4075551F6F1D6AE918219B30B103FB091
sec rsa2048 2013-02-09 [SC] [expires: 2026-12-09]
    E9CD 5AB4 0755 51F6 F1D6 AE91 8219 B30B 103F B091
uid      [ultimate] Christoph Piechula <christoph@nullcat.de>
ssb> rsa2048 2013-02-09 [E] [expires: 2018-12-11]
    6258 6E4C D843 F566 0488 0EB0 0B81 E5BF 8582 1570
    Card serial no. = 0006 00000000
ssb> rsa2048 2016-12-11 [S] [expires: 2018-12-11]
    7CD8 DB88 FBF8 22E1 3005 66D1 2CC4 F84B E43F 54ED
    Card serial no. = 0006 00000000
ssb> rsa2048 2016-12-11 [A] [expires: 2018-12-11]
    2BC3 8804 4699 B83F DEA0 A323 74B0 50CC 5ED6 4D18
    Card serial no. = 0006 00000000

```

GnuPG teilt mit `Card serial no. = 0006 00000000` dem Benutzer mit, auf welcher Smartcard sich die Schlüssel befinden. Am »><-Symbol erkennt der Benutzer, dass für die mit diesem Zeichen gekennzeichneten Schlüssel nur ein Stub existiert. Der auffällige Teil an dieser Stelle ist, dass der Hauptschlüssel weiterhin existiert. Dies hängt damit zusammen, dass nur die Unterschlüssel auf den YubiKey übertragen wurden. Die gekürzte Variante von `gpg2 --card-status` zeigt die Schlüssel auf der Smartcard.

```

$ gpg2 --expert --card-status | head -n 21 | tail -n 6

Signature key ....: 7CD8 DB88 FBF8 22E1 3005 66D1 2CC4 F84B E43F 54ED
                  created ....: 2016-12-11 16:32:58
Encryption key....: 6258 6E4C D843 F566 0488 0EB0 0B81 E5BF 8582 1570
                   created ....: 2013-02-09 23:18:50
Authentication key: 2BC3 8804 4699 B83F DEA0 A323 74B0 50CC 5ED6 4D18
                   created ....: 2016-12-11 16:34:21

```

Um den in Abschnitt 8.2.2.3 vorgeschlagenen Weg zu gehen und den Hauptschlüssel nur zum Signieren neuer Schlüssel zu verwenden, sollte dieser am Schluss aus dem Schlüsselbund gelöscht werden. Dies kann mit `gpg2 --delete-secret-keys E9CD5AB4075551F6F1D6AE918219B30B103FB091` erledigt werden. Wird der Hauptschlüssel gelöscht, so erscheint beim Hauptschlüssel das »#«-Symbol, um anzugeben, dass es sich nur um einen Stub handelt. Anschließend können die neuen öffentlichen Unterschlüssel dem Web of Trust mit `gpg2 --send-keys E9CD5AB4075551F6F1D6AE918219B30B103FB091` bekannt gemacht werden.

Anschließend sollte noch die Standard-PIN 123456 und die Standard-Admin-PIN 12345678 geändert werden. Diese Einstellung kann ebenso mit `gpg2 --card-edit` im Untermenü `admin/passwd` getätigt werden. Abschnitt 8.5.1.1 zeigt das Signieren von Daten mit und ohne Smartcard.

8.5. Sichere Entwicklung und Entwicklungsumgebung

8.5.1. Bereitstellung der Software

8.5.1.1. Erstellen und Validieren von Signaturen

Um die Applikation sicher an den Benutzer ausliefern zu können, gibt es verschiedene Möglichkeiten. Die für den Benutzer einfachste Möglichkeit ist es, im Falle einer Linux-Distribution, sich die signierten Pakete mittels Paketmanager zu beschaffen. In diesem Fall muss sich der Paketierer der Applikation um die Auslieferung einer korrekt signierten Version kümmern.

Eine weitere Möglichkeit, die dem Benutzer mehr Kontrolle gibt, ist das direkte Herunterladen auf der Entwickler/Anbieter-Webseite. Diese Art der Bereitstellung von Software bietet beispielsweise auch das GnuPG und das Tor-Projekt an.

Hier gibt es die Möglichkeit, die Daten direkt zu signieren oder eine separate Signatur zu erstellen. Für die Bereitstellung von Binärdaten ist die Bereitstellung einer separaten Signatur empfehlenswert, da die eigentlichen Daten dabei nicht modifiziert werden.

Folgendes Kommandozeilen-Snippet zeigt das Signieren der Daten ohne Entwickler-YubiKey:

```
$ gpg2 --armor --output brig-version-1.0.tar.gz.asc \
--detach-sign brig-version-1.0.tar.gz
gpg: signing failed: Card error
gpg: signing failed: Card error
```

Unter Einsatz des YubiKey und der korrekten PIN können die Daten wie folgt signiert werden. Die `--armor`-Option bewirkt, dass eine Signatur im ASCII-Format anstatt des Binär-Formates erstellt wird. Diese lässt sich beispielsweise auf der Download- beziehungsweise Entwicklerseite neben dem Fingerprint der Signierschlüssel publizieren.

```
$ gpg2 -v --armor --output brig-v1.0.tar.gz.asc --detach-sign brig-v1.0.tar.gz
gpg: using pgp trust model
gpg: using subkey 2CC4F84BE43F54ED instead of primary key 8219B30B103FB091
gpg: writing to 'brig-v1.0.tar.gz.asc'
gpg: pinentry launched (pid 26215, flavor gtk2, version 1.0.0)
```

```
gpg: RSA/SHA256 signature from: "2CC4F84BE43F54ED \
Christoph Piechula <christoph@nullcat.de>"
```

Die `-v`-Option visualisiert den Signiervorgang detaillierter. Zusätzlich zu sehen ist beispielsweise, dass als Schlüssel zum Signieren ein Unterschlüssel mit der ID 2CC4F84BE43F54ED anstelle des Hauptschlüssels mit der ID 8219B30B103FB091 verwendet wird.

Nach dem Signiervorgang entsteht folgende Signaturdatei:

```
$ cat brig-v1.0.tar.gz.asc
-----BEGIN PGP SIGNATURE-----

iQEzBAABCAAdFiEEfNjbiPv4IuEwBWbRLMT4S+Q/V00FA1hNuggACgkQLMT4S+Q/
V01DDQgAkAlF3y16rvwQjRgkWyAL1ujeWrecxdplNjde44zToBuFNutP66wUKnqr
ZohLP3TEdCuEJtvxzv7ahEw8IC0v4375IvyediKjXV+f8t8Kau64bqCoZOHiJYWY
tbMwuOrG+rgP38crZSEfRjLSc2ZgUntvmQRI103Id9K8XZtLPx8NK8qUvfyI8D5c
27oLmGrfGcgnywr+2dqUWhdCRa7J176vmRl25631PBU807k5mew1L7hXMRFnwBZ
mNgj91cAGJETaKJIMjnxl+GI4u0BNAszKXzmQFYE3sm+VBvJ89vkNvc1C8wx+eva
VOotfneAcQoRgHOpouNpHew+uJ0/eg==
=N0vn
-----END PGP SIGNATURE-----
```

Der Benutzer kann durch diesen Ansatz die heruntergeladenen Daten auf einfache Art und Weise verifizieren. Bei Angabe des Verbose-Flags sieht dieser auch, dass die Datei mit einem Unterschlüssel signiert wurde und zu welchem Hauptschlüssel dieser gehört.

```
$ gpg2 -v --verify brig-v1.0.tar.gz.asc
gpg: assuming signed data in 'brig-v1.0.tar.gz'
gpg: Signature made Do 15 Dez 2016 16:55:28 CET
gpg:           using RSA key 7CD8DB88FBF822E1300566D12CC4F84BE43F54ED
gpg: using subkey 2CC4F84BE43F54ED instead of primary key 8219B30B103FB091
gpg: using pgp trust model
gpg: Good signature from "Christoph Piechula <christoph@nullcat.de>" [ultimate]
gpg: binary signature, digest algorithm SHA256, key algorithm rsa2048
```

Wurde eine Manipulation an den Daten oder an der Signatur – beispielsweise durch einen Angreifer oder Malware – durchgeführt, so schlägt die Verifizierung fehl:

```
$ gpg2 --verify brig-v1.0.tar.gz.asc
gpg: assuming signed data in 'brig-v1.0.tar.gz'
gpg: Signature made So 11 Dez 2016 21:41:44 CET
gpg:           using RSA key 7CD8DB88FBF822E1300566D12CC4F84BE43F54ED
gpg: BAD signature from "Christoph Piechula <christoph@nullcat.de>" [ultimate]
```

8.5.2. Update-Management

Auch bei Updates wäre in erster Linie die Bereitstellung über den Paketmanager die sinnvollste Variante der Auslieferung von »brig«. Um auch Betriebssysteme ohne Paketmanager unterstützen zu können, würde sich eine integrierte Updatefunktion besser eignen.

Diese könnte innerhalb von »brig« das oben genannte Verfahren mit der manuellen Überprüfung einer Signatur aufgreifen und automatisieren. Abb. 8.19 visualisiert den Updateprozess. Bei erfolgreicher initialer Validierung der Signatur nach dem ersten Herunterladen der Software (Schritt 1.1–1.3) kann davon ausgegangen werden, dass die Integrität gewährleistet ist und Softwareinterna nicht verändert wurden.

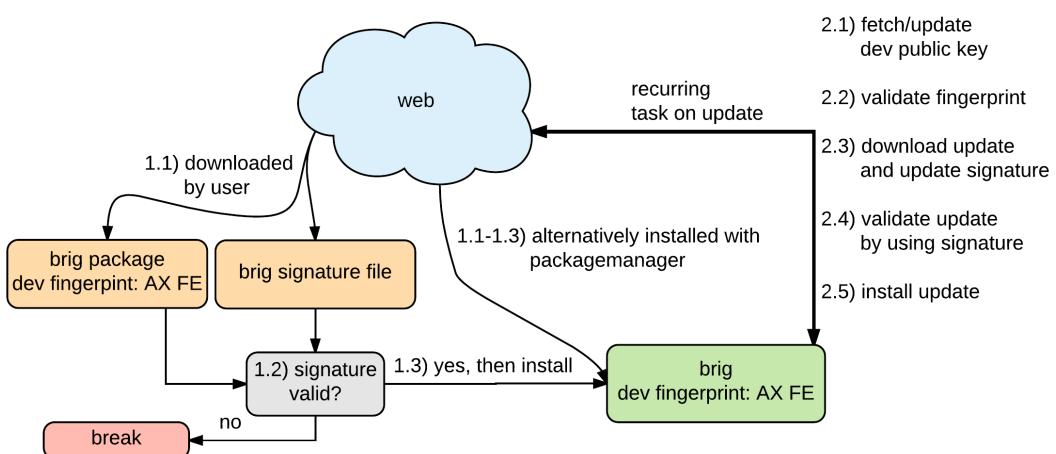


Abbildung 8.19.: Updateprozess zur sicheren Softwareverteilung.

Auf dieser Basis könnte die Software intern ein Schlüsselmanagement (revoked keys, valid keys) betreiben, anhand welchem sie Updates von der Entwicklerquelle lädt und automatisiert validiert (Schritt 2.1–2.5). Bei erfolgreicher Ausführung/Validierung aller Teilschritte kann die Software ein Update durchführen, ansonsten wird der Benutzer gewarnt und ein Update verweigert. Dies sollte sich beispielsweise mit einer High-Level-Bibliothek wie openpgp⁵⁴ realisieren lassen.

8.5.3. Signieren von Quellcode

Um die Verantwortlichkeit, sowie den Urheber bestimmter Quellcodeteile besser identifizieren zu können und dadurch die Entwickler auch vor Manipulationen am Quelltext-Repository besser zu schützen, bietet sich das Signieren von Quellcode (auch code signing genannt) an. Zwar bietet beispielsweise git bereits mit kryptographischen Prüfsummen (SHA-1) eine gewisse Manipulationssicherheit, jedoch ermöglicht SHA-1 keine Authentifizierung der Quelle.

Die Authentifizierung einer Quelle ist keine einfache Aufgabe. Gerade bei Open-Source-Projekten, bei welchen jeder der Zugriffsrechte besitzt, Quellcode zum Projekt beitragen kann, ist eine Authentifizierung um so wichtiger.

⁵⁴ Go OpenPGP-Bibliothek: <https://godoc.org/golang.org/x/crypto/openpgp>

Eine weit verbreitete Möglichkeit zur Authentifizierung, ist hierbei die Digitale Signatur. Quelltext, beziehungsweise Commits (Beitrag und Änderungen an einem Projekt), können auf eine ähnliche Art wie Binärdateien signiert werden.

Im Fall von »brig«, welches git zur Quellcode–Verwaltung verwendet und über GitHub entwickelt wird, bietet sich das Signieren und Verifizieren von Commits und Tags an. Hierzu kann beispielsweise mit `git config global` die Schlüssel-ID des Schlüssels hinzugefügt werden, mit welchem der Quelltext in Zukunft signiert werden soll. Anschließend kann mit dem zusätzlichen Parameter (`-S[<keyid>]`, `--gpg-sign[=<keyid>]`) ein digital unterschriebener Commit abgesetzt werden.

Für signierte Commits unter git sollte die Key-ID beziehungsweise der Fingerprint in der git-Konfigurationsdatei hinterlegt werden:

```
$ git config --global user.signingkey 8219B30B103FB091
```

Anschließend kann mit der `--gpg-sign`-Option ein signierter Commit abgesetzt werden:

```
$ git commit -S -m 'Brig evaluation update. Signed.'
```

Möchte man das Signieren dauerhaft aktivieren, so muss noch die `gpgsign`-Option gesetzt werden:

```
$ git config --global commit.gpgsign true
```

Durch das Signieren von Commits wird auf GitHub über ein Label ein erweiterter Status zum jeweiligen Commit beziehungsweise Tag angezeigt, siehe Abb. 8.20.

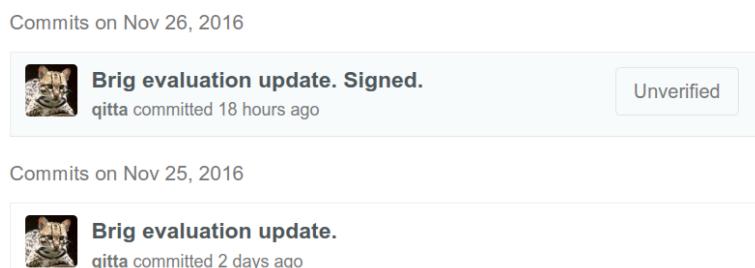


Abbildung 8.20.: Nach dem Absetzen eines signierten Commits/Tags erscheint auf der GitHub-Plattform ein zusätzliches Label »Unverified«, wenn der öffentliche Schlüssel des Entwicklers bei GitHub nicht hinterlegt ist.

Pflegen die Entwickler zusätzlich ihren öffentlichen Schlüssel im GitHub-Account ein, so signalisiert das Label eine verifizierte Signatur des jeweiligen Commits beziehungsweise Tags, siehe Abb. 8.21.

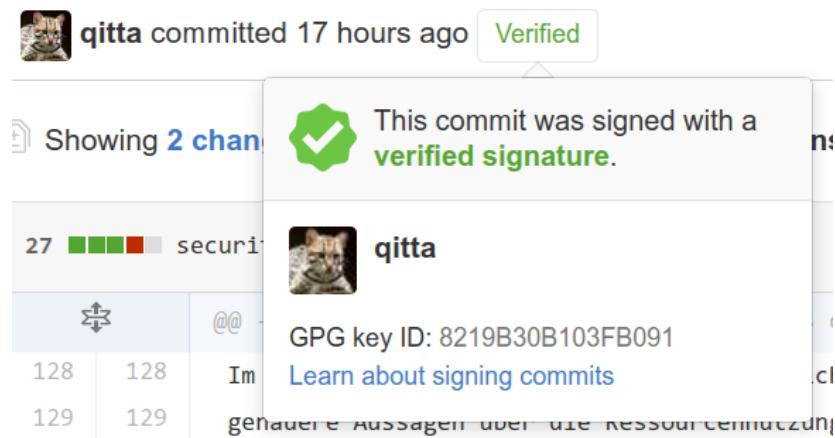


Abbildung 8.21.: Verifiziertes GitHub-Signatur-Label eines Commits/Tags welches aufgeklappt wurde.

Auf der Kommandozeile kann eine Signatur einfach mit der `--show-signature`-Option von `git` angezeigt werden (gekürzte Ausgabe):

```
$ git log --show-signature
[...]
commit 9fefd0e69791dfacade4bb1c9930776dcf73b8bc
gpg: Signature made Do 15 Dez 2016 16:23:02 CET
gpg:                               using RSA key 7CD8DB88FBF822E1300566D12CC4F84BE43F54ED
gpg: Good signature from "Christoph Piechula <christoph@nullcat.de>" [ultimate]
Author: qitta <christoph@nullcat.de>
Date:   Thu Dec 15 16:23:02 2016 +0100
```

Bibliography updated.

[...]

8.5.4. Sichere Authentifizierung für Entwickler

8.5.4.1. GitHub-Plattform

Um sich als Entwickler, beispielsweise sicher gegenüber der GitHub-Plattform zu authentifizieren, bietet die GitHub-Plattform Universal-Zwei-Faktor-Authentifizierung an, welche auch mit dem YubiKey genutzt werden kann⁵⁵.

Für diesen Einsatzzweck muss Universal-Zwei-Faktor-Authentifizierung (U2F) auf dem YubiKey aktiviert sein, dies kann nach dem Anstecken des YubiKey mit `dmesg` validiert werden:

```
$ dmesg | tail -n 2
[448904.638703] input: Yubico YubiKey NEO OTP+U2F+CCID as /devices/pci[...]
[448904.694278] hid-generic 0003:1050:0116.0364: input,hidraw4: USB HID[...]
```

⁵⁵GitHub-U2F: <https://github.com/blog/2071-github-supports-universal-2nd-factor-authentication>

Falls nötig kann der korrekte Modus analog zu Abschnitt 8.4.9.1 aktiviert werden.

Anschließend muss die Zwei-Faktor-Authentifizierung im persönlichen Account unter dem Menüpunkt *Setting* aktiviert und der YubiKey auf der GitHub-Plattform registriert werden. Nach erfolgreicher Registrierung wird man nach der Eingabe seines persönlichen Passworts um den zweiten Faktor (beispielsweise YubiKey) gebeten, siehe Abb. 8.22.

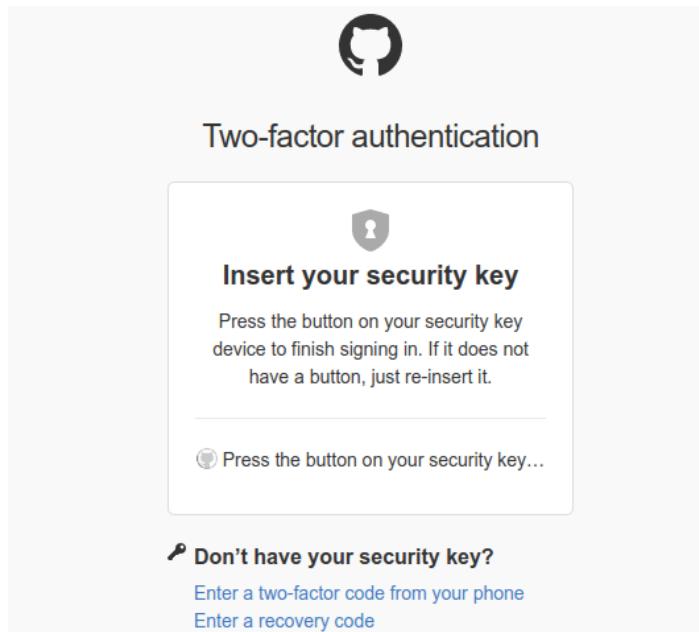


Abbildung 8.22.: GitHub-Anmeldung mit Zwei-Faktor-Authentifizierung.

8.5.4.2. Arbeiten über Secure-Shell (SSH)

Grundsätzlich gibt es, wie bei anderen Plattformen die Möglichkeit, sich mit dem Benutzernamen und dem Passwort gegenüber einer Plattform wie beispielsweise GitHub zu authentifizieren.

Public-Key-Authentifizierung:

Eine erweiterte Möglichkeit ist es, sich über einen SSH-Schlüssel zu authentifizieren. Abb. 8.23 zeigt schematisch den Ablauf einer Authentifizierung mittels Public-Key-Verfahren (vgl. [41], S. 63 f.).

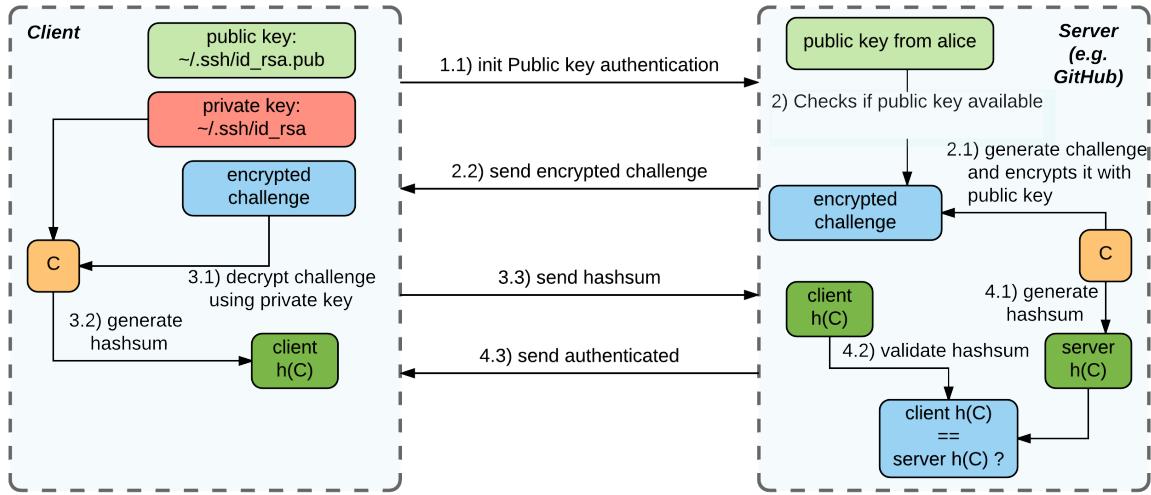


Abbildung 8.23.: SSH-Authentifizierungsvorgang mittels Public-Key-Verfahren.

- Der Client sendet eine Authentifizierungsanfrage und sendet dabei seinen öffentlichen Schlüssel an den Server.
- Der Server validiert ob der öffentliche Schlüssel in der Authorisierungsdatei des Accounts zu finden ist. Ist dies der Fall, so wird eine zufällige Challenge generiert, mit dem öffentlichen Schlüssel des Clients verschlüsselt und an diesen gesendet.
- Der Client entschlüsselt die Challenge, bildet eine Prüfsumme von dieser und sendet diese an den Server zurück.
- Der Server generiert auch eine Prüfsumme der Challenge, prüft ob seine generierte Prüfsumme mit der vom Client generierten Prüfsumme übereinstimmt. Ist dies der Fall, so erfolgt die Authentifizierung.

Der Client sendet an dieser Stelle nur die Prüfsumme der Challenge als Antwort zurück. Somit wird sichergestellt, dass ein kompromittierter Server nicht beliebig Daten wie beispielsweise verschlüsselte E-Mails an den Client schicken kann, damit dieser die Daten entschlüsselt. Bei der Authentifizierung über Public-Key-Verfahren liegen die Schlüssel in der Regel auf der Festplatte des Client-PC. Es besteht so die Gefahr, dass diese entwendet werden.

Public-Key-Authentifizierung mit Smartcard und GnuPG:

Das genannte Konzept lässt sich durch die Nutzung einer Smartcard erweitern. Die Vorteile liegen hier in erster Linie beim Schutz der kryptographischen Schlüssel.

Für die SSH-Authentifizierung mit einer Smartcard/GnuPG gegenüber GitHub – oder auch anderen Entwicklerplattformen – gibt es in dieser Kombination die folgenden zwei Varianten:

- OpenPGP-Schlüssel als SSH-Schlüssel verwenden (Schlüssel liegen auf Smartcard).
- GnuPG-Agent als SSH-Agent laufen lassen und sich mit den bisherigen SSH-Schlüsseln authentifizieren. Hierbei werden die SSH-Schlüssel vom gpg-agent geschützt verwaltet.

Bei der ersten Variante kann der gpg-agent so konfiguriert werden, dass er die SSH-Anwendung direkt unterstützt. Dazu muss die SSH-Unterstützung in der Konfigurationsdatei vom gpg-agent aktiviert werden:

```
echo "enable-ssh-support" >> ~/.gnupg/gpg-agent.conf
```

Weiterhin muss die Kommunikationsschnittstelle zwischen gpg-agent und SSH-Anwendung bekannt gemacht werden (Auszug aus `man gpg-agent`):

```
unset SSH_AGENT_PID
if [ "${gnupg_SSH_AUTH_SOCK_by:-0}" -ne $$ ]; then
    export SSH_AUTH_SOCK=$(gpgconf --list-dirs agent-ssh-socket)
fi
```

Der öffentliche Schlüssel kann mit dem `ssh-add`-Befehl extrahiert werden. Dieser muss anschließend wahlweise auf der GitHub-Plattform oder einem eigenen Entwicklungssystem, welches einen SSH-Zugang bereitstellt, hinterlegt werden.

```
$ ssh-add -L
ssh-rsa AAAAB3NzaC1yc2EAAAQABAAQDKLhnN7pTVB6YHn
7H5swJcJ+gABK6AtPQ1HqMBQHi9byXTPmtyrKBVo0yUfHkMphkyV
tFo6a892NfPSHDxxv0Ks3rZ36LdJ88K8VgT3F08N1nR1Vh1gIQzfr
MaaHPhC7m1cLjPdcXgTXPE6Bui3AP78xc/z5zGeUr1N5DLygTkqy8
BnZTB0UX8xMXUKtGSunPbHdjn7rStXX9LDX6BP32XvTHeseI0BYa/
2A1KRzxXhH9rHmpKLmMXVp9fRYmZ/51cVVR+XIBR23WBIJrm0vj3a
/Uh54ylZ3zKIql7E8PH8rpljQ9Uw/5dLSltotF0lw2zjF1t51EQrj
fViXwnf91 cardno:0006000000011
```

Betreibt man einen eigenen Server, so muss bei einem typischen unixoiden System mit OpenSSH der Public-Key auf dem Server in der `~/.ssh/authorized_keys`-Datei hinterlegt werden und das Public-Key-Verfahren in der `sshd`-Konfigurationsdatei aktiviert werden.

Ob die Authentifizierung über die Smartcard funktioniert, kann beim SSH-Login mittels `Verbose`-Flag validiert werden. Dieses ist auch für Debuggingzwecke empfehlenswert. Folgender gekürzter Auszug, zeigt eine erfolgreich konfigurierte GitHub-SSH-Smartcard-Authentifizierung, `cardno:000600000011` zeigt hierbei die erfolgreich verwendete Smartcard-ID:

```
#GIT_SSH_COMMAND wird benötigt um die Verbose Option von SSH zu aktivieren
$ GIT_SSH_COMMAND="ssh -vv" git push
[...]
debug1: Authentications that can continue: publickey,password
      # vom gpg-agent verwalteter Schlüssel auf Smartcard
debug1: Offering RSA public key: cardno:000600000011
debug2: we sent a publickey packet, wait for reply
debug1: Server accepts key: pkalg ssh-rsa blen 279
debug2: input_userauth_pk_ok: fp SHA256:eo0gD/8ri0RCblmJgTbPx/Ci6pBBssLtjMulpx4br1Y
debug1: Authentication succeeded (publickey).
```

```
Authenticated to github.com ([192.30.253.113]:22).
```

```
[...]
```

Ohne Smartcard/PIN schlägt der Loginversuch fehl. Bei der Verwendung eines eigenen Servers ist es empfehlenswert, eine Fallback-Loginmethode in der sshd-Konfigurationsdatei des Servers zu aktivieren, mehr hierzu siehe man `sshd_config` unter `AuthenticationMethods`. Für weitere Details zum Thema Konfiguration und OpenSSH-Authentifizierung siehe [42].

Für die Einrichtung der zweiten Variante (`gpg--agent` fungiert als `ssh-agent`) muss lediglich ergänzend ein generierter SSH-Schlüssel mit `ssh-add <identity>` dem `gpg-agent` bekannt gemacht werden. Der `gpg-agent` schützt den Schlüssel anschließend mit einer vom Nutzer vergebenen Passphrase und ersetzt somit einen separat laufenden `ssh-agent`. Mit `ssh-add -l` können die aktuell verwalteten Schlüssel-Fingerprints überprüft werden:

```
2048 SHA256:eo0gD/8ri0RCblmJgTbPx/Ci6pBBssLtjMulpX4br1Y cardno:000600000011 (RSA)
2048 SHA256:V00F8adokJZ1jo1WD+XPjSP51rT1/GVS3bh5YfJ0Ltk /home/qitta/.ssh/id_rsa (RSA)
```

Lädt man anschließend seine Änderungen bei GitHub über SSH hoch, so wird primär der vom `gpg--agent` verwaltete Schlüssel mittels Passwortabfrage freigegeben. Würde man diese abbrechen, so verwendet der Agent den öffentlichen Schlüssel auf der Smartcard. Im Folgenden wird die SSH-Public-Key-Authentifizierung über den vom `gpg-agent` verwalteten SSH-Schlüssel gezeigt:

```
$ GIT_SSH_COMMAND="ssh -vv" git push
[...]
debug1: Authentications that can continue: publickey
debug1: Next authentication method: publickey
# vom gpg-agent verwalteter Schlüssel
debug1: Offering RSA public key: /home/qitta/.ssh/id_rsa
debug2: we sent a publickey packet, wait for reply
debug1: Server accepts key: pkalg ssh-rsa blen 279
debug2: input_userauth_pk_ok: fp SHA256:V00F8adokJZ1jo1WD+XPjSP51rT1/GVS3bh5YfJ0Ltk
debug1: Authentication succeeded (publickey).
Authenticated to github.com ([192.30.253.112]:22).
[...]
```

9.1. Zusammenfassung

9.1.1. Allgemein

Wie bereits unter [27], S. 104 erwähnt, ist mit dem aktuellen Stand von »brig« eine solide Basis für ein sicheres und dezentrales Synchronisationswerkzeug entstanden.

Die übergreifende Anforderung an das Projekt, eine gute Balance zwischen Sicherheit und Usability zu finden, kann noch nicht endgültig bewertet werden. Die bisher getroffenen Entscheidungen bezüglich des Einsatzes von IPFS können durchaus positiv bewertet werden, siehe [Abschnitt 6.6](#). Die Datenhaltungsschicht bietet aufgrund des Merkle-DAG eine solide Basis mit Fehlererkennung, die Netzwerkschicht auf CAN-Basis setzt den dezentralen Ansatz von »brig« gut um. Die aktuell von »brig« umgesetzten Erweiterungen machen aus dem IPFS-Unterbau eine sichere und vollständig dezentrale Synchronisationslösung. Weiterhin wurden durch die Evaluation Fehler behoben und Verbesserungen für die bisherige Implementierung identifiziert (siehe [Kapitel 8](#)).

9.1.2. Schlüsselverwaltung

Die erweiterte Evaluation von IPFS hat ergeben, dass die IPFS-Identität (privater Schlüssel) im Klartext in der Konfigurationsdatei gespeichert wird (siehe [Abschnitt 6.4](#)). Zukünftige Versionen von »brig« müssen diesen kryptographischen Schlüssel sichern. Hier wurde ein mögliches Konzept für einen transparenten, verschlüsselten Zugriff auf Basis des Virtual Filesystem (VFS) vorgestellt (siehe [Abschnitt 8.2.1](#)).

Um die IPFS-Identität zu sichern und die zukünftige Schlüsselverwaltung von IPFS unabhängiger zu machen, wurde als Konzept eine »externe Identität« eingeführt ([Abschnitt 8.2.1](#)). Das Konzept mit dem Hauptschlüssel (externe Identität auf Basis eines Public-Key-Schlüsselpaares) gibt »brig« auch bei zukünftigen Veränderungen an der IPFS-Schlüsselverwaltung die Kontrolle über diese. Weiterhin wurde auf der Basis dieser Identität ein erweitertes Authentifizierungskonzept vorgestellt (siehe [Abschnitt 8.3.2](#)).

Die Kontrolle über die Identität ermöglicht eine feingranulare Kontrolle der verwendeten kryptographischen Schlüssel. Dies ermöglicht eine Sicherung der kryptographischen Schlüssel, beispielsweise auf einer Smartcard (siehe [Abschnitt 8.4.9.3](#)).

Die Thematik der externen Identität wurde weiterhin auf Basis von GnuPG anhand unterschiedlicher Konzepte und Möglichkeiten im Detail erläutert. Für mögliche Probleme bei der Verwaltung des Schlüsselpaares wurden verschiedene Lösungen evaluiert. Zusammengefasst stellt der Ansatz auf Basis der externen Identität einen hohen Sicherheitszugewinn dar.

9.1.3. Verschlüsselung

Die Sicherheitsentscheidungen, welche bisher für »brig« getroffen wurden, sind größtenteils positiv zu bewerten. Die Datenverschlüsselungsschicht hat aktuell zwar mit der vorhandenen IPFS-Transportlayer-Verschlüsselungsschicht einen gewissen Overhead, befindet sich mit standardisierten Verfahren (AES-GCM, ChaCha20/Poly1305) aber auf der sicheren Seite.

Die Implementierung einer modularen Verschlüsselungsschicht macht den Algorithmus leichter austauschbar. Die bisher gewählte Blockgröße der Verschlüsselungsschicht könnte noch teilweise optimiert werden. Die bisherigen Annahmen zur Algorithmuswahl konnten jedoch mit der Performance-Evaluation gut bestätigt werden. Systeme ohne kryptographischer Befehlssatzerweiterung profitieren vom ChaCha20/Poly1305 (siehe [Abschnitt 7.2.5](#)). Ab Go v1.6 profitieren Systeme mit kryptographischen Befehlserweiterungssatz stark von den AES-NI-Optimierungen. Schwächere Systeme wie der Raspberry Pi Zero haben trotz des recht performanten ChaCha20/Poly1305-Verfahrens eine relativ schlechte Schreib- und Lesegeschwindigkeit.

9.1.4. Authentifizierung

Die aktuelle Implementierung über die zxcvbn-Bibliothek setzt eine robuste Passwortvalidierung um. Das Problem hierbei ist einerseits die schlechte Usability, andererseits ist die alleinige Authentifizierung über ein Passwort — bei einem System mit Fokus auf Sicherheit — als negativ zu bewerten (siehe [Abschnitt 7.6](#)).

Hier wurden erweiterte Konzepte der Zwei-Faktor-Authentifizierung mit dem YubiKey OTP für Privatpersonen (siehe [Abschnitt 8.4.6](#)) und Institutionen ([Abschnitt 8.4.7](#)) evaluiert. Weiterhin wurde ein Konzept zur einfachen Passworthärtung mit dem YubiKey vorgestellt ([Abschnitt 8.4.8](#)).

Im Bereich der Synchronisationspartner-Authentifizierung wurden verschiedene Konzepte vorgestellt (siehe [Abschnitt 8.3](#)), über welche sich eine manuelle und automatisierte Authentifizierung des Synchronisationspartners durchführen lässt.

9.1.5. Softwareentwicklung und Softwareverteilung

Abschließend wurden Konzepte zur sicheren Verteilung der Software evaluiert. Hierbei wurden neben Konzepten zum Update-Management (siehe [Abschnitt 8.5.2](#)), auch Konzepte eines sicherheitsbewussten Softwareentwicklungsprozesses evaluiert (siehe [Abschnitt 8.5.3](#)). Weiterhin wurden unterschiedliche Verfahren zur Authentifizierung (Zwei-Faktor-Authentifizierung, Public-Key/Smartcard-Authentifizierung) gegenüber der GitHub-Plattform evaluiert (siehe [Abschnitt 8.5.4](#)).

9.2. Selbtskritik und aktuelle Probleme

Auch wenn die unter [Kapitel 3](#) gesetzten Anforderungen zum größten Teil als prototypischer Ansatz oder als Konzept umgesetzt wurden, sind bei der aktuellen Implementierung jedoch ein paar Aspekte nicht vernünftig gelöst oder könnten durch bessere Ansätze ergänzt werden.

Ein weiterhin bestehendes Problem ist die Umsetzung der Schlüsselgenerierung für das Verschlüsseln der Dateien in einem »brig«-Repository. Die aktuelle Implementierung erstellt zufallsgenerierte

Schlüssel. Dies hat den Nachteil, dass die Deduplizierungsfunktionalität außer Kraft gesetzt wird, siehe [Abschnitt 7.2.5.4](#). Hingegen würde die Verwendung von Convergent Encryption »brig« für bestimmte Angriffe, wie beispielsweise den Confirmation of a File-Angriff, anfällig machen. Die Empfehlung an dieser Stelle wäre, die Schlüsselgenerierung weiterhin auf Zufallsbasis zu realisieren und das dadurch entstandene Problem (IPFS kann Daten nicht mehr sinnvoll deduplizieren, siehe [Abschnitt 7.2.5.4](#)) in Kauf zu nehmen und eine abgemilderte Variante der Deduplizierung über Packfiles – wie von Herrn Pahl vorgeschlagen – zu realisieren (siehe [27], S. 101 f.).

Ein weiterer diskussionswürdiger Punkt ist die Verwendung von IPFS als Basis. Zwar erfüllt diese hier die benötigten Anforderungen, jedoch liegt der Fokus der Entwicklung des Projektes in erster Linie nicht im Bereich der Sicherheit. Aufgrund dieses Umstandes ist die Implementierung von Sicherheitsfeatures durch »brig« nicht zwangsläufig optimal. Die Implementierung bestimmter Sicherheits-Funktionalität wie beispielsweise Datenverschlüsselung wäre laut aktueller Einschätzung besser im IPFS-Backend zu realisieren. Weiterhin macht beispielsweise auch die seit Monaten andauernde Definition einer Spezifikation¹ für das IPFS-Keystore weitere Entwicklungsentscheidungen für »brig« schwierig.

Die Evaluation der Performance hat weiterhin gezeigt, dass beispielsweise die Geschwindigkeit mit den implementierten Algorithmen auf dem Raspberry Pi untragbar ist. Hier wäre eine Evaluation möglicher Optimierungen bezüglich der Geschwindigkeit nötig, wenn man den Raspberry Pi als geeignete Plattform ansehen möchte.

Weiterhin wurde die Entwicklung aktuell hauptsächlich problemorientiert vorangetrieben. Eine striktere Umsetzung der BSI-Richtlinien wäre hier sinnvoll, um die Software auch für den öffentlichen Bereich tauglich zu machen.

9.3. Ausblick

Die Evaluation der Sicherheit von »brig« sowie die evaluierten Sicherheitskonzepte stellen einen Erstentwurf für weitere Entwicklungen dar. Wie bereits unter [Kapitel 4](#) und [Kapitel 5](#) erläutert, ist die Implementierung und Evaluierung von Sicherheit keine triviale Aufgabe. Weiterhin kann auch die Umsetzung kryptographischer Elemente aufgrund von Missverständnissen fehlerhaft implementiert sein.

Da »brig« einen Schwerpunkt auf Sicherheit setzt, ist es essentiell, dass das in dieser Arbeit evaluierte System und die vorgestellten Konzepte von weiteren unabhängigen Sicherheitsexperten – beispielsweise von der HSASec² – beurteilen zu lassen und die vorgestellten Konzepte kritisch zu diskutieren. Zusätzlich sollte ein unabhängiges Sicherheitsaudit der Implementierung vor der ersten offiziellen Veröffentlichung durchgeführt werden.

Weiterhin wäre für anknüpfende Arbeiten im Bereich von »brig« oder allgemein von dezentralen und sicherheitsorientierten Dateiverteilungslösungen die Evaluation von Plattformen wie Keybase.io³ als Authentifizierungsplattform erwähnenswert.

¹Keystore Spezifikation: <https://github.com/ipfs/specs/tree/25411025e787e12b17f621fca25d636c5684316e/keystore>

²HSASec: <https://www.hsasec.de/>

³Keybase.io: <https://keybase.io/>

Im Bereich der Authentifizierung wäre eine Evaluation weiterer Authentifizierungsmöglichkeiten am »brig«-Repository, beispielsweise direkt über eine Smartcard–Authentifizierung, sinnvoll.

Die durchgeführten Benchmarks zur Geschwindigkeit zeigen nur eine Tendenz bezüglich der Geschwindigkeit auf den getesteten Plattformen. Hier wäre die Evaluation weiterer Hardware–Systeme nötig, um bessere Entscheidungen für Optimierungen und bezüglich der Auswahl geeigneter Algorithmen treffen zu können.

Die aktuelle Empfehlung ist es, auf Convergent Encryption zu verzichten. Einerseits kommt hier neue Angriffsfläche hinzu, andererseits besteht das Problem, dass bei der aktuellen Architektur die Daten zweimal gelesen werden müssten, da die Prüfsumme erst nach dem Hinzufügen in das IPFS–Backend bekannt ist. Hier wäre nichtsdestotrotz eine Evaluierung des unter [Abschnitt 8.1](#) erwähnten Ansatzes – Prüfsumme nur über wenige Bytes und Dateigröße – interessant.

Weiterhin bleibt neben einer benutzerfreundlichen Implementierung der vorgestellten Sicherheitskonzepte die Fragestellung der automatisierten Schlüsselsicherung offen – wie kann der Benutzer seine Identität auf einfache Art und Weise sichern?

Unabhängig davon ist es wichtig zu erwähnen, dass von den implementierten technischen Maßnahmen, der Mensch eine essentielle Rolle in jedem sicherheitskritischen System spielt. Diese Arbeit soll Software–Entwicklern als Einstiegspunkt für die Thematik der Sicherheit dienen und sie weiterhin für die sicherheitsorientierte Softwareentwicklung sensibilisieren.

Der Otto Normalbenutzer sollte immer wieder aufs Neue für das Thema Sicherheit und Datenschutz sensibilisiert werden, da gerade in unserer heutigen digitalen Informationsgesellschaft Fehleinschätzungen fatale Folgen haben können.

Prinzipiell kann gesagt werden, dass im begrenztem Zeitrahmen der Masterarbeit eine vorzeigbare Lösung zum dezentralen und sicheren Austausch von Daten entstanden ist. Auch wenn es nicht möglich war, in diesem kurzen Zeitraum weitere Fördermöglichkeiten zu finden, sollte die Entwicklung dennoch aufgrund des vielversprechenden Ansatzes weitergeführt werden. Hier wäre ein konstruktives Feedback von der Open–Source–Community und ein baldiges Release eines ersten Prototypen wichtig. Nicht nur, um den Bekanntheitsgrad weiterhin zu steigern, jedoch auch um in naher Zukunft eine ernsthafte und benutzerfreundliche Alternative zu den vorherrschenden Cloud–Speicher–Anbietern zu etablieren.

A

Umfang IPFS–Codebasis

Nach einem frischen git clone vom IPFS–Repository wurde der Umfang des Projekts ermittelt indem alle Abhängigkeiten mit x install --global=false beschafft wurden. Im Anschluss wurden alle erkenntlichen Drittanbieter–Bibliotheken in ein separates 3rd–Verzeichnis verschoben. Für die Analyse wurde das Werkzeug cloc¹ verwendet. Bei der Analyse wurden die autogenerierten Quelltextdateien (Protobuf, Endung pb.go) ausgeschlossen. Die Analyse wurde auf die in der Programmiersprache Go geschriebenen Teile begrenzt. Umfang IPFS–Projekt:

```
$ cloc $(find ./go-ipfs -iname '*.go' -type f | grep -v 'pb.go')  
 858 text files.  
 844 unique files.  
 14 files ignored.
```

github.com/AlDanial/cloc v 1.70 T=0.82 s (1026.5 files/s, 155950.9 lines/s)

Language	files	blank	comment	code
Go	844	19983	10605	97639
SUM:	844	19983	10605	97639

Umfang vom IPFS–Projekt genutzter Drittanbieter–Bibliotheken:

```
$ cloc $(find ./3rd -iname '*.go' -type f | \ grep -v 'pb.go')  
 1975 text files.  
 1742 unique files.  
 233 files ignored.
```

github.com/AlDanial/cloc v 1.70 T=6.02 s (289.3 files/s, 157793.5 lines/s)

Language	files	blank	comment	code
Go	1742	57033	70893	822177
SUM:	1742	57033	70893	822177

¹GitHub–Seite des Projektes: <https://github.com/AlDanial/cloc>

B

IPFS–Grundlagen

Die Initialisierung als IPFS-Einstiegspunkt (gekürzt):

```
$ ipfs cat /ipfs/QmYwAPJzv5CZsnA625s3Xf2nemtYgPpHdWEz79ojWnPbdG/readme
Hello and Welcome to IPFS!
```

[...]

If you're seeing this, you have successfully installed
IPFS and are now interfacing with the ipfs merkledag!

| Warning:

| This is alpha software. Use at your own discretion!
| Much is missing or lacking polish. There are bugs.
| Not yet secure. Read the security notes for more.

Check out some of the other files in this directory:

```
./about
./help
./quick-start      <-- usage examples
./readme          <-- this file
./security-notes
```

Sicherheitshinweis von IPFS:

```
$ ipfs cat /ipfs/QmYwAPJzv5CZsnA625s3Xf2nemtYgPpHdWEz79ojWnPbdG/security-notes
IPFS Alpha Security Notes
```

We try hard to ensure our system is safe and robust, but all software has bugs, especially new software. This distribution is meant to be an alpha preview, don't use it for anything mission critical.

Please note the following:

- This is alpha software and has not been audited. It is our goal to conduct a proper security audit once we close in on a 1.0 release.

- ipfs is a networked program, and may have serious undiscovered vulnerabilities. It is written in Go, and we do not execute any user provided data. But please point any problems out to us in a github issue, or email security@ipfs.io privately.
- ipfs uses encryption for all communication, but it's NOT PROVEN SECURE YET! It may be totally broken. For now, the code is included to make sure we benchmark our operations with encryption in mind. In the future, there will be an "**"unsafe"** mode for high performance intranet apps. If this is a blocking feature for you, please contact us.

IPFS-Entwickler:

- ▶ <https://github.com/whyrusleeping>
- ▶ <https://github.com/Kubuxu>

Brig-Entwickler:

- ▶ manny (Christoph Piechula)

IRC-Logauszug vom 14.10.2016

```
(17:06:38) manny: Hi, is ipfs using currently any encryption (like TLS) for data transport? Is there a spec?  
(17:08:15) Kubuxu: yes, we are using minimalistic subset of TLS, secio  
(17:10:27) manny: is there a spec online?  
(17:11:22) manny: secio?  
(17:11:24) whyrusleeping: manny: theres not a spec yet  
(17:11:27) whyrusleeping: the code is here: https://github.com/libp2p/go-libp2p-secio  
(17:11:33) manny: thx  
(17:11:52) whyrusleeping: its not a '1.0' type release, we are still going to be changing a couple things moving forward  
(17:12:04) whyrusleeping: and probably just straight up replace it with tls 1.3 once its more common  
(17:13:55) manny: As i never heard of secio, is it a known standard protocol - or something 'homemade'?  
(17:16:27) Kubuxu: it is based of one of the modes of TLS 1.2 but it is homemade  
(17:17:10) Kubuxu: it is based off one of TLS1.2 modes of operation but it is "homemade", that was the best option at the time  
(17:18:00) Kubuxu: we plan to move to TLS1.3 when it is available
```

D Details zur CPU-Architektur

System 1: Das erste System (Entwicklersystem Herr Piechula) ist ein Notebook mit Intel-i5-Architektur und AES-NI-Befehlserweiterungssatz. Der folgende `lscpu`-Ausschnitt zeigt die genauen Spezifikationen der CPU.

```
Architecture:           x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                4
On-line CPU(s) list:  0-3
Thread(s) per core:   2
Core(s) per socket:   2
Socket(s):             1
NUMA node(s):          1
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 58
Model name:            Intel(R) Core(TM) i5-3320M CPU @ 2.60GHz
Stepping:               9
CPU MHz:               1460.278
CPU max MHz:           3300.0000
CPU min MHz:           1200.0000
BogoMIPS:              5190.46
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:               256K
L3 cache:               3072K
NUMA node0 CPU(s):     0-3
Flags:                 fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca
                      cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx
                      rdtscp lm constant_tsc arch_perfmon pebs bts rep_good nopl xtopology
                      nonstop_tsc aperf mperf eagerfpu pni pclmulqdq dtes64 monitor ds_cpl vmx smx est
                      tm2 ssse3 cx16 xtpr pdcm pcid sse4_1 sse4_2 x2apic popcnt tsc_deadline_timer
                      aes xsave avx f16c rdrand lahf_lm epb tpr_shadow vnmi flexpriority ept vpid
                      fsgsbase smep erms xsaveopt dtherm ida arat pln pts
```

System 2: Das zweite System (Entwicklersystem Herr Pahl) ist ein Desktopsystem mit AMD-Phenom-X4-Architektur ohne AES-NI-Befehlserweiterungssatz. Der folgende lscpu-Ausschnitt zeigt die genauen Spezifikationen der CPU.

```

Architektur:          x86_64
CPU Operationsmodus: 32-bit, 64-bit
Byte-Reihenfolge:     Little Endian
CPU(s):               4
Liste der Online-CPU(s): 0-3
Thread(s) pro Kern:   1
Kern(e) pro Socket:  4
Sockel:               1
NUMA-Knoten:          1
Anbieterkennung:      AuthenticAMD
Prozessorfamilie:     16
Modell:                4
Modellname:            AMD Phenom(tm) II X4 955 Processor
Stepping:              2
CPU MHz:               2100.000
Maximale Taktfrequenz der CPU: 3200,0000
Minimale Taktfrequenz der CPU: 800,0000
BogoMIPS:              6432.74
Virtualisierung:       AMD-V
L1d Cache:             64K
L1i Cache:             64K
L2 Cache:              512K
L3 Cache:              6144K
NUMA-Knoten0 CPU(s):   0-3
Markierungen:          fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca
                      cmov pat pse36 clflush mmx fxsr sse sse2 ht syscall nx mmxext fxsr_opt pdpe1gb
                      rdtscp lm 3dnowext 3dnow constant_tsc rep_good nopl nonstop_tsc extd_apicid
                      eagerfpu pni monitor cx16 popcnt lahf_lm cmp_legacy svm extapic cr8_legacy abm
                      sse4a misalignsse 3dnowprefetch osvw ibs skinit wdt nodeid_msrs hw_pstate
                      vmmcall npt lbrv svm_lock nrip_save

```

System 3: Das erste der schwächeren Systeme ist ein Netbook auf Intel-Atom-Basis mit einer 32-Bit-CPU. Der folgende lscpu-Ausschnitt zeigt die genauen Spezifikationen der CPU.

```

Architecture:          i686
CPU op-mode(s):        32-bit
Byte Order:             Little Endian
CPU(s):                 2
On-line CPU(s) list:   0,1
Thread(s) per core:    2

```

```

Core(s) per socket:      1
Socket(s):              1
Vendor ID:               GenuineIntel
CPU family:              6
Model:                  28
Model name:              Intel(R) Atom(TM) CPU N270  @ 1.60GHz
Stepping:                2
CPU MHz:                 1600.000
CPU max MHz:             1600.0000
CPU min MHz:             800.0000
BogoMIPS:                3193.82
Flags:                   fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca
                           cmov pat clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe nx constant_tsc
                           arch_perfmon pebs bts aperfmpf perf pni dtes64 monitor ds_cpl est tm2 ssse3 xtpr
                           pdcm movbe lahf_lm dtherm

```

System 4: Das zweite der schwächeren Systeme ist ein Raspberry Pi Zero. Dieser hat die gleiche CPU wie die erste Version des Raspberry Pi, jedoch mit einer dynamisch erhöhten Frequenz. Der folgende lscpu-Ausschnitt zeigt die genauen Spezifikationen der CPU.

```

Architecture:           armv6l
Byte Order:              Little Endian
CPU(s):                  1
On-line CPU(s) list:    0
Thread(s) per core:     1
Core(s) per socket:      1
Socket(s):              1
Model name:              ARMv6-compatible processor rev 7 (v6l)
CPU max MHz:             1000.0000
CPU min MHz:             700.0000

```

Bash-Skript zur Ermittlung der Lese- und Schreibgeschwindigkeit mittels dd:

```
#/bin/sh

BENCHDIR=thisfoldershouldnotexist
TESTFILE=thisfileshouldnotexist

READACC=0
WRITEACC=0

function read_bench() {
    sudo echo 3 > /proc/sys/vm/drop_caches
    RESULT=`dd if=/boot/$TESTFILE of="$BENCHDIR/$TESTFILE" bs=1M count=256 \
    oflag=sync 2>&1 | tail -n 1| awk '{ gsub(",","."); print $(NF-1) }'` \
    READACC=`python3 -c "print($READACC+$RESULT)"` \
}

function write_bench() {
    sudo echo 3 > /proc/sys/vm/drop_caches
    RESULT=`dd if=$BENCHDIR/$TESTFILE of=/boot/$TESTFILE bs=1M count=256 \
    oflag=sync 2>&1 | tail -n 1| awk '{ gsub(",","."); print $(NF-1) }'` \
    WRITEACC=`python3 -c "print($WRITEACC+$RESULT)"` \
}

echo "Mounting benchmark dir..."
sudo swapoff -a
mkdir -p $BENCHDIR
sudo mount -t ramfs -o size=260M ramfs $BENCHDIR
sudo chmod 0777 $BENCHDIR
sudo dd if=/dev/urandom of=/boot/$TESTFILE bs=1M count=256 oflag=sync

echo "Running read benchmark..."
for i in `seq 1 10`;
do
    read_bench
done
echo Read performance: `python3 -c "print($READACC/10)"`
```

```

echo "Running write benchmark..."
for i in `seq 1 10`;
do
    write_bench
done
echo Write performance: `python3 -c "print($WRITEACC/10)"`"

sudo sync
sudo swapon -a
echo "Cleaning up..."
sudo umount $BENCHDIR
rmdir $BENCHDIR
sudo rm /boot/$TESTFILE

```

Kommandozeilen-Tool für Ver- und Entschlüsselung:

```

package main

import (
    "crypto/rand"
    "flag"
    "fmt"
    "io"
    "log"
    "os"
    "time"

    "github.com/disorganizer/brig/store/compress"
    "github.com/disorganizer/brig/store/encrypt"
    "golang.org/x/crypto/scrypt"
)

const (
    aeadCipherChaCha = iota
    aeadCipherAES
)

type options struct {
    zipalgo      string
    encalgo      string
    keyderiv    string
    output       string
    args         []string
}

```

```

        write          bool
        read           bool
        maxblocksize   int64
        useDevNull     bool
        forceDstOverwrite bool
    }

func withTime(fn func()) time.Duration {
    now := time.Now()
    fn()
    return time.Since(now)
}

func openDst(dest string, overwrite bool) *os.File {
    if !overwrite {
        if _, err := os.Stat(dest); !os.IsNotExist(err) {
            log.Fatalf("Opening destination failed, %v exists.\n", dest)
        }
    }

    fd, err := os.OpenFile(dest, os.O_CREATE|os.O_WRONLY|os.O_TRUNC, 0755)
    if err != nil {
        log.Fatalf("Opening destination %v failed: %v\n", dest, err)
    }
    return fd
}

func openSrc(src string) *os.File {
    fd, err := os.Open(src)
    if err != nil {
        log.Fatalf("Opening source %v failed: %v\n", src, err)
    }
    return fd
}

func dstFilename(compressor bool, src, algo string) string {
    if compressor {
        return fmt.Sprintf("%s.%s", src, algo)
    }
    return fmt.Sprintf("%s.%s", src, "uncompressed")
}

```

```

func dieWithUsage() {
    fmt.Printf("Usage of %s:\n", os.Args[0])
    flag.PrintDefaults()
    os.Exit(-1)

}

func die(err error) {
    log.Fatal(err)
    os.Exit(-1)
}

func parseFlags() options {
    read := flag.Bool("r", false, "Read mode.")
    write := flag.Bool("w", false, "Write mode.")
    maxblocksize := flag.Int64("b", 64*1024, "BlockSize.")
    zipalgo := flag.String( // just some formating to match appendix width.
        "c", "none", "Possible compression algorithms: none, snappy, lz4.",
    )
    output := flag.String(
        "o", "", "User defined output file destination."
    )
    encalgo := flag.String(
        "e", "aes", "Possible encryption algorithms: aes, chacha."
    )
    keyderiv := flag.String(
        "k", "none", "Use random or scrypt as key derivation: random, scrypt"
    )
    forceDstOverwrite := flag.Bool("f", false, "Force overwriting destination file.")
    useDevNull := flag.Bool("D", false, "Write to /dev/null.")
    flag.Parse()
    return options{
        read:          *read,
        write:         *write,
        zipalgo:       *zipalgo,
        encalgo:       *encalgo,
        output:        *output,
        keyderiv:      *keyderiv,
        maxblocksize:  *maxblocksize,
        forceDstOverwrite: *forceDstOverwrite,
        useDevNull:    *useDevNull,
        args:          flag.Args(),
    }
}

```

```

        }

}

func derivateAesKey(pwd, salt []byte, keyLen int) []byte {
    key, err := scrypt.Key(pwd, salt, 16384, 8, 1, keyLen)
    if err != nil {
        panic("Bad scrypt parameters: " + err.Error())
    }
    return key
}

func main() {
    opts := parseFlags()
    if len(opts.args) != 1 {
        dieWithUsage()
    }
    if opts.read && opts.write {
        dieWithUsage()
    }
    if !opts.read && !opts.write {
        dieWithUsage()
    }
}

srcPath := opts.args[0]
algo, err := compress.FromString(opts.zipalgo)
if err != nil {
    die(err)
}

src := openSrc(srcPath)
defer src.Close()

if opts.useDevNull && opts.output != "" {
    fmt.Printf(
        "%s\n", "dev/null (-D) and outputfile (-o) may not be set at the same time."
    )
    os.Exit(-1)
}

dstPath := dstFilename(opts.write, srcPath, opts.zipalgo)
if opts.useDevNull {
    dstPath = os.DevNull
}

```

```

}

if opts.output != "" {
    dstPath = opts.output
}

dst := openDst(dstPath, opts.forceDstOverwrite)
defer dst.Close()

var cipher uint16 = aeadCipherAES
if opts.encalgo == "chacha" {
    cipher = aeadCipherChaCha
}

if opts.encalgo == "aes" {
    cipher = aeadCipherAES
}

if opts.encalgo != "aes" && opts.encalgo != "chacha" && opts.encalgo != "none" {
    opts.encalgo = "none"
}

key := make([]byte, 32)

if opts.keyderiv == "scrypt" {
    fmt.Printf("%s\n", "Using scrypt key derivation.")
    key = deriveAesKey([]byte("defaultpassword"), nil, 32)
    if key == nil {
        die(err)
    }
}

if opts.keyderiv == "random" {
    fmt.Printf("%s\n", "Using random key derivation.")
    if _, err := io.ReadFull(rand.Reader, key); err != nil {
        die(err)
    }
}

// Writing
if opts.write {
    ew := io.WriteCloser(dst)
}

```

```

// Encryption is enabled
if opts.encalgo != "none" {
    ew, err = encrypt.NewWriterWithTypeAndBlockSize(
        dst, key, cipher, opts.maxblocksize
    )
    if err != nil {
        die(err)
    }
}
zw, err := compress.NewReader(ew, algo)
if err != nil {
    die(err)
}
_, err = io.Copy(zw, src)
if err != nil {
    die(err)
}
if err := zw.Close(); err != nil {
    die(err)
}
if err := ew.Close(); err != nil {
    die(err)
}
}

// Reading
if opts.read {
    var reader io.ReadSeeker = src
    // Decryption is enabled
    if opts.encalgo != "none" {
        er, err := encrypt.NewReader(src, key)
        if err != nil {
            die(err)
        }
        reader = er
    }
    zr := compress.NewReader(reader)
    _, err = io.Copy(dst, zr)
    if err != nil {
        die(err)
    }
}
}

```

Python Skript für die Erhebung der Benchmark-Daten im Hilfe des Kommandozeilen-Tools zur Ver- und Entschlüsselung von Dateien:

```

#!/usr/bin/env python
# encoding: utf-8

import sys, json, time, timeit, getpass, subprocess
from statistics import mean

BINARY="../data/main"

def get_write_dummy(filesize):
    return "./data/movie_{0}".format(filesize)

def get_read_dummy(filesize):
    return "./movie_{0}".format(filesize)

def get_blocksizes(filesize):
    return [64 * 1024]
    #return [(2**x) for x in range(30) if 2**x >= 64 and (2**x)/1024**2 <= filesize]

def benchmark_preprocessing(config):
    print(config)
    if config["type"] == "write":
        path = get_write_dummy(config["filesize"])

    if config["type"] == "read":
        path = get_read_dummy(config["filesize"])

    for cmd in [
        "mkdir -p data",
        "sudo mount -t ramfs -o size=2G ramfs data",
        "sudo chmod 0777 data",
        "sudo dd if=/dev/urandom of={0} bs=1M count={1} conv=sync".format(
            path, config["filesize"]),
        ),
        "sudo chown -R {user}:users data".format(user=getpass.getuser()),
        "go build main.go",
        "cp ./main ./data/main"
    ]:
        print(cmd)
        if subprocess.call(cmd.split(), shell=False) != 0:
            return -1, "Error occured during setup of read benchmark."

```

```

    return 0, ""

def teardown(data):
    cmd = ["sudo umount -l data", "sudo rm data -rv"]
    if data["type"] == "read":
        cmd += ["sudo rm {}".format(get_read_dummy(data["filesize"]))]
    for cmd in cmd:
        if subprocess.call(cmd.split(), shell=False) != 0:
            print("Error occurred during teardown.")
            sys.exit(-1)

def build_write_cmd(data, block):
    cmd = "{binary} -k {keyderiv} -w -b {block} -D -e {enc} -f {inputfile}".format(
        keyderiv=data["kgfunc"],
        binary=BINARY,
        block=block,
        enc=data["encryption"],
        inputfile=get_write_dummy(data["filesize"]))
    )
    return cmd.split()

def build_read_cmd(data, block):
    cmd = "{binary} -k {keyderiv} -r -b {block} -D -e {enc} -f {inputfile}".format(
        keyderiv=data["kgfunc"],
        binary=BINARY,
        block=block,
        enc=data["encryption"],
        inputfile=get_write_dummy(data["filesize"]))
    )
    return cmd.split()

def build_prepare_read_cmd(data, block):
    cmd = "{binary} -k {keyderiv} -w -b {block} -e {enc} -o {outputfile} -f {inputfile}\\".format(
        keyderiv=data["kgfunc"],
        binary=BINARY,
        block=block,
        enc=data["encryption"],
        inputfile=get_read_dummy(data["filesize"]),
        outputfile=get_write_dummy(data["filesize"]))
    )
    return cmd.split()

```

```

def write_bench_data(data):
    filename = "{sys}_{type}_{enc}_{zip}_{fs}_{kd}.json".format(
        sys=data["system"],
        type=data["type"],
        enc=data["encryption"],
        zip=data["compression"],
        fs=data["filesize"],
        kd=data["kgfunc"])
    .replace(" ", "_").replace("(", "[").replace(")", "]")
    with open(filename, "w") as fd:
        print("Writing {}".format(filename))
        fd.write(json.dumps(data))

def benchmark(data):
    print("** Running {} bench using {} runs. **".format(data["type"], data["runs"]))
    print("Parameters for this run: {}".format(data))

    if data["type"] != "read" and data["type"] != "write":
        return -1, "Preparing read cmd failed."

    for blocksize in get_blocksizes(data.get("filesize")):

        if data["type"] == "read":
            pre_cmd = build_prepare_read_cmd(data, blocksize)
            if subprocess.call(pre_cmd) != 0:
                return -1, "Preparing read cmd failed."

            plaincmd = build_read_cmd(data, blocksize)
            cmd = "subprocess.call({})".format(
                cmd=plaincmd
            )
            print("BS: {} \t CMD:{} ".format(blocksize, plaincmd))
            run = timeit.timeit(cmd, number=data["runs"], setup="import subprocess")

        if data["type"] == "write":
            cmd = "subprocess.call({})".format(
                cmd=build_write_cmd(data, blocksize)
            )
            print("{} bytes blocksize run...".format(blocksize))
            run = timeit.timeit(cmd, number=data["runs"], setup="import subprocess")

    data["results"].append(round(run / data["runs"] * 1000))

```

```
data["result-max"] = max(data["results"])
data["result-min"] = min(data["results"])
data["result-avg"] = mean(data["results"])
return 0, data

def initialize(algo, runs, filesize):
    run_data = []
    enc, title = config["algos"]
    if enc in ["aes", "chacha", "none"]:
        run_config = {
            "encryption": enc,
            "compression": "none",
            "title": title,
            "runs": runs,
            "results": [],
            "kgfunc": config["kgfunc"],
            "system": config["system"],
            "filesize": filesize,
            "type": config["type"]
        }
        run_data.append(run_config)
    return run_data

def run_benchmark(config, runs=5, filesize=128):
    bench_inputs = initialize(config, runs, filesize)
    for bench_input in bench_inputs:
        err, msg = benchmark_preprocessing(bench_input)
        if err != 0:
            print(msg)
            sys.exit(err)
        err, output = benchmark(bench_input)
        if err != 0:
            print(output)
            sys.exit(err)
        write_bench_data(output)
        teardown(bench_input)

    if __name__ == '__main__':
        pass
```

Python Skript für das Plotten der Benchmark-Daten:

```
#!/usr/bin/env python
# encoding: utf-8

import json, os, sys, pprint, subprocess, pygal.style, pygal
from math import log
from collections import OrderedDict
from collections import defaultdict
from statistics import mean

LEGEND_MAP = {
    'aesread': 'AES/GCM [R]',
    'aeswrite': 'AES/GCM [W]',
    'chacharead': 'ChaCha20/Poly1305 [R]',
    'chachawrite': 'ChaCha20/Poly1305 [W]',
    'noneread': 'Base (no crypto) [R]',
    'nonewrite': 'Base (no crypto) [W]'
}

LEGEND_MAP_SCRYPT = {
    'random': 'Dev Random generated key',
    'none': 'Static key',
    'scrypt': 'Scrypt generated key'
}

LEGEND_SYS_MAP = {
    "Intel i5 (Go 1.7.1)": "Intel i5",
    "Intel i5 (Go 1.5.3)": "Intel i5",
    "Intel i5 (Go 1.6)": "Intel",
    "AMD Phenom II X4 (Go 1.5.3)": "AMD Phenom",
    "AMD Phenom II X4 (Go 1.7.1)": "AMD Phenom",
    "ARM11 (Go 1.7.1)": "RPi Zero",
    "Intel Atom N270 SSE2 (Go 1.7.1)": "Intel Atom",
    "Intel Atom N270 387fpu (Go 1.7.1)": "Intel Atom (387FPU)"
}

def get_blocksizes(filesize):
    return [(2**x) for x in range(30) if 2**x >= 64 and (2**x)/1024**2 <= filesize]

# http://stackoverflow.com/questions/1094841/
# reusable-library-to-get-human-readable-version-of-file-size
def pretty_size(n,pow=0,b=1024,u='B',pre=['']+['i' for p in 'KMGTPEZY']):
    pass
```

```

pow,n=min(int(log(max(n*b**pow,1),b)),len(pre)-1),n*b**pow
return "%%.%if %%$%abs(pow%-pow-1))%($/b**float(pow),pre[pow],u)

def render_line_plot_scrypt(data):
    line_chart = pygal.Line(
        legend_at_bottom=True,
        logarithmic=data["logarithmic"],
        style=pygal.style.LightSolarizedStyle,
        x_label_rotation=25,
        interpolate='cubic'
    )
    filesizes = set()

    plot_data = data["plot-data"]
    for item in plot_data:
        filesizes.add(item["filesize"])

    line_chart.title = data["title"]
    line_chart.x_labels = [pretty_size(x * 1024**2) for x in sorted(list(filesizes))]
    line_chart.x_title = data["x-title"]
    line_chart.y_title = data["y-title"]

    plot_data = sorted(plot_data, key=lambda d: d["system"], reverse=False)

    sys1 = [
        s for s in plot_data if s["system"] == "Intel Keygen (Go 1.7.1)" and
        s["kgfunc"] == "scrypt"
    ]
    sys1 = sorted(sys1, key=lambda d: d["filesize"], reverse=False)

    sys2 = [
        s for s in plot_data if s["system"] == "Intel Keygen (Go 1.7.1)" and
        s["kgfunc"] == "random"
    ]
    sys2 = sorted(sys2, key=lambda d: d["filesize"], reverse=False)

    sys3 = [s for s in plot_data if s["system"] == "Intel Keygen (Go 1.7.1)" and
            s["kgfunc"] == "none"]
    sys3 = sorted(sys3, key=lambda d: d["filesize"], reverse=False)

    d1 = {}

```

```

for item in sys1 + sys2 + sys3:
    d1.setdefault(item["kgfunc"], []).append(item["results"])

for v in d1:
    line_chart.add(LEGEND_MAP_SCRYPT[v], [round(x.pop()) for x in d1[v]])
line_chart.render_to_file(data["outputfile"])

def format_min(values, min, filesize):
    values_str = []
    for v in values:
        val = str(v) + " ms; " + str(pretty_size((filesize) / (v/1000)) + "/s")
        if v == min:
            values_str.append("**" + val + "**")
        else:
            values_str.append(val)
    return values_str

def render_table(table, header, filesize):
    print("||" + "|".join([str(pretty_size(h)) + " [ms, B/s]" for h in header]) + "|")
    for title, row in table.items():
        print("|" + title + "|", end="")
        row = [999999 if v is None else v for v in row]
        n = format_min(row, min(row), filesize)
        print("|".join(n))

def render_line_plot(data):
    line_chart = pygal.Line(
        legend_at_bottom=True,
        logarithmic=data["logarithmic"],
        style=pygal.style.LightSolarizedStyle,
        x_label_rotation=25,
        interpolate='cubic'
    )
    line_chart.title = data["title"]
    line_chart.x_labels = [pretty_size(x) for x in get_blocksizes(
        data["needs"]["filesize"])]
    line_chart.x_title = data["x-title"]
    line_chart.y_title = data["y-title"]

    table = {}
    plot_data = data["plot-data"]

```

```

#print("|System| " + "|".join(x for x in line_chart.x_labels) + "|")
header = get_blocksizes(data["needs"]["filesize"])
for item in plot_data:
    avg_sec = mean(item["results"]) / 1000
    fs_bytes = megabytes_to_bytes(item["filesize"])
    avg_mb_sec = round(fs_bytes/avg_sec, 2)
    op = item["type"][0]
    title = LEGEND_SYS_MAP[item["system"]] + " (" + LEGEND_MAP[item["encryption"]] + \
            item["type"] + ")"
    table[title]= item["results"]
    line_chart.add(title, item["results"])
    line_chart.render_to_file(data["outputfile"])
render_table(table, header, fs_bytes)

def megabytes_to_bytes(bytes):
    return bytes * 1024 ** 2

def render_bar_plot(data):
    line_chart = pygal.HorizontalBar(
        print_values=True,
        value_formatter=lambda x: '{}/s'.format(pretty_size(x)),
        truncate_legend=220,
        legend_at_bottom=True,
        logarithmic=data["logarithmic"],
        style=pygal.style.LightSolarizedStyle,
        x_label_rotation=60,
        y_label_rotation=300,
        interpolate='cubic'
    )
    line_chart.title = data["title"]

    plot_data = data["plot-data"]
    plot_data = sorted(
        plot_data, key=lambda d: d["system"] + d["encryption"] + d["type"], reverse=False
    )

    line_chart.x_labels = list(sorted(set([x["system"] for x in plot_data])))
    line_chart.x_title = data["x-title"]
    line_chart.y_title = data["y-title"]

    d = {}
    for item in plot_data:

```

```

# mbytes to bytes, and msecounds to seconds
fs = item["filesize"] * 1024**2
s = item["results"][10] / 1000
d.setdefault(item["encryption"] + item["type"], []).append(fs/s)
print(item["filesize"], min(item["results"])/1000)

for k in sorted(d):
    line_chart.add(LEGEND_MAP[k], [round(v) for v in d[k]])
line_chart.render_to_file(data["outputfile"])

def is_valid(jdir, metadata):
    if jdir.get("system") not in metadata["needs"]["system"]:
        return False

    if jdir.get("type") not in metadata["needs"]["type"]:
        return False

    if jdir.get("encryption") not in metadata["needs"]["algo"]:
        return False

    return True

def get_input_data(path):
    with open(path, 'r') as fd:
        metadata = json.loads(fd.read())
        metadata["input-data"] = os.path.abspath(metadata["input-data"])

    benchmark_files = []
    for file in sorted(os.listdir(metadata["input-data"])):
        jfile = os.path.abspath(os.path.join(metadata["input-data"], file))
        if not os.path.isdir(jfile) and jfile.endswith("json"):
            with open(jfile, "r") as fd:
                benchmark_files.append(json.loads(fd.read()))

    needed_files = []
    for jfile in benchmark_files:
        if is_valid(jfile, metadata):
            metadata["plot-data"].append(jfile)
    return metadata

if __name__ == '__main__':

```

```
config_path = os.path.abspath(sys.argv[1])
dir_path = os.path.dirname(config_path)
input_data = get_input_data(config_path)
input_data["outputfile"] = os.path.join(
    dir_path, os.path.basename(config_path) + ".svg"
)

if input_data["plot-data"] == []:
    print("No Plot data found with this attributes.")
    sys.exit(0)

if input_data["type"] == "line":
    render_line_plot(input_data)

if input_data["type"] == "bar":
    render_bar_plot(input_data)

if input_data["type"] == "scrypt":
    render_line_plot_scrypt(input_data)

subprocess.call(
    [
        "inkscape",
        "{0}".format(input_data["outputfile"]),
        "--export-pdf={0}.pdf".format(input_data["outputfile"])
    ]
)
subprocess.call(
    ["chromium", "{0}".format(input_data["outputfile"]), "&"]
)
```

```
gpg/card> admin
Admin commands are allowed

gpg/card> generate
Make off-card backup of encryption key? (Y/n) Y
gpg: error checking the PIN: Card error

gpg/card> generate
Make off-card backup of encryption key? (Y/n) Y
Please specify how long the key should be valid.
    0 = key does not expire
    <n> = key expires in n days
    <n>w = key expires in n weeks
    <n>m = key expires in n months
    <n>y = key expires in n years
Key is valid for? (0) 5y
Key expires at Fr 10 Dez 2021 13:44:18 CET
Is this correct? (y/N) y

GnuPG needs to construct a user ID to identify your key.

Real name: Christoph Piechula
Email address: christoph@nullcat.de
Comment:
You selected this USER-ID:
    "Christoph Piechula <christoph@nullcat.de>"

Change (N)ame, (C)omment, (E)mail or (O)kay/(Q)uit? o
We need to generate a lot of random bytes. It is a good idea to perform
some other action (type on the keyboard, move the mouse, utilize the
disks) during the prime generation; this gives the random number
generator a better chance to gain enough entropy.
gpg: Note: backup of card key saved to '/home/qitta/.gnupg/sk_E5A1965037A8E37C.gpg'
gpg: key 932AEBFDD72FE59C marked as ultimately trusted
gpg: revocation certificate stored as '/home/qitta/.gnupg/openpgp-revocs.d/ \
D61CEE19369B9C330A4A482D932AEBFDD72FE59C.rev'
public and secret key created and signed.
```

Generierte Schlüssel anzeigen lassen

```
gpg/card> list
```

```
Reader .....: 0000:0000:X:0
Application ID ....: 00000000000000000000000000000000
Version .....: 2.0
Manufacturer .....: Yubico
Serial number ....: 00000000
Name of cardholder: Christoph Piechula
Language prefs ....: [not set]
Sex .....: male
URL of public key : [not set]
Login data .....: [not set]
Signature PIN ....: forced
Key attributes ....: rsa2048 rsa2048 rsa2048
Max. PIN lengths ..: 127 127 127
PIN retry counter : 3 3 3
Signature counter : 4
Signature key ....: D61C EE19 369B 9C33 0A4A 482D 932A EBFD D72F E59C
                   created ....: 2016-12-11 12:44:36
Encryption key.....: DD5E 14EE D04D 58AB 85D7 0AB3 E5A1 9650 37A8 E37C
                   created ....: 2016-12-11 12:44:36
Authentication key: 4E45 FC88 A1B1 292F 6CFA B577 4CE8 E35B 8002 9F6E
                   created ....: 2016-12-11 12:44:36
General key info...: pub rsa2048/932AEBFDD72FE59C 2016-12-11 \
                     Christoph Piechula <christoph@nullcat.de>
sec>   rsa2048/932AEBFDD72FE59C created: 2016-12-11 expires: 2021-12-10
                  card-no: 0006 00000000
ssb>   rsa2048/4CE8E35B80029F6E created: 2016-12-11 expires: 2021-12-10
                  card-no: 0006 00000000
ssb>   rsa2048/E5A1965037A8E37C created: 2016-12-11 expires: 2021-12-10
                  card-no: 0006 00000000
```

```
gpg/card>
```

Schlüssel mit GnuPG anzeigen lassen:

```
$ gpg2 --list-keys 932AEBFDD72FE59C
pub    rsa2048 2016-12-11 [SC] [expires: 2021-12-10]
      D61CEE19369B9C330A4A482D932AEBFDD72FE59C
uid          [ultimate] Christoph Piechula <christoph@nullcat.de>
sub    rsa2048 2016-12-11 [A] [expires: 2021-12-10]
sub    rsa2048 2016-12-11 [E] [expires: 2021-12-10]
```

```
$ gpg2 --list-secret-keys 932AEBFDD72FE59C
sec> rsa2048 2016-12-11 [SC] [expires: 2021-12-10]
      D61CEE19369B9C330A4A482D932AEBFDD72FE59C
      Card serial no. = 0006 00000000
uid          [ultimate] Christoph Piechula <christoph@nullcat.de>
ssb> rsa2048 2016-12-11 [A] [expires: 2021-12-10]
ssb> rsa2048 2016-12-11 [E] [expires: 2021-12-10]
```

```
$ gpg2 --expert --edit-key E9CD5AB4075551F6F1D6AE918219B30B103FB091
gpg (GnuPG) 2.1.16; Copyright (C) 2016 Free Software Foundation, Inc.
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
```

Secret key is available.

```
sec  rsa2048/8219B30B103FB091
      created: 2013-02-09  expires: 2017-01-31  usage: SC
      trust: ultimate      validity: ultimate
ssb  rsa2048/0B81E5BF85821570
      created: 2013-02-09  expires: 2017-01-31  usage: E
[ultimate] (1). Christoph Piechula <christoph@nullcat.de>
```

gpg> addkey

Please select what kind of key you want:

- (3) DSA (sign only)
- (4) RSA (sign only)
- (5) Elgamal (encrypt only)
- (6) RSA (encrypt only)
- (7) DSA (set your own capabilities)
- (8) RSA (set your own capabilities)
- (10) ECC (sign only)
- (11) ECC (set your own capabilities)
- (12) ECC (encrypt only)
- (13) Existing key

Your selection? 4

RSA keys may be between 1024 and 4096 bits long.

What keysize do you want? (2048)

Requested keysize is 2048 bits

Please specify how long the key should be valid.

- 0 = key does not expire
- <n> = key expires in n days
- <n>w = key expires in n weeks
- <n>m = key expires in n months
- <n>y = key expires in n years

```
Key is valid for? (0) 2y
Key expires at Di 11 Dez 2018 17:33:54 CET
Is this correct? (y/N) y
Really create? (y/N) y
We need to generate a lot of random bytes. It is a good idea to perform
some other action (type on the keyboard, move the mouse, utilize the
disks) during the prime generation; this gives the random number
generator a better chance to gain enough entropy.
```

```
sec rsa2048/8219B30B103FB091
    created: 2013-02-09  expires: 2017-01-31  usage: SC
        trust: ultimate      validity: ultimate
ssb rsa2048/0B81E5BF85821570
    created: 2013-02-09  expires: 2017-01-31  usage: E
ssb rsa2048/2CC4F84BE43F54ED
    created: 2016-12-11  expires: 2018-12-11  usage: S
[ultimate] (1). Christoph Piechula <christoph@nullcat.de>
```

```
gpg> addkey
Please select what kind of key you want:
 (3) DSA (sign only)
 (4) RSA (sign only)
 (5) Elgamal (encrypt only)
 (6) RSA (encrypt only)
 (7) DSA (set your own capabilities)
 (8) RSA (set your own capabilities)
 (10) ECC (sign only)
 (11) ECC (set your own capabilities)
 (12) ECC (encrypt only)
 (13) Existing key
Your selection? 8
```

```
Possible actions for a RSA key: Sign Encrypt Authenticate
Current allowed actions: Sign Encrypt
```

```
(S) Toggle the sign capability
(E) Toggle the encrypt capability
(A) Toggle the authenticate capability
(Q) Finished
```

```
Your selection? e
```

Possible actions for a RSA key: Sign Encrypt Authenticate

Current allowed actions: Sign

- (S) Toggle the sign capability
- (E) Toggle the encrypt capability
- (A) Toggle the authenticate capability
- (Q) Finished

Your selection? s

Possible actions for a RSA key: Sign Encrypt Authenticate

Current allowed actions:

- (S) Toggle the sign capability
- (E) Toggle the encrypt capability
- (A) Toggle the authenticate capability
- (Q) Finished

Your selection? a

Possible actions for a RSA key: Sign Encrypt Authenticate

Current allowed actions: Authenticate

- (S) Toggle the sign capability
- (E) Toggle the encrypt capability
- (A) Toggle the authenticate capability
- (Q) Finished

Your selection? q

RSA keys may be between 1024 and 4096 bits long.

What keysize do you want? (2048)

Requested keysize is 2048 bits

Please specify how long the key should be valid.

0 = key does not expire

<n> = key expires in n days

<n>w = key expires in n weeks

<n>m = key expires in n months

<n>y = key expires in n years

Key is valid for? (0) 2y

Key expires at Di 11 Dez 2018 17:35:18 CET

Is this correct? (y/N) y

Really create? (y/N) y

We need to generate a lot of random bytes. It is a good idea to perform some other action (type on the keyboard, move the mouse, utilize the disks) during the prime generation; this gives the random number generator a better chance to gain enough entropy.

```
sec rsa2048/8219B30B103FB091
    created: 2013-02-09  expires: 2017-01-31  usage: SC
        trust: ultimate      validity: ultimate
ssb rsa2048/0B81E5BF85821570
    created: 2013-02-09  expires: 2017-01-31  usage: E
ssb rsa2048/2CC4F84BE43F54ED
    created: 2016-12-11  expires: 2018-12-11  usage: S
ssb rsa2048/74B050CC5ED64D18
    created: 2016-12-11  expires: 2018-12-11  usage: A
[ultimate] (1). Christoph Piechula <christoph@nullcat.de>
```

```
gpg> save
```

H

Ablaufdatum ändern

Ablaufdatum für den Hauptschlüssel und für den Unterschlüssel zum Ver-/Entschlüsseln ändern:

```
$ gpg2 --expert --edit-key E9CD5AB4075551F6F1D6AE918219B30B103FB091
gpg (GnuPG) 2.1.16; Copyright (C) 2016 Free Software Foundation, Inc.
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
```

Secret key is available.

```
sec  rsa2048/8219B30B103FB091
      created: 2013-02-09  expires: 2017-01-31  usage: SC
      trust: ultimate      validity: ultimate
ssb  rsa2048/0B81E5BF85821570
      created: 2013-02-09  expires: 2017-01-31  usage: E
ssb  rsa2048/2CC4F84BE43F54ED
      created: 2016-12-11  expires: 2018-12-11  usage: S
ssb  rsa2048/74B050CC5ED64D18
      created: 2016-12-11  expires: 2018-12-11  usage: A
[ultimate] (1). Christoph Piechula <christoph@nullcat.de>
```

gpg> expire

Changing expiration time for the primary key.

Please specify how long the key should be valid.

0 = key does not expire

<n> = key expires in n days

<n>w = key expires in n weeks

<n>m = key expires in n months

<n>y = key expires in n years

Key is valid for? (0) 10y

Key expires at Mi 09 Dez 2026 17:45:52 CET

Is this correct? (y/N) y

```
sec  rsa2048/8219B30B103FB091
      created: 2013-02-09  expires: 2026-12-09  usage: SC
      trust: ultimate      validity: ultimate
ssb  rsa2048/0B81E5BF85821570
      created: 2013-02-09  expires: 2017-01-31  usage: E
ssb  rsa2048/2CC4F84BE43F54ED
```

```

    created: 2016-12-11  expires: 2018-12-11  usage: S
ssb  rsa2048/74B050CC5ED64D18
    created: 2016-12-11  expires: 2018-12-11  usage: A
[ultimate] (1). Christoph Piechula <christoph@nullcat.de>
```

gpg> key 1

```

sec  rsa2048/8219B30B103FB091
    created: 2013-02-09  expires: 2026-12-09  usage: SC
    trust: ultimate      validity: ultimate
ssb* rsa2048/0B81E5BF85821570
    created: 2013-02-09  expires: 2017-01-31  usage: E
ssb  rsa2048/2CC4F84BE43F54ED
    created: 2016-12-11  expires: 2018-12-11  usage: S
ssb  rsa2048/74B050CC5ED64D18
    created: 2016-12-11  expires: 2018-12-11  usage: A
[ultimate] (1). Christoph Piechula <christoph@nullcat.de>
```

gpg> expire

Changing expiration time for a subkey.

Please specify how long the key should be valid.

0 = key does not expire

<n> = key expires in n days

<n>w = key expires in n weeks

<n>m = key expires in n months

<n>y = key expires in n years

Key is valid for? (0) 2y

Key expires at Di 11 Dez 2018 17:46:03 CET

Is this correct? (y/N) y

```

sec  rsa2048/8219B30B103FB091
    created: 2013-02-09  expires: 2026-12-09  usage: SC
    trust: ultimate      validity: ultimate
ssb* rsa2048/0B81E5BF85821570
    created: 2013-02-09  expires: 2018-12-11  usage: E
ssb  rsa2048/2CC4F84BE43F54ED
    created: 2016-12-11  expires: 2018-12-11  usage: S
ssb  rsa2048/74B050CC5ED64D18
    created: 2016-12-11  expires: 2018-12-11  usage: A
[ultimate] (1). Christoph Piechula <christoph@nullcat.de>
```

gpg> save

I

Exportieren der GnuPG-Schlüssel

Exportieren der privaten Schlüssel:

```
$ gpg2 --armor --export E9CD5AB4075551F6F1D6AE918219B30B103FB091 \
> E9CD5AB4075551F6F1D6AE918219B30B103FB091.pub
$ gpg2 --armor --export-secret-keys \
> E9CD5AB4075551F6F1D6AE918219B30B103FB091.sec
$ gpg2 --armor --export-secret-subkeys \
> E9CD5AB4075551F6F1D6AE918219B30B103FB091.secsub
```

```
$ gpg2 --expert --edit-key E9CD5AB4075551F6F1D6AE918219B30B103FB091
gpg (GnuPG) 2.1.16; Copyright (C) 2016 Free Software Foundation, Inc.
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
```

Secret key is available.

```
sec  rsa2048/8219B30B103FB091
      created: 2013-02-09  expires: 2026-12-09  usage: SC
      trust: ultimate      validity: ultimate
ssb  rsa2048/0B81E5BF85821570
      created: 2013-02-09  expires: 2018-12-11  usage: E
ssb  rsa2048/2CC4F84BE43F54ED
      created: 2016-12-11  expires: 2018-12-11  usage: S
ssb  rsa2048/74B050CC5ED64D18
      created: 2016-12-11  expires: 2018-12-11  usage: A
[ultimate] (1). Christoph Piechula <christoph@nullcat.de>
```

gpg> key 1

```
sec  rsa2048/8219B30B103FB091
      created: 2013-02-09  expires: 2026-12-09  usage: SC
      trust: ultimate      validity: ultimate
ssb* rsa2048/0B81E5BF85821570
      created: 2013-02-09  expires: 2018-12-11  usage: E
ssb  rsa2048/2CC4F84BE43F54ED
      created: 2016-12-11  expires: 2018-12-11  usage: S
ssb  rsa2048/74B050CC5ED64D18
      created: 2016-12-11  expires: 2018-12-11  usage: A
[ultimate] (1). Christoph Piechula <christoph@nullcat.de>
```

gpg> keytocard

Please select where to store the key:

(2) Encryption key

Your selection? 2

```
sec  rsa2048/8219B30B103FB091
```

```
        created: 2013-02-09  expires: 2026-12-09  usage: SC
          trust: ultimate      validity: ultimate
ssb* rsa2048/0B81E5BF85821570
        created: 2013-02-09  expires: 2018-12-11  usage: E
ssb  rsa2048/2CC4F84BE43F54ED
        created: 2016-12-11  expires: 2018-12-11  usage: S
ssb  rsa2048/74B050CC5ED64D18
        created: 2016-12-11  expires: 2018-12-11  usage: A
[ultimate] (1). Christoph Piechula <christoph@nullcat.de>
```

```
gpg> key 2
```

```
sec  rsa2048/8219B30B103FB091
        created: 2013-02-09  expires: 2026-12-09  usage: SC
          trust: ultimate      validity: ultimate
ssb* rsa2048/0B81E5BF85821570
        created: 2013-02-09  expires: 2018-12-11  usage: E
ssb* rsa2048/2CC4F84BE43F54ED
        created: 2016-12-11  expires: 2018-12-11  usage: S
ssb  rsa2048/74B050CC5ED64D18
        created: 2016-12-11  expires: 2018-12-11  usage: A
[ultimate] (1). Christoph Piechula <christoph@nullcat.de>
```

```
gpg> keytocard
```

You must select exactly one key.

```
gpg> key 1
```

```
sec  rsa2048/8219B30B103FB091
        created: 2013-02-09  expires: 2026-12-09  usage: SC
          trust: ultimate      validity: ultimate
ssb  rsa2048/0B81E5BF85821570
        created: 2013-02-09  expires: 2018-12-11  usage: E
ssb* rsa2048/2CC4F84BE43F54ED
        created: 2016-12-11  expires: 2018-12-11  usage: S
ssb  rsa2048/74B050CC5ED64D18
        created: 2016-12-11  expires: 2018-12-11  usage: A
[ultimate] (1). Christoph Piechula <christoph@nullcat.de>
```

```
gpg> keytocard
```

Please select where to store the key:

(1) Signature key

(3) Authentication key

Your selection? 1

```
sec rsa2048/8219B30B103FB091
    created: 2013-02-09  expires: 2026-12-09  usage: SC
    trust: ultimate      validity: ultimate
ssb  rsa2048/0B81E5BF85821570
    created: 2013-02-09  expires: 2018-12-11  usage: E
ssb* rsa2048/2CC4F84BE43F54ED
    created: 2016-12-11  expires: 2018-12-11  usage: S
ssb  rsa2048/74B050CC5ED64D18
    created: 2016-12-11  expires: 2018-12-11  usage: A
[ultimate] (1). Christoph Piechula <christoph@nullcat.de>
```

gpg> key 2

```
sec rsa2048/8219B30B103FB091
    created: 2013-02-09  expires: 2026-12-09  usage: SC
    trust: ultimate      validity: ultimate
ssb  rsa2048/0B81E5BF85821570
    created: 2013-02-09  expires: 2018-12-11  usage: E
ssb* rsa2048/2CC4F84BE43F54ED
    created: 2016-12-11  expires: 2018-12-11  usage: S
ssb  rsa2048/74B050CC5ED64D18
    created: 2016-12-11  expires: 2018-12-11  usage: A
[ultimate] (1). Christoph Piechula <christoph@nullcat.de>
```

gpg> key 3

```
sec rsa2048/8219B30B103FB091
    created: 2013-02-09  expires: 2026-12-09  usage: SC
    trust: ultimate      validity: ultimate
ssb  rsa2048/0B81E5BF85821570
    created: 2013-02-09  expires: 2018-12-11  usage: E
ssb* rsa2048/2CC4F84BE43F54ED
    created: 2016-12-11  expires: 2018-12-11  usage: S
ssb* rsa2048/74B050CC5ED64D18
    created: 2016-12-11  expires: 2018-12-11  usage: A
[ultimate] (1). Christoph Piechula <christoph@nullcat.de>
```

gpg> keytocard

Please select where to store the key:

(3) Authentication key

Your selection? 3

```
sec  rsa2048/8219B30B103FB091
    created: 2013-02-09  expires: 2026-12-09  usage: SC
    trust: ultimate      validity: ultimate
ssb  rsa2048/0B81E5BF85821570
    created: 2013-02-09  expires: 2018-12-11  usage: E
ssb  rsa2048/2CC4F84BE43F54ED
    created: 2016-12-11  expires: 2018-12-11  usage: S
ssb* rsa2048/74B050CC5ED64D18
    created: 2016-12-11  expires: 2018-12-11  usage: A
[ultimate] (1). Christoph Piechula <christoph@nullcat.de>
```

gpg> save

```
$ gpg2 --card-edit

Reader .....: 0000:0000:X:0
Application ID ...: 00000000000000000000000000000000
Version .....: 2.0
Manufacturer ....: Yubico
Serial number ....: 00000000
Name of cardholder: [not set]
Language prefs ....: [not set]
Sex .....: unspecified
URL of public key : [not set]
Login data .....: [not set]
Signature PIN ....: forced
Key attributes ...: rsa2048 rsa2048 rsa2048
Max. PIN lengths ..: 127 127 127
PIN retry counter : 3 3 3
Signature counter : 1
Signature key ....: 7CD8 DB88 FBF8 22E1 3005 66D1 2CC4 F84B E43F 54ED
                   created ....: 2016-12-11 16:32:58
Encryption key....: 6258 6E4C D843 F566 0488 0EB0 0B81 E5BF 8582 1570
                   created ....: 2013-02-09 23:18:50
Authentication key: 2BC3 8804 4699 B83F DEA0 A323 74B0 50CC 5ED6 4D18
                   created ....: 2016-12-11 16:34:21
General key info...: sub rsa2048/2CC4F84BE43F54ED 2016-12-11 \
                     Christoph Piechula <christoph@nullcat.de>
sec   rsa2048/8219B30B103FB091 created: 2013-02-09 expires: 2026-12-09
ssb>  rsa2048/0B81E5BF85821570 created: 2013-02-09 expires: 2018-12-11
                  card-no: 0006 00000000
ssb>  rsa2048/2CC4F84BE43F54ED created: 2016-12-11 expires: 2018-12-11
                  card-no: 0006 00000000
ssb>  rsa2048/74B050CC5ED64D18 created: 2016-12-11 expires: 2018-12-11
                  card-no: 0006 00000000

gpg/card> admin
Admin commands are allowed

gpg/card> passwd
```

```
gpg: OpenPGP card no. 00000000000000000000000000000000 detected
```

```
1 - change PIN  
2 - unblock PIN  
3 - change Admin PIN  
4 - set the Reset Code  
Q - quit
```

```
Your selection? 1
```

```
PIN changed.
```

```
1 - change PIN  
2 - unblock PIN  
3 - change Admin PIN  
4 - set the Reset Code  
Q - quit
```

```
Your selection? 3
```

```
PIN changed.
```

```
1 - change PIN  
2 - unblock PIN  
3 - change Admin PIN  
4 - set the Reset Code  
Q - quit
```

```
Your selection? Q
```

```
gpg/card> quit
```

Das folgende Code-Snippet zeigt den Quellcode zum Generieren des »brig« QR-Codes:

```
package main

import (
    "fmt"
    "os"

qrcode "github.com/skip2/go-qrcode"
)

func main() {
    t := os.Args[1] // On cli: "your qr code string"
    if err := qrcode.WriteFile(t, qrcode.Highest, 256, "qr.png"); err != nil {
        fmt.Printf("%v\n", err)
    }
}
```

Das folgende Code-Snippet zeigt nur eine vereinfachte Proof-of-Concept-Implementierung. Im Produktiv-Code müssten beispielsweise Daten wie Passwort und YubikeyID mittels einer Passwortableitungsfunktion verwaltet werden.

```
package main

import (
    "fmt"
    "log"
    "os"

    "github.com/GeertJohan/yubigo"
    "github.com/fatih/color"
)

const (
    RegisteredYubikeyID = "cccccccelefli"
    UserPassword        = "katzenbaum"
)

func boolToAnswer(answer bool) string {
    if answer {
        return color.GreenString("OK")
    }
    return color.RedString("X")
}

func main() {
    password, otp := os.Args[1], os.Args[2]
    yubiAuth, err := yubigo.NewYubiAuth("00000", "00000000000000000000000000000000=")

    if err != nil {
        log.Fatalln(err)
    }

    // Validierung an der Yubico Cloud
    _, ok, err := yubiAuth.Verify(otp)
    if err != nil {
```

```
    log.Println(err)
}

fmt.Printf("[yubico: %s, ", boolToAnswer(ok))
fmt.Printf("brig : %s, ", boolToAnswer(otp[0:12] == RegisteredYubikeyID))
fmt.Printf("password: %s]\n", boolToAnswer(password == UserPassword))
}
```

Literaturverzeichnis

- [1] G. Greenwald and E. MacAskill, “NSA Prism program taps in to user data of Apple, Google and others,” *The Guardian*. <https://www.theguardian.com/world/2013/jun/06/us-tech-giants-nsa-data>, 2013.
- [2] M. Borgmann, T. Hahn, M. Herfert, T. Kunz, M. Richter, U. Viebeg, and S. Vowe, “On the security of cloud storage services.” https://www.sit.fraunhofer.de/fileadmin/dokumente/studien_und_technical_reports/Cloud-Storage-Security_a4.pdf, 2012.
- [3] C. S. Peter Mahlmann, *Peer-to-Peer-Netzwerke*. eXamen.press, 2007.
- [4] C. Pauly, *Netzwerk- und Internetsicherheitstechnologien in der IT-basierten Unternehmensberatung*. Diplom.de, 2004.
- [5] R. Ko and R. Choo, *The Cloud Security Ecosystem: Technical, Legal, Business and Management Issues*. Elsevier Science, 2015.
- [6] M. Mulazzani, S. Schrittwieser, M. Leithner, M. Huber, and E. R. Weippl, “Dark Clouds on the Horizon: Using Cloud Storage as Attack Vector and Online Slack Space.” San Francisco, CA, USA; https://www.usenix.org/legacy/event/sec11/tech/full_papers/Mulazzani.pdf, 2011.
- [7] D. Kholia and P. Węgrzyn, “Looking inside the (Drop) box.” <https://www.usenix.org/system/files/conference/woot13/woot13-kholia.pdf>, 2013.
- [8] IMPERVA - HACKER INTELLIGENCE INITIATIVE, “Man in the Cloud (MITC) Attacks.” https://www.imperva.com/docs/HII_Man_In_The_Cloud_Attacks.pdf, 2015.
- [9] C. Paar and J. Pelzl, *Understanding Cryptography: A Textbook for Students and Practitioners*. Springer Berlin Heidelberg, 2009.
- [10] W. Ertel, *Angewandte Kryptographie*. Carl Hanser Verlag GmbH & Company KG, 2012.
- [11] R. R. Ponemon Institute, “Cost of Data Breach Study: Global Analysis.” <https://public.dhe.ibm.com/common/ssi/ecm/se/en/sel03094wwen/SEL03094WWEN.PDF>, 2016.
- [12] J. Quintard, “Towards a worldwide storage infrastructure.” University of Cambridge; <https://www.repository.cam.ac.uk/bitstream/handle/1810/243442/thesis.pdf>, 2012.
- [13] G. Schryen, “Is Open Source Security a Myth?” *Commun. ACM*. <http://doi.acm.org/10.1145/1941487.1941516>; ACM, 2011.
- [14] D. Spinellis, “A Tale of Four Kernels.” <http://doi.acm.org/10.1145/1368088.1368140>; ACM, 2008.
- [15] Synopsys, Inc., “Coverity Scan – Open Source Report 2013.” <http://softwareintegrity.coverity.com/rs/appsec/images/2013-Coverity-Scan-Report.pdf>, 2014.

- [16] Synopsys, Inc., “Coverity Scan – Open Source Report 2014.” <http://go.coverity.com/rs/157-LQW-289/images/2014-Coverity-Scan-Report.pdf>, 2015.
- [17] H. Bleich, “Nichts zu verbergen?” *c't* 17/2015. 2015.
- [18] M. Y. C. Wei, L. M. Grupp, F. E. Spada, and S. Swanson, “Reliably Erasing Data from Flash-Based Solid State Drives.” <https://cseweb.ucsd.edu/~swanson/papers/Fast2011SecErase.pdf>, 2011.
- [19] J. Nielsen, “Windows 8—disappointing usability for both novice and power users.” <https://www.nngroup.com/articles/windows-8-disappointing-usability/>, 2012.
- [20] Bundesamt für Sicherheit in der Informationstechnik (BSI), “BSI – Technische Richtlinie; Kryptographische Verfahren: Empfehlungen und Schlüssellängen.” <https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Publikationen/TechnischeRichtlinien/TR02102/BSI-TR-02102.pdf>, 2016.
- [21] K. Martin, *Everyday Cryptography: Fundamental Principles and Applications*. OUP Oxford, 2012.
- [22] C. Percival, “Everything you need to know about cryptography in 1 hour.” <http://www.daemonology.net/papers/crypto1hr.pdf>, 2010.
- [23] C. Percival, “Encrypt-then-MAC.” <http://www.daemonology.net/blog/2009-06-24-encrypt-then-mac.html>, 2009.
- [24] I. Baumgart and S. Mies, “S/kademlia: A practicable approach towards secure key-based routing.” http://www.tm.uka.de/doc/SKademlia_2007.pdf, 2007.
- [25] B. Panzer-Steindel, “Data integrity,” CERN/IT. https://indico.cern.ch/event/13797/contributions/1362288/attachments/115080/163419/Data_integrity_v3.pdf, 2007.
- [26] L. N. Bairavasundaram, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, G. R. Goodson, and B. Schroeder, “An Analysis of Data Corruption in the Storage Stack,” *Trans. Storage*. <http://doi.acm.org/10.1145/1416944.1416947>; ACM, 2008.
- [27] C. Pahl, “brig: Ein Werkzeug zur sicheren und verteilten Dateisynchronisation.” <https://disorganizer.github.io/brig-thesis/brig/thesis.pdf>, 2016.
- [28] D. A. McGrew and J. Viega, “The security and performance of the Galois/Counter Mode (GCM) of operation.” <https://eprint.iacr.org/2004/193.pdf>, 2008.
- [29] X. D. C. D. Carnavalet and M. Mannan, “A Large-Scale Evaluation of High-Impact Password Strength Meters,” *ACM Trans. Inf. Syst. Secur.* <http://doi.acm.org/10.1145/2739044>; ACM, 2015.
- [30] C. Percival and S. Josefsson, “The scrypt password-based key derivation function.” <https://tools.ietf.org/html/rfc7914.html>, 2016.
- [31] M. W. Storer, K. Greenan, D. D. Long, and E. L. Miller, “Secure Data Deduplication.” <http://doi.acm.org/10.1145/1456469.1456471>; ACM, 2008.
- [32] A. Shamir, “How to Share a Secret.” <http://doi.acm.org/10.1145/359168.359176>; ACM, 1979.
- [33] S. M. Hallberg, “Individuelle Schlüsselverifikation via Socialist Millionaires’ Protocol.” <https://wiki.attraktor.org/images/7/77/Smp.pdf>, 2008.

- [34] F. Boudot, B. Schoenmakers, and J. Traore, “A fair and efficient solution to the socialist millionaires’ problem.” <http://www.sciencedirect.com/science/article/pii/S0166218X00003425/pdf?md5=9546d527297cbd8401289cf0671f071&pid=1-s2.0-S0166218X00003425-main.pdf>; Elsevier, 2001.
- [35] A. Abdul-Rahman and S. Hailes, “A Distributed Trust Model.” <http://doi.acm.org/10.1145/283699.283739>; ACM, 1997.
- [36] J. Jonczy, M. Wüthrich, and R. Haenni, “A probabilistic trust model for GnuPG.” <https://events.ccc.de/congress/2006/Fahrplan/attachments/1101-JWH06.pdf>, 2006.
- [37] Jörgen Cederlöf, “Dissecting the Leaf of Trust.” Lysator Academic Computer Society, Linköping University; <http://www.lysator.liu.se/~jc/wotsap/leafoftrust.html>.
- [38] Henk P. Penning, “analysis of the strong set in the PGP web of trust.” <http://pgp.cs.uu.nl/plot/>.
- [39] C. Wu and J. Irwin, *Introduction to Computer Networks and Cybersecurity*. CRC Press, 2016.
- [40] M. Fox, J. Giordano, L. Stotler, and A. Thomas, “Selinux and grsecurity: A case study comparing linux security kernel enhancements.” <https://pdfs.semanticscholar.org/0490/28083cf99fe1a90b2c9b03907bec57e446ee.pdf>; Citeseer, 2009.
- [41] D. J. Barrett and R. E. Silverman, *SSH: Secure Shell: Ein umfassendes Handbuch*. O'Reilly, 2002.
- [42] M. Stahnke, *Pro OpenSSH*. Apress, 2005.

Eidesstattliche Erklärung

Hiermit versichere ich, die eingereichte Abschlussarbeit selbständig verfasst und keine andere als die von mir angegebenen Quellen und Hilfsmittel benutzt zu haben. Wörtlich oder inhaltlich verwendete Quellen wurden entsprechend den anerkannten Regeln wissenschaftlichen Arbeitsens zitiert. Ich erkläre weiterhin, dass die vorliegende Arbeit noch nicht anderweitig als Abschlussarbeit eingereicht wurde.

Das Merkblatt zum Täuschungsverbot im Prüfungsverfahren der Hochschule Augsburg habe ich gelesen und zur Kenntnis genommen. Ich versichere, dass die von mir abgegebene Arbeit keinerlei Plagiate, Texte oder Bilder umfasst, die durch von mir beauftragte Dritte erstellt wurden.

Augsburg, den 15. Februar 2017

Christoph Piechula