Hochschule Düsseldorf
University of Applied Sciences

Fachbereich Medien
Faculty of Media

Analyse, Modellierung und hashcat-basierte Implementierung von Angriffen auf passwortgeschützte Systeme unter Einbeziehung des Faktors Mensch

Bachelorarbeit

im Studiengang Medieninformatik
zur Erlangung des akademischen Grades
Bachelor of Science

Fachbereich Medien
Hochschule Düsseldorf

Lukas Friedrichs
Matrikel-Nr.: 526560
Datum: 10. September 2018

Erst- und Zweitprüfer

Prof. Dr.-Ing. Holger Schmidt
Prof. Dr.-Ing. M.Sc. Markus Dahm

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Bachelorarbeit selbständig und ohne unzulässige fremde Hilfe angefertigt habe. Die verwendeten Quellen sind vollständig zitiert. Diese Arbeit wurde weder in gleicher noch ähnlicher Form einem anderen Prüfungsamt vorgelegt oder veröffentlicht. Ich erkläre mich ausdrücklich damit einverstanden, dass diese Arbeit mittels eines Dienstes zur Erkennung von Plagiaten überprüft wird.

Ort, Datum *Lukas Friedrichs*

Kontaktinformationen
Lukas Friedrichs
Rottberger Straße 201
42551 Velbert
—
lukas.friedrichs@hs-duesseldorf.de

Zusammenfassung

Analyse, Modellierung und hashcat-basierte Implementierung von Angriffen auf passwortgeschützte Systeme unter Einbeziehung des Faktors Mensch

Lukas Friedrichs

In der vorliegenden Bachelorarbeit geht es um das menschliche Verhalten bei der Passworterstellung. Hierbei wird die Möglichkeit untersucht dieses menschliche Verhalten über die Software `hashcat` nachzubilden, um so Passwörter effizienter anzugreifen. Durch die Konzeption und den Test von Passwortangriffsszenarien, deren Fokus auf dem Faktor Mensch liegt, wird versucht aufzuzeigen, dass selbst sichere Passwortverfahren, durch das individuelle Verhalten von Menschen an Sicherheit verlieren können. Zudem werden die Tests der Szenarien Schwächen der Software `hashcat` aufzeigen, die im späteren Verlauf der Arbeit als Grundlage für die Entwicklung einer selbstprogrammierten Erweiterung von `hashcat` dienen.

Abstract

Analysis, modeling and hashcat-based implementation of attacks on password-protected systems, in consideration of the human factor

Lukas Friedrichs

The present Bachelor-Thesis deals with human behavior concerning the creation of passwords. Hereby the possibility is examined to reproduce this human behavior through the software hashcat, so that passwords can be attacked more efficiently. Through the conception and testing of password-attack scenarios, when the focus is situated on the human factor, it will be examined to demonstrate that even secure password processes can lose their security through the individual behavior of humans. The tests of the scenarios will also show weaknesses of the hashcat software, which will later serve as the basis for the development of a self-programmed extension of hashcat.

Inhaltsverzeichnis

Abbildungsverzeichnis

Listings

Tabellenverzeichnis

Abkürzungsverzeichnis

BSI Bundesamt für Sicherheit in der Informationstechnik
HPI Hasso Plattner Institut
XT Extended Technology
SDK Software Development Kit
MIT Massachusetts Institute of Technology
GPU Graphics Processing Unit
KDFs Key Derivation Functions

Kapitel 1

Einleitung

Zur Authentifikation an Systemen wie Webservern oder lokalen Programmen ist die passwortbasierte Authentifizierung auch heute noch die am häufigsten verwendete Methode (Manaras, Hertlein und Pohlmann, 2016, S. 206–211). Der Begriff „Authentifikation“ wird in diesem Zusammenhang entsprechend der aktuellen Fassung der ISO/IEC 27000 wie folgt definiert:

> **authentication**
> *provision of assurance that a claimed characteristic of an entity is correct*

Somit ist die Authentifikation (im Falle der passwortbasierten Authentifizierung) die Überprüfung des übermittelten Passworts auf seine Richtigkeit.
In der folgenden Arbeit wird auf die Art und Weise, wie Menschen Passwörter bilden, eingegangen und untersucht, inwiefern es möglich ist, menschliches Verhalten beim Erzeugen und Hashen von Passwörtern zu berücksichtigen, um so die Effizienz bekannter Passwortangriffsverfahren zu erhöhen.

1.1 Kontext und Motivation

Der folgende Abschnitt stellt einen Einstieg in das Thema dar, in dem einerseits grundlegende Begrifflichkeiten beschrieben werden und andererseits auf die Motivation zum Thema der Arbeit eingegangen wird.

1.1.1 Passwortbasierte Authentifikation

Die passwortbasierte Authentifikation beschreibt den Prozess, in dem aus einem übermittelten Passwort anhand einer Hashfunktion (laut Karpfinger und Kiechle, 2009, S. 77, im Deutschen auch „Streuwertfunktion“ genannt) ein Hashwert errechnet wird, welcher dann im Anschluss mit dem Hashwert des Passworts, einer Anwendung oder eines Systems verglichen wird. Stimmen beide Hashwerte überein, erfolgt eine Authentifikation am System, liegt keine Übereinstimmung vor, gilt die Authentifizierung als fehlgeschlagen. Bei Systemen mit mehreren Nutzern, wie beispielsweise einem Webserver, wird häufig neben der Authentifikation durch ein Passwort auch ein Nutzername (ein Identifikationsmerkmal) zur Identifikation individueller Nutzer übertragen, sodass in der Datenbank des Webservers nach den passenden Vergleichswerten gesucht werden kann.

Ein Beispielszenario hierfür ist ein Nutzer, der sich während einer Registrierung an einem Webserver einen eindeutigen Nutzernamen aussucht und ein Passwort entweder auswählt oder vorgegeben bekommt, welches den Passwortanforderungen (Länge des Passworts, Zahlen, Groß- und Kleinbuchstaben, erlaubte Sonderzeichen) des Webservers entspricht. Das vom Nutzer gewählte Passwort wird durch den Webserver anhand einer Hashfunktion in einen Hashwert umgewandelt und zusammen mit dem gewählten Nutzernamen in einer Datenbank gespeichert. Nach einer erfolgreichen Registrierung kann der Nutzer sich mit seinem Nutzernamen und Passwort an dem Webserver anmelden. Hierbei findet eine unidirektionale Übertragung des Nutzernamens und des gewählten Passworts an eine verifizierende Partei (Webserver) statt. Nach der erfolgreichen Übertragung wird der Nutzername in der Datenbank des Webservers gesucht und anhand des übertragenen Passworts (mit Hilfe desselben Hashingverfahrens wie zuvor bei der Registrierung) ein Hashwert errechnet. Dieser errechnete Hashwert wird mit dem bei der Registrierung initial erzeugten Hashwert in der Datenbank des Webservers verglichen. Stimmt der Hashwert des übertragenen Passworts mit dem Hashwert in der Datenbank des Webservers überein, wird der Nutzer erfolgreich am Webserver angemeldet, bei Nichtübereinstimmung erfolgt keine Anmeldung.
Abbildung 1.1 stellt diesen Vorgang ab der Anmeldung grafisch anhand des LaTeX-Pakets „tikz-uml“[1] (UML Version 2.4.1[2]) dar:

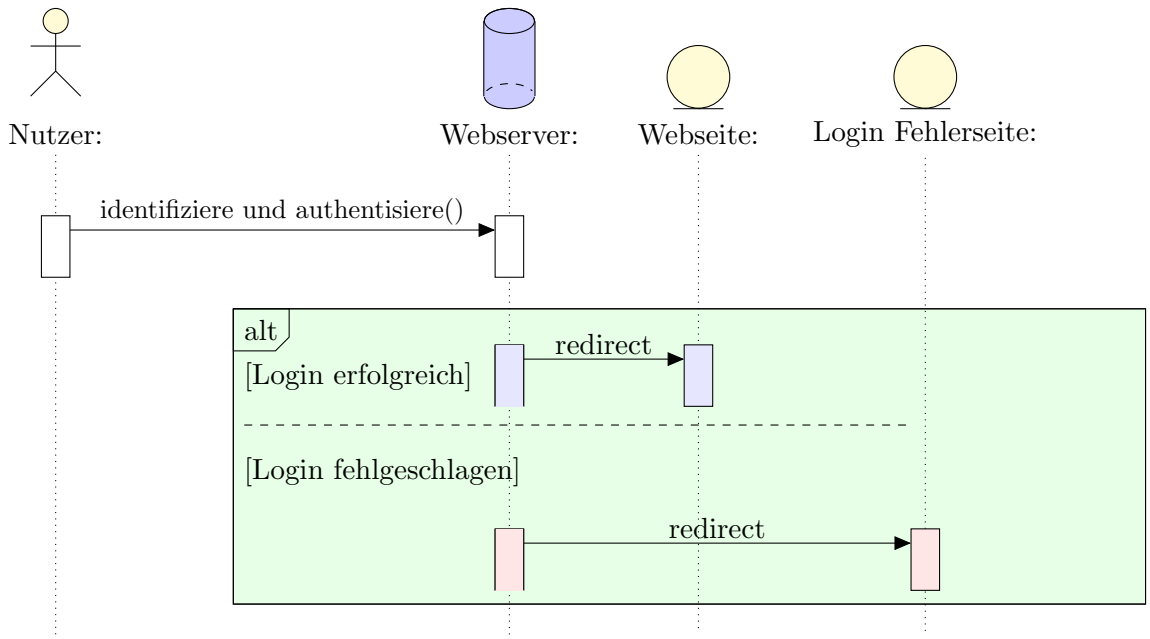

Abbildung 1.1: UML-Sequenzdiagramm zur passwortbasierten Authentifizierung

1.1.2 Menschen und Passwörter

Menschen erzeugen Passwörter häufig anhand von Wörtern, Zahlen oder Zeichen, mit denen sie etwas verbinden, um sich das so erzeugte Passwort besser merken zu

[1] http://perso.ensta-paristech.fr/~kielbasi/tikzuml/index.php?lang=en, abgerufen am 28.02.2018
[2] http://www.omg.org/spec/UML/, abgerufen am 28.02.2018

können. Beispiele hierfür sind Passwörter nach dem folgenden Schema:

- der Name und Geburtstag des Kindes
- der Hochzeitstag und Name der Ehefrau
- der Name eines Haustieres

So erzeugte Passwörter können zwar leicht von einem Nutzer im Gedächtnis behalten werden, jedoch eben so leicht durch einen Angreifer erraten werden. Somit wird deutlich, dass die größte Schwäche der Authentifizierung anhand eines Passworts der Faktor Mensch ist. Dies schrieb auch Mitnick in seiner Veröffentlichung *Art of Deception*. Folglich sind sichere Passwörter oft benutzerunfreundlich, während leicht zu merkende Passwörter häufig unsicher sind. Hieraus ergibt sich zum einen die Frage, auf welche Weise Menschen Passwörter alternativ definieren können und zum anderen inwiefern dabei die Sicherheit der Passwörter gewährleistet und die Benutzerfreundlichkeit erhalten bleibt. Dementgegen stellt sich die Frage, ob ein Passwortangriff der anhand von Angriffsszenarien die explizit menschliches Verhalten mit einbeziehen effizienter ist als ein herkömmlicher Passwortangriff, wie beispielsweise die Brute-Force-Methode (Paar und Pelzl, 2016, S. 7).

1.2 Wissenschaftlicher Beitrag

Die priorisierten Ziele dieser Arbeit teilen sich in zwei Bereiche auf. Zum einen werden theoretische Angriffsszenarien auf der Basis von passwortbasierten Authentifikationsverfahren entwickelt, bei denen explizit menschliches Verhalten mit einbezogen wird. Der Fokus dieser Szenarien liegt dabei sowohl auf dem Einfluss beziehungsweise der Auswirkung den der menschliche Faktor auf das jeweilige Szenario hat, als auch auf der Durchführbarkeit der Szenarien in einem an die Konzeption anschließenden Test. Getestet werden die Szenarien hierbei anhand der Software `hashcat`[3]. Dieser Teil der Arbeit wird Aufschluss darüber geben, inwiefern ein über `hashcat` durchgeführter Angriff durch die Einbeziehung menschlichen Verhaltens verbessert werden könnte. Außerdem wird aufgezeigt, welche Möglichkeiten und Funktionen `hashcat` einem Nutzer für die Nachahmung menschlichen Verhaltens bietet und welche nicht abgebildet werden können. Im zweiten Teil der Arbeit wird die Entwicklung einer Erweiterung von `hashcat` beschrieben, deren Funktionen sich direkt an den zuvor beschriebenen Testergebnissen orientieren.
Diese Arbeit hat nicht den Anspruch vollkommen neue Angriffsmöglichkeiten auf Passwörter zu entwerfen. Vielmehr werden im Verlauf der Arbeit bekannte Verfahren zum Ermitteln von Passwörtern so angepasst, dass sie (soweit möglich) menschliches Verhalten nachbilden und somit an Effizienz gewinnen.

1.3 Weiteres Vorgehen

Im weiteren Verlauf der Arbeit werden im nächsten Kapitel der aktuelle Forschungsstand mit den Grundlagen zu Passwörtern und ihrer Erstellung, den maßgeblichen

[3] https://hashcat.net/hashcat/, abgerufen am 28.02.2018

Faktoren der Passwortsicherheit, Hashingverfahren, Passwortangriffsverfahren sowie einem Überblick über die Software `hashcat` beschrieben. Anschließend wird im folgenden Kapitel „Angriffsszenarien und Optimierungen“ auf die Konzeption der Angriffsszenarien, mit dem Faktor Mensch im Fokus, eingegangen. Darauf folgt ein praktischer Test der in `hashcat` abgebildeten Angriffsszenarien, wobei nach jedem der durchgeführten Tests die Ergebnisse analysiert und beschrieben werden. Anhand der Testergebnisse werden daraufhin einerseits Empfehlungen zur Nutzung von `hashcat` dargelegt und andererseits wird auf mögliche Probleme während der Tests mit `hashcat` eingegangen und erläutert, wie man diese lösen könnte. Das darauf folgende Kapitel „Implementierung“ beschreibt die Entwicklung einer Erweiterung von `hashcat` zur Lösung der während der Tests aufgetretenen Probleme. Das letzte Kapitel „Schlussfolgerung“ enthält sowohl ein Fazit zur Arbeit als auch einen weiteren Ausblick. Das Kapitel und die Arbeit enden mit einem persönlichen Schlusswort.

Kapitel 2

Aktueller Forschungsstand und Grundlagen

Nach dem voranstehenden Einstieg ins Thema wird nun auf den aktuellen Forschungsstand eingegangen, um grundlegende Begriffe und die derzeitigen Forschungsgegebenheiten zu klären und in einen gesamtheitlichen Kontext zu setzen. Hierbei wird auf Passwörter im Allgemeinen, sowie auf Faktoren, die die Sicherheit eines Passworts beeinflussen, eingegangen. Bei diesen Faktoren wird besonderer Bezug auf den Faktor Mensch genommen und wie sich dieser auf ein Passwort auswirken kann. Im Weiteren wird eine Auswahl an Verfahren zum Erzeugen von Hashwerten thematisiert. Hierbei wird unterschieden zwischen kryptografischen Hashingverfahren und Hashingverfahren, die sogenannte Schlüsselableitungsfunktionen (engl. „key derivation function“) einsetzen und sich besonders für das Hashen von Passwörtern eignen. Im Anschluss daran wird eine Auswahl bekannter Passwortangriffsverfahren beschrieben, wobei auf Techniken und Verfahren zum Ermitteln von Passwörtern eingegangen wird. Der letzte Abschnitt des Kapitels gibt eine Übersicht über die in Kapitel 3.2 eingesetzte Software `hashcat`.

2.1 Was ist ein Passwort?

Der Begriff „Passwort“ leitet sich aus dem Englischen „pass“ zu deutsch „Ausweis“, „Passierschein“ und „word“ zu deutsch „Wort“ ab. Es ist ein Synonym für die Begriffe „Losungswort“ beziehungsweise „Parole“ und setzt sich im Allgemeinen aus einer Zeichenfolge, welche aus Buchstaben, Ziffern und/oder Sonderzeichen besteht, zusammen[1] . Ein Passwort ist ein Geheimnis, mit dem sich eine Partei A (ein Nutzer) an einer verifizierenden Partei B (beispielsweise einem Server) identifiziert (Spitz, Pramateftakis und Swoboda, 2011, S. 150). Passwörter treten in verschiedenen Formen und Arten auf. Sie werden dabei unterteilt in „Einmalpasswörter“ und „Dauerpasswörter“. Einmalpasswörter sind nach einmaligem Gebrauch ungültig. Dies verleiht ihnen ein besonderes Sicherheitsmerkmal, da ein Angreifer aus abgehörten Passwörtern keinen Nutzen ziehen kann, da sie bei der nächsten Benutzung nicht mehr gültig sind. Beispiele für Einmalpasswörter sind z.B. TAN-Codes die häufig beim Online-Banking genutzt werden. Weiter verbreitet sind Dauerpasswörter, welche zum Beispiel von einem Nutzer zur Entsperrung seines Smartphones oder PCs verwendet werden. Hierbei

[1] https://de.wikipedia.org/wiki/Passwort, abgerufen am 19.05.2018

kann unterschieden werden zwischen der passwortbasierten Authentifikation mit und ohne Identifizierungsmerkmal. Wie zuvor in Kapitel 1.1.1 erwähnt, dient ein Identifizierungsmerkmal zur Identifikation eines bestimmten Nutzers an einem Mehrbenutzersystem. Dies kann beispielsweise (wie im Fall des Computerlogins) anhand eines eindeutigen Nutzernamens oder auch einer E-Mail-Adresse geschehen. Der PIN-Code eines Smartphones hingegen benötigt kein zusätzliches Identifizierungsmerkmal, da ein Smartphone nur einem Benutzer zugeordnet ist.

2.2 Verfahren zum Bilden von Passwörtern

Es gibt diverse Verfahren zum Entwickeln von Passwörtern. Im Folgenden werden fünf Verfahren beschrieben.

2.2.1 Akronym-Methode

Die Akronym-Methode nutzt als Basis eine sogenannte „Passphrase“ (siehe Kapitel 3.4.3) und ähnelt sehr der nach ihrem Entwickler benannten „Bruce Schneier's“-Methode. Laut Moschgath, Roedig und Schumacher, 2012, S. 200 muss sich der Nutzer hierbei einen oder mehrere Sätze überlegen, die leicht wieder in Erinnerung gerufen werden können. Das eigentliche Passwort wird jeweils aus den Anfangsbuchstaben der einzelnen Wörter des Satzes (oder der Sätze) gebildet. Sicherer wird das daraus resultierende Passwort, wenn zusätzlich beispielsweise alle Vokale gegen Ziffern und/oder Sonderzeichen ersetzt werden oder aber einem Wort ein Sonderzeichen zugeordnet wird. Beispielsweise könnte man statt dem Wort „Passwort“ ein Fragezeichen (?) schreiben. Am Schluss können nach Belieben weitere Zahlen oder Sonderzeichen angehangen werden. Für diese Methode eignen sich besonders Gedichte oder auch bekannte Sprichwörter, da sie dem Nutzer gut und leicht in Erinnerung bleiben. Darüber hinaus bestehen sie in der Regel aus einer hinreichenden Menge an Wörtern, woraus ein Passwort mit ausreichender Länge resultiert. Wichtig hierbei ist die Verwendung von Variationen aus Zahlen und Sonderzeichen, am Besten durch Ersetzen von Satzteilen am Ende, um sicherzustellen, dass falls ein Angreifer den Merksatz eines Passworts herausfinden sollte, dieser trotzdem nicht in der Lage ist, das daraus resultierende Passwort abzuleiten. Beispiele für die Akronym-Methode sind:

- Dies ist einer von drei Passwortsätzen = Di1v3?S-1234!
- Nur ein Narr würde mit Kanonenkugeln auf Spatzen schießen! = N1N@33wmK@Ss!
- Dieses Akronym ist schwach und leicht zu erraten = DAisulze

2.2.2 Doppelwort-Methode

Bei der Doppelwort-Methode werden zwei beliebige Wörter zu einem Wort kombiniert. Die Idee ist, dass ein Nutzer zwei ihm bekannte Wörter - durch weglassen und neu anordnen von Buchstaben - zu einem neuen Wort vereint. Sicherer wird diese Methode durch Hinzufügen von Zahlen und Sonderzeichen an passenden Stellen. Beispielsweise können Vokale durch Zahlen ersetzt werden und die Buchstaben der

jeweiligen Ursprungsworte können durch ein Sonderzeichen getrennt werden. Denkbar wäre auch an passenden Stellen Buchstaben durch passende Sonderzeichen zu ersetzen. Beispiele für die Doppelwort-Methode sind:

- Leberwurst und schweizer Rösti = berwzti oder berw%zti
- Strand und Meer = trndeer oder 5trnd&3er!
- Skifahren und Berg = kifahrerg oder 5k1f@%3rg?

Der Nachteil dieser Methode liegt unter anderem darin, dass die resultierenden Passwörter meist entweder sehr kurz sind oder der Nutzer sich zwar die Schlüsselwörter merken kann, aber Probleme hat sich alle Modifikationen einzuprägen. Da bei dieser Methode die ausgewählten Wörter gekürzt beziehungsweise entstellt werden, kann dies schnell zu Problemen bei der Einprägsamkeit des Passworts führen.

2.2.3 Collage-Methode

Bei der Collage-Methode wählt der Nutzer ein Wort aus einer natürlichen Sprache und übersetzt es in mindestens eine, besser in zwei oder mehr Sprachen. Anschließend werden jedem der übersetzten Wörter Buchstaben entnommen (am Anfang, in der Mitte oder am Ende) und mit Zahlen oder Sonderzeichen verbunden (Moschgath et al., 2012, S. 200). Beispiele für die Collage-Methode sind:

- Haus und house mit der Hausnummer 17 = hou:17Hau!
- Pferd : Horse $ Cheval = P:rs$val
- Huhn & Chicken und davon 15 = un&ck3n15

Ähnlich der Doppelwort-Methode besteht hier die Gefahr wenig einprägsame Passwörter zu erzeugen, da die Collage-Methode beim Beschneiden und Modifizieren der übersetzten Wörter keinem vorgegebenen Schema folgt. Dies sorgt zwar einerseits für eine hohe Unvorhersehbarkeit und somit für eine erhöhte Sicherheit der resultierenden Passwörter, andererseits sorgt es aber auch dafür, dass das Passwort vom Nutzer leichter vergessen werden kann.

2.2.4 Zufallsmethode

Bei der Zufallsmethode werden nach dem „Zufallsprinzip“ acht oder zwölf Zeichen (je nachdem wie viele Zeichen das Passwort haben soll) miteinander kombiniert. Da es in diesem Fall für einen Angreifer kein nachvollziehbares System gibt, ist es bezogen auf die Sicherheit ein nicht mehr zu überbietendes Passwort. Allerdings bringt diese Methode auch große Nachteile mit sich. Da solche Passwörter beim Auswendiglernen und auch bei der Blindeingabe große Probleme bereiten, bieten sie nur eine scheinbare Sicherheit. Die größte Gefahr besteht wohl darin, dass ein Nutzer sich die zufällige Kombination nicht merken kann und sie irgendwo notiert, sodass ein Angreifer das notierte Passwort finden kann. Darüber hinaus besteht eine erhöhte Gefahr, wenn die Eingabe des Passworts zu viel Zeit in Anspruch nimmt und somit die Möglichkeit besteht, dass anwesende Personen die eingegebene Zeichenfolge unauffällig mitlesen könnten, laut Raza, Iqbal, Sharif und Haider, 2012, S. 2 wird in diesem Fall auch von „shoulder surfing“ gesprochen.

2.2.5 Diceware-Methode

Die Diceware-Methode wurde 1995 von Arnold G. Reinhold erfunden[2]. Die Idee hinter der Diceware-Methode ist die Erzeugung einer sehr sicheren Passphrase (siehe Kapitel 2.3.2), die trotzdem leicht in Erinnerung behalten werden kann. Wie zuvor am Beispiel der Zufallsmethode erklärt, ist eine Methode zur Passworterzeugung, die per Zufall Zeichen miteinander kombiniert, um ein Passwort zu erzeugen sehr sicher, aber auch sehr benutzerunfreundlich. So erzeugte Passwörter können durch einen Nutzer nur schwer bis gar nicht in Erinnerung behalten werden. Bei der Diceware-Methode hingegen ist jedes Wort der Passphrase (siehe Kapitel 2.3.2) aussprechbar, es handelt sich um einfache Begriffe oder Abkürzungen die somit leicht durch Menschen in Erinnerung behalten werden können. Das macht Diceware zu einer sehr benutzerfreundlichen Passwortmethode. Der Ablauf ist hierbei folgendermaßen: Ein Nutzer rollt einen sechsseitigen Würfel fünfmal, wobei jede gewürfelte Zahl notiert werden muss. Nach dem fünften Wurf ist eine fünfstellige Zahl entstanden, die in einer Diceware-Wortliste (welche in diversen Sprachen auf der Diceware-Webseite zur Verfügung steht) nachgeschlagen werden kann. In diesen Wortlisten sind jeder fünfstelligen Zahl immer ein Wort zugeordnet. Aufgrund der festgelegten Anzahl an Wörtern innerhalb einer Diceware-Wortliste, hat jedes „erwürfelte“ Wort eine feste Entropie (siehe Kapitel 2.3.3) von 12,9 Bits. Seit Anfang 2014 empfiehlt Arnold G. Reinhold eine Diceware-Passphrase bestehend aus mindestens sechs Wörtern, was einer Entropie von 77,4 Bit entspricht. Da eine Diceware-Wortliste 7.776 Wörter enthält, würde das bei einer Passphrase bestehend aus sechs Wörtern, 7.776^6(221.073.919.720.733.357.899.776) mögliche Passwörter ergeben. Ausgehend von einem einzelnen Computersystem, welches eine Milliarde (1.000.000.000.000) Passwörter pro Sekunde erraten kann (Edward Snowden ende 2013[3]), würde sich die Zeit, die das System bräuchte um alle Kombinationen durchzuprobieren, folgendermaßen errechnen:

$$\frac{2^{77,4}}{1.000.000.000.000} = 199.398.397.814 \text{ Sekunden}$$

Diese 199.398.397.814 Sekunden müssen nun in Jahre umgerechnet werden (1 Jahr hat ungefähr 31.536.000 Sekunden):

$$\frac{199.398.397.814}{31.536.000} \approx 6.323 \text{ Jahre}$$

Somit würde ein System, welches eine Milliarde Passwörter pro Sekunde ermitteln kann, ungefähr 6.323 Jahre benötigen um alle Kombinationen einer Diceware-Passphrase mit sechs Wörtern (77,4 Bit Entropie) durchzuprobieren.

2.3 Faktoren der Passwortsicherheit

Die Sicherheit eines Passworts wird anhand verschiedener Faktoren bestimmt bzw. ist von diesen abhängig. Im folgenden Abschnitt wird auf einige dieser Faktoren eingegangen und beschrieben, wie die Beachtung dieser Faktoren einen Nutzer dabei

[2]http://world.std.com/~reinhold/diceware.html, abgerufen am 30.03.2018, f.
[3]https://www.wired.com/2014/10/snowdens-first-emails-to-poitras/, abgerufen am 30.03.2018

unterstützt, gute und somit sichere Passwörter zu erstellen. Ebenso werden Faktoren betrachtet, die die Sicherheit eines im Grunde guten Passworts gefährden oder mindern können.

2.3.1 Was ist ein gutes Passwort?

Einer der wichtigsten Punkte der Passwortsicherheit ist die Qualität beziehungsweise Stärke eines Passworts. Ein gutes und somit starkes Passwort zeichnet sich durch eine möglichst geringe Wahrscheinlichkeit aus, anhand eines Passwortangriffs ermittelt werden zu können (Rohr, 2015, S. 203). Je mehr Zeit ein Angreifer benötigt um ein Passwort zu ermitteln, desto stärker ist ein Passwort. Wichtige Faktoren sind hierbei sowohl der Aufbau - im speziellen die Länge des Passworts, sowie die Vorhersehbarkeit oder auch die Entropie (siehe 2.3.3).

2.3.2 Aufbau und Länge eines Passworts

Aus dem Aufbau und der Länge eines Passworts, wie auch schon in Kapitel 1 und 2.2 beschrieben, ergeben sich diverse Möglichkeiten ein Passwort zu erzeugen. Beim Aufbau eines Passworts geht es zum einen um die Art und Weise wie ein Passwort erzeugt wurde (siehe Kapitel 2.2.1 bis 2.2.5) und zum anderen aus welcher Menge an Zeichen ein Passwort gebildet wurde. In der Kategorie der textbasierten Passwörter können Passwörter anhand von alphanumerischen Zeichen gebildet werden. Im engeren Sinne gehören zu den alphanumerischen Zeichen nur Zahlen und Buchstaben, jedoch werden dem Begriff in der Telekommunikation und Computertechnik auch Sonderzeichen zugeordnet[4]. Neben der Menge aus Zahlen, Buchstaben und Sonderzeichen spielt die Länge eines Passworts eine wichtige Rolle. Selbst wenn bei der Erstellung eines Passworts der Passwortraum vollständig ausgenutzt wurde und ein Passwort sowohl aus Groß- als auch aus Kleinbuchstaben, Zahlen und Sonderzeichen erstellt wurde, so beträgt die minimale Länge des Passworts nur vier Zeichen (ein Zeichen pro Kategorie). Tests mit sogenannten „Password Strength Metern“ wie dem des Unternehmens Kaspersky[5] (Abbildung 2.1) zeigen, dass ein solches vierstelliges Passwort, obwohl es die maximale Komplexität an Zeichen aufweist, innerhalb weniger Sekunden bis Minuten ermittelt werden kann.

[4] https://de.wikipedia.org/wiki/Alphanumerische_Zeichen, abgerufen am 05.05.2018
[5] https://password.kaspersky.com/, abgerufen am 03.05.2018

Abbildung 2.1: Test eines vierstelligen Passworts, bei dem Zeichen aus allen vier möglichen Zeichengruppen verwendet werden, anhand des „Passwort Strength Meters“ des Unternehmens Kaspersky. (Quelle: https://password.kaspersky.com)

Dies macht deutlich, wie wichtig die Länge eines Passworts ist. Eine besondere Passwortart ist die sogenannte „Passphrase“. Eine Passphrase bezeichnet ein Passwort mit einer besonders großen Länge. Hierbei handelt es sich nicht um ein einzelnes Wort, sondern um die Aneinanderreihung mehrerer Wörter. Passphrases haben den Vorteil, dass sie häufig, trotz ihrer größeren Länge, wesentlich einprägsamer sind als normale Passwörter. Das liegt daran, dass beispielsweise ein achtstelliges Passwort, welches den kompletten Passwortraum voll ausnutzt und aus zufällig kombinierten Zeichen besteht, nur schwierig in Erinnerung behalten werden kann. Dementgegen ist eine Passphrase häufig nichts anderes als ein Alltagssatz, wie z.B. „2018 habe ich mein Ziel erreicht!“. Dieser einfache Satz hat (Leerzeichen eingeschlossen) eine Länge von 33 Zeichen und aufgrund der Jahresangabe und des Ausrufezeichens am Ende, wird der komplette Passwortzeichenraum ausgenutzt. Wenn dies nun verglichen wird mit einem achtstelligen Passwort aus zufällig kombinierten Zeichen, wie z.B. „zUmAcr@4“, wird schnell ein großer Vorteil von Passphrases offensichtlich. Trotz ihrer Länge sind Passphrases wesentlich einprägsamer als herkömmliche Passwörter und bleiben einem Nutzer daher besser im Gedächtnis. Abbildung 2.2 zeigt hierzu einen oft diskutierten xkcd-Comic (Shay et al., 2012), der versucht die Vorteile einer Passphrase gegenüber einem einfachen Passwort mit wenigen Worten darzustellen.

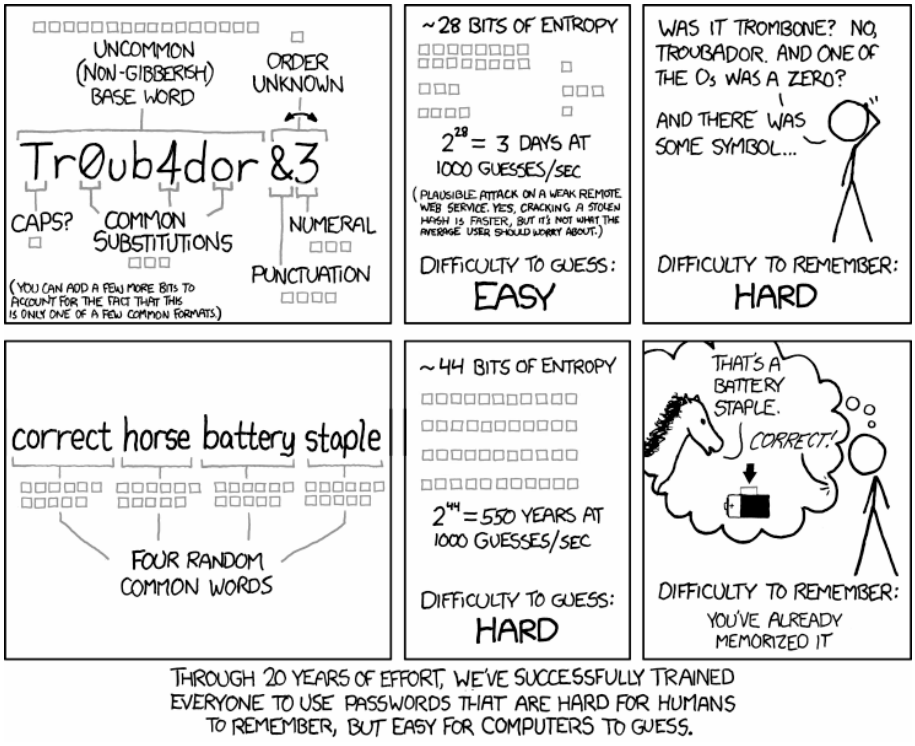

Abbildung 2.2: xkcd-Comic, Gegenüberstellung von Passphrases und Passwörtern (Quelle: https://xkcd.com/936/)

Passphrases bieten allerdings auch Nachteile, einerseits muss ein Nutzer bei der Eingabe einer Passphrase mehr Zeit aufwenden als bei einem einfachen Passwort und andererseits besteht aufgrund der Menge an Zeichen eine größere Wahrscheinlichkeit sich bei der Eingabe zu vertippen. Beides kann schnell zu einer Unzufriedenheit auf Nutzerseite führen (Nielsen, Vedel und Jensen, 2014).

2.3.3 Entropie

Die Sicherheit der passwortbasierten Authentifikation ist im Besonderen abhängig von der Stärke des verwendeten Passworts. Die Stärke eines Passworts wird in Bit gemessen und als Entropie bezeichnet. Bei der Entropie geht es um den möglichen Informationsgehalt einer Nachricht, beziehungsweise im Fall der Passwortauthentifikation um den Informationsgehalt eines Passworts. Durch sie wird die Auftrittswahrscheinlichkeit jedes Zeichens des definierten Passworts beschrieben (Fox, 2008). Wenn nun die Entropie eines Passworts bestimmt werden soll, wird der Informationsgehalt aller Zeichen summiert. Um nicht die Entropie jedes einzelnen Zeichens berechnen zu müssen, kann anhand der Länge des Passworts, sowie der Menge an Zeichen aus denen es gebildet wurde, der Entropiewert des gesamten Passworts errechnet werden. Zu dieser Menge an Zeichen können die folgenden Zeichengruppen gehören:

- Kleinbuchstaben des Alphabets (26 Zeichen)
- Großbuchstaben des Alphabets (26 Zeichen)
- Zahlen von 0 bis 9 (10 Zeichen)

- Sonderzeichen wie zum Beispiel $, &, %, #, {, }, [,], _, @, §, £, <, >, \, ?, !, sowie Leerzeichen (32 Zeichen)

Laut **MathematicalTheoryofCommunication1963** berechnet sich die Entropie eines Passworts wie folgt:

$$H = \text{Entropie}$$
$$N = \text{Menge der möglichen (/zulässigen) Werte}$$
$$L = \text{Länge des Passworts}$$

$$\text{Formel: } H = L * \log(N)/\log(2)$$

Je größer der resultierende Entropiewert ist, umso besser wird der zur Verfügung stehende Passwortraum genutzt und desto kleiner ist die Vorhersehbarkeit des generierten Passworts (Rohr, 2015, S. 204). Wichtig hierbei ist, ob die verwendeten Zeichen zufällig gewählt wurden oder nicht. Denn auch wenn ein Passwort einen hohen Entropiewert aufgrund seiner Länge und der Menge der verwendeten Zeichen aufweist, macht es einen großen Unterschied, ob es sich um eine zufällige Zeichenkombination oder beispielsweise um ein sinngebendes Wort handelt. Ein Angreifer der das Passwort eines Nutzers ermitteln möchte, kann anhand diverser Angriffstechniken ein Passwort erraten. Dies kann deutlich erschwert werden, wenn es sich bei dem Passwort um eine Menge von zufällig gewählten Zeichen handelt. Der Nachteil ist allerdings, dass ein so erzeugtes Passwort nur schwer durch einen menschlichen Nutzer in Erinnerung gerufen werden kann und somit häufig vergessen wird. Dies kann schnell zu einer Nichtakzeptanz auf Nutzerseite führen.

2.3.4 Mehrfaktor-Authentifikation

Häufig werden zur Realisierung von Authentifikationsverfahren Kombinationen aus mehr als einer Authentifikationstechnik verwendet. Dies geschieht um einerseits die Sicherheit zu erhöhen und andererseits die Vorteile der unterschiedlichen Techniken gezielt zu nutzen und zu kombinieren. Wenn beispielsweise zwei Techniken kombiniert werden, spricht man von einer Zwei-Faktor-Authentifikation. Diese basiert in der Regel darauf, dass zum einen ein Identifikationsgegenstand und zum anderen eine geheime Information, eine Art Passwort, vorhanden sein muss (Eckert, 2008, S. 431). Ein Beispiel ist der Bezahlvorgang in einem Kaufhaus mit einer EC- oder Kreditkarte. Die Identität eines Kunden wird hierbei durch den Besitz der EC- oder Kreditkarte und durch die Kenntnis seiner PIN-Nummer (spezifisches Wissen) bestätigt.

2.3.5 Faktor Mensch

Der Faktor Mensch stellt in der IT-Sicherheit, und vor allem in der Passwortsicherheit, die menschliche Fähigkeit dar, den Grad der Sicherheit eines Passworts zu beeinflussen. Dies zeigt sich besonders bei der Erstellung eines Passworts und der Art und Weise wie mit einem erstellten Passwort (im Alltag) umgegangen wird. Viele Sicherheitskonzepte sehen den Menschen als Schwachstelle eines Systems an. Unabhängig von der theoretischen Sicherheit eines Systems - oder im Fall von Passwörtern - einem Passwortverfahren, besteht immer die Gefahr, dass ein Mensch durch unbewusstes

oder ungeschultes Verhalten die Sicherheit dieses Systems mindert. Wenn die Wahl eines textbasierten Passworts einem Menschen überlassen wird, wird diese Wahl immer durch die Erfahrungen und das Wissen dieses Menschens beeinflusst. Hierzu zählen zum Beispiel die Herkunft, aber auch die Sprache oder die persönlichen Präferenzen wie Interessen oder Hobbys. Das Resultat sind häufig schwache und meist kurze Passwörter, was durch Bonneau, 2012 nachgewiesen werden konnte. Um dem entgegenzuwirken, werden fortlaufend Maßnahmen entwickelt, um die durch den Menschen eingebüßte Sicherheit eines Passworts auszugleichen und immer weiter zu minimieren. Hierzu zählen zum einen zusätzliche Authentifikationsmethoden, die mit der Abfrage eines Passworts kombiniert werden (siehe hierzu Kapitel 2.3.4), zum anderen die bessere Anleitung eines Nutzers bei der Passworterstellung. Schon lange werden Nutzer bei der Erstellung eines Passworts darauf hingewiesen, wie ein Passwort aufgebaut sein sollte, um ein höheres Maß der Sicherheit zu erreichen. In der Vergangenheit waren viele solcher „Password Strength Meter“ (beispielsweise bei der Registrierung auf einer Webseite) darauf ausgerichtet, eine bestimmte Passwortrichtlinie zu erzwingen, da sie nur Passwörter zuließen, die der geforderten Richtlinie entsprachen. Dies hat dem Nutzer jedoch in vielen Fällen weniger die Relevanz eines starken Passworts vermittelt, als viel mehr die Art und Weise wie ein Passwortassistent „zufriedenzustellen“ ist. Neuere Implementierungen dieser „Password Strength Meter“ verfolgen hier einen besseren Ansatz, indem sie dem Nutzer eine ungefähre Einschätzung des nötigen Zeitaufwandes, um das vom Nutzer gewählte Passwort zu ermitteln, geben. Gleichzeitig werden dem Nutzer aber auch nützliche Hinweise gegeben um die angegebene Zeit zu vergrößern. Ein gutes Beispiel hierfür ist das durch Wheeler, 2016 und Dropbox Inc.[6] entwickelte Tool `zxcvbn`[7]. Der Faktor Mensch ist die entscheidende Variable zwischen der Sicherheit und der Usability (dt. Benutzerfreundlichkeit) eines Passworts. Das Problem ist, dass dieser Faktor nur schwer in Gänze kalkuliert werden kann, da er nicht vorhersehbar ist. Es besteht lediglich die Möglichkeit, an einem bestehenden System anhand von Tests (mit verschiedenen Nutzergruppen) einen gewissen Anteil dieses Faktors zu evaluieren. Eine weitere Ausprägung des Faktors Mensch wurde durch Jaeger, Pelchen, Graupner, Cheng und Meinel, 2016, S. 15 nachgewiesen. Hierbei handelt es sich um den Nachweis, dass Menschen oft dazu tendieren Passwörter wiederzuverwenden. Hierzu wurden die 2015 veröffentlichten Nutzerdaten der Online-Dating-Plattform Ashley Madison[8] untersucht. Da die mit den Nutzerdaten veröffentlichten Passwort-Hashwerte anhand des bcrypt-Verfahrens (siehe Kapitel 2.4.2) mit einem Kostenfaktor von 12 generiert wurden, wäre ein direkter Angriff äußerst rechenintensiv gewesen. Aus diesem Grund wurde der Untersuchungsansatz, inwiefern E-Mail-Adressen des „Ashley Madison Leaks“ auch Teil anderer Passwort-Leaks waren, gewählt, da sie vermuteten, dass viele der betroffenen Nutzer, das Passwort welches sie auf der Webseite von Ashley Madison gewählt haben, auch auf anderen Webportalen verwendet haben. Sie haben daraufhin alle Klartextpasswörter von Vorfällen, bei denen E-Mail-Adressen des Ashley Madison Vorfalls involviert waren, gesammelt und versucht mit ihnen die gesuchten Hashwerte zu ermitteln. Daraufhin war es ihnen möglich 2,9 Millionen Passwörter innerhalb weniger Stunden zu ermitteln und den entsprechenden E-Mail-Adressen zuzuordnen.

[6] https://www.dropbox.com/de, abgerufen am 13.05.2018
[7] https://github.com/dropbox/zxcvbn, abgerufen am 11.05.2018
[8] https://en.wikipedia.org/wiki/Ashley_Madison_data_breach, abgerufen am 11.05.2018

Dies zeigt sehr deutlich die große Gefahr der Wiederverwendung von Passwörtern. Trotz der sehr sicheren Aufbewahrung der Nutzerpasswörter seitens Ashley Madison (anhand des bcrypt-Verfahrens), konnte das eigentlich gute Sicherheitskonzept durch die nicht kontrollierbare Handlungsweise von Menschen massiv geschwächt und schließlich durchbrochen werden.

2.4 Hashingverfahren und ihre Unterschiede

In diesem Abschnitt geht es um zwei verschiedene Kategorien von Hashfunktionen (Streuwertfunktionen). Bei jeder Kategorie wird beschrieben, worum es bei dieser Art von Hashverfahren geht und was es ausmacht. Im weiteren Verlauf werden jeweils zwei Hashfunktionen jeder Kategorie vorgestellt und beschrieben. Darüber hinaus wird auf Sicherheitsmerkmale von Hashfunktionen wie „Hash-Kollisionen“, „Salt“ und „Pepper“ eingegangen.
Was ist nun also ein Hashwert?
Schwenk, 2014, S. 14 beschreibt einen Hashwert als eine Abbildung, die aus einem beliebig langen Datensatz (Eingabewert) unter Zuhilfenahme einer Hashfunktion errechnet wird. Der resultierende Hashwert ist eine kryptografische Prüfsumme mit einer festen (definierten) Länge.

2.4.1 Kryptographische Hashfunktionen

Dieser Abschnitt beschreibt die Kategorie der kryptografischen Hashfunktionen. Laut Spitz et al., 2011, S. 31 unterscheiden sich kryptografische Hashfunktionen von anderen Hashfunktionen durch die Eigenschaft der Kollisionsresistenz (siehe hierzu Kapitel 2.4.3). Sie werden weiterhin als „Einwegfunktionen“ bezeichnet, da es nicht möglich ist (und nicht möglich sein darf) eine solche Hashfunktion umzukehren, um so über den Hashwert Rückschlüsse auf die ursprüngliche Nachricht (z.B ein Passwort) ziehen zu können. Mögliche Anwendungsgebiete für kryptografische Hashfunktionen sind:

- Generierung von Blockchiffren[9]
- Integritätsprüfungen von Dateien oder Texten
- digitale Signaturen oder Passworte sicher zu speichern

Im weiteren Verlauf werden die zwei kryptografischen Funktionen „MD5“ und „SHA-512“ beschrieben.

MD5-Verfahren

Das MD5-Verfahren (Message-Digest-Algorithm 5) wurde für viele Hash-Algorithmen als Vorbild genommen. Die Funktion kann aus einer beliebigen Nachricht (zum Beispiel ein Text, ein Passwort oder eine Datei) einen 128-Bit-Hashwert generieren. Aufgrund der Geschwindigkeit der Funktion, eignet sie sich beispielsweise sehr gut um einen Download auf seine Korrektheit zu überprüfen[10]. Laut Spitz et al., 2011, S. 103

[9] https://de.wikipedia.org/wiki/Blockverschl%C3%BCsselung, abgerufen am 13.05.2018
[10] https://de.wikipedia.org/wiki/Message-Digest_Algorithm_5, abgerufen am 13.05.2018

wird, zum Generieren eines Hashwerts, die übergebene Nachricht jeweils in 512-Bit-Blöcke unterteilt. Der letzte Block wird anhand von „Padding“-Daten[11] und einer Längenangabe aufgefüllt. Die Längenangabe markiert hierbei die Grenze zwischen der zu hashenden Nachricht und den Padding-Daten. Jeder der 512-Bit-Blöcke wird in 16 Wörter (0, ..., 15) mit jeweils 32 Bit heruntergebrochen. Jeder der Blöcke wird in 64 Iterationen verarbeitet, bei denen je eines der 16 Wörter miteinbezogen wird. Innerhalb von vier unterschiedlichen Runden wird jedes der 32-Bit-Wörter verwendet.

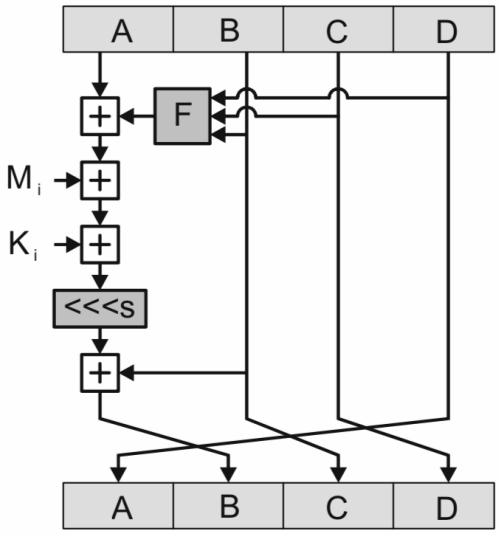

Abbildung 2.3: Aufbau einer Iteration von MD5 (Spitz, Pramateftakis und Swoboda, 2011, S. 103)

MD5 hat hierbei einen Zustand von vier Wörtern (A, B, C, D) mit jeweils 32 Bit (siehe Abbildung 2.3). Die 32-Bit-Wörter (A, B, C, D) werden vor dem Anfang des ersten Blocks jeweils mit einer definierten Konstanten vorbelegt. Am Ende der Verarbeitung gibt dieser Zustand den Hashwert von $4 * 32 = 128$ Bit zurück.

SHA-512-Verfahren

Die Hashfunktion SHA-512 (engl. secure hash algorithm, dt. sicherer Hash-Algorithmus) ist eine der Hashfunktionen die unter dem Oberbegriff SHA-2 zusammengefasst wurden. Hierzu gehören SHA-256/224 und SHA-512/384. Laut Spitz et al., 2011, S. 106 werden bei der SHA-512-Funktion die zu hashenden Quelldaten oder auch Nachrichten (beispielsweise ein Passwort) in 1024-Bit-Blöcke unterteilt, welche aus 16 Wörtern mit jeweils 64 Bit bestehen. Die 16 Wörter werden intern auf 80 Wörter (0, ..., 79) erweitert. Die Funktion arbeitet mit einer Verarbeitungsbreite von 64 Bit über 80 Runden die Nachrichtenblöcke nacheinander (iterativ) ab (siehe Abbildung 2.4).

[11] https://de.wikipedia.org/wiki/Padding_(Informatik), abgerufen am 13.05.2018

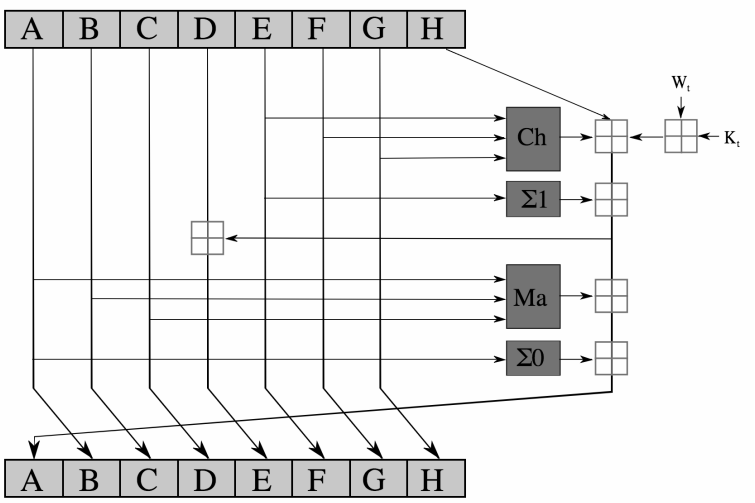

Abbildung 2.4: Aufbau einer Iteration von SHA-2
(Quelle: https://de.wikipedia.org/wiki/SHA-2, abgerufen am 13.05.2018)

Die Nachrichtenblöcke dienen hierbei als Schlüssel für die Verschlüsselung eines Datenblocks von acht Wörtern (A, B, C, D, E, F, G, H). Nach der erfolgreichen Verarbeitung der gesamten Nachricht, gibt die Funktion einen Hashwert von $8 * 64 = 512$ Bit zurück.

2.4.2 Passwort-Hashing-Verfahren

Der folgende Abschnitt beschreibt sogenannte „Passwort-Hashing-Verfahren“, hierbei handelt es sich um Schlüsselableitungsfunktionen (engl. Key Derivation Functions (KDFs)), die den Hashwert eines Passworts oder Passphrase anhand einer Pseudozufallsfunktion ableiten. KDFs dienen dabei dem Zweck die Entropie (siehe Kapitel 2.3.3) eines gewählten Passworts anhand von „Key Stretching“ (dt. Schlüsselstreckung) zu erhöhen. Dies passiert durch die Verwendung eines sogenannten „Salts“ (siehe Kapitel 2.4.4) und einem Iterations-Faktor[12]. Über den Iterations-Faktor wird die Anzahl der durchzuführenden Hashoperationen angegeben. Dies bewirkt, dass nicht nur der Hashwert des eigentlichen Passworts generiert wird, sondern dass anhand dieses ersten Hashwerts weitere Hashwerte (abhängig von der Anzahl der angegebenen Iterationen) auf Basis des jeweils zuvor erzeugten Hashwerts, gebildet werden (siehe Abbildung 2.5).

[12]https://en.wikipedia.org/wiki/Key_derivation_function, abgerufen am 15.05.2018

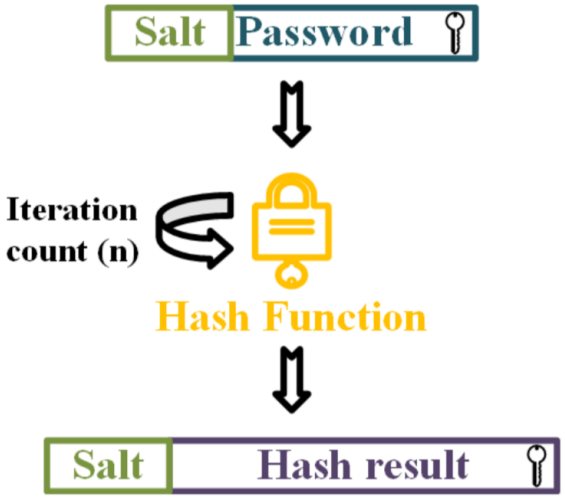

Abbildung 2.5: Schema eines Passwort-Hashing-Verfahrens, Hatzivasilis, Papaefstathiou und Manifavas, 2015, S. 4

Daraus resultierend steigen die „Kosten“ (Zeit und Rechenleistung), die beispielsweise während eines Angriffs anhand der Brute-Force-Methode (siehe Kapitel 2.5.1) benötigt werden, erheblich. Weiterhin werden hierdurch auch Passwortangriffsverfahren, welche Listen vorberechneter Hashwerte einsetzen, wie zum Beispiel Lookup- (siehe Kapitel 2.5.4) oder Rainbow-Tables (siehe Kapitel 2.5.6) unbrauchbar, da der Aufwand alle Hashwerte, für jede mögliche Kombination eines Passworts mit jeder möglichen Kombination eines Salts, im Voraus zu berechnen zu groß wäre. Laut Hatzivasilis, Papaefstathiou und Manifavas, 2015, S. 7 sollten sichere KDFs die folgenden Eigenschaften besitzen:

- Die Funktion darf nicht umkehrbar sein (Einwegfunktion). Es muss unmöglich sein aus einem generierten Hashwert, das dem Hashwert entsprechende Passwort zu errechnen.
- Die Ausgabe der Funktion sollte wie eine zufallsbasierte Kombination von Zeichen wirken.
- Die Funktion sollte möglichst kollisionsresistent (siehe Kapitel 2.4.3) sein.
- Die Funktion sollte immun gegenüber einer sogenannten „Length Extension Attack[13]“ sein.
- Die Funktion sollte abgesichert sein gegen bekannte Angriffstechniken, sowie gegen den Angriff über spezialisierte Hardware (siehe Kapitel 2.5.8)

Im weiteren Verlauf werden die zwei passwortbasierten Schlüsselableitungsfunktionen „bcrypt“ und „scrypt“ vorgestellt und näher beschrieben.

[13]https://en.wikipedia.org/wiki/Length_extension_attack, abgerufen am 20.04.2018

bcrypt-Verfahren

Die kryptologische Hashfunktion bcrypt wurde 1999 von Niels Provos und David Mazières veröffentlicht. Im Kern basiert sie auf dem Blowfish-Algorithmus[14] und wurde speziell für das Hashen und Speichern von Passwörtern entwickelt. Das Betriebssystem OpenBSD setzt bcrypt seit Version 2 als Standard-Passwort-Hashverfahren ein. Im Gegensatz zu regulären Hashingverfahren wurde bcrypt mit dem Ziel entwickelt den Prozess der Hashwertgenerierung möglichst aufwendig zu machen. Die daraus resultierenden Auswirkungen werden jedoch erst spührbar, sobald sehr viele Hashwerte in sehr kurzer Zeit generiert werden, wie beispielsweise bei einem Angriff anhand der Brute-Force-Methode (siehe Kapitel 2.5.1). Um einen 192-Bit-Hashwert zu generieren, benötigt bcrypt als Eingabe einen Kostenfaktor (über den die Anzahl der durchzuführenden Schleifeniterationen angegeben wird), einen 128-Bit-Salt (siehe Kapitel 2.4.4), sowie einen 448-Bit-Schlüssel (Dürmuth und Kranz, 2014). Der Schlüssel enthält das Passwort, welches bis zu 56 Bytes umfassen kann. Durch Wiemer und Zimmermann, 2014 konnte nachgewiesen werden, dass sofern die Energieeffizienz und die Kosten für die Hardware berücksichtigt werden, das bcrypt-Verfahren anhand eines Wörterbuch- oder Brute-Force-Angriffs durch parallel rechnende FPGAs (siehe Kapitel 2.5.8) besonders angreifbar ist.

scrypt-Verfahren

Das Passwort-Hashverfahren scrypt ist eine passwortbasierte Schlüsselableitungsfunktion, die 2010 von Colin Percival entwickelt und veröffentlicht wurde. Das Ziel war es die Schwäche anderer Schlüsselableitungsfunktionen wie beispielsweise PBKDF2 oder bcrypt gegen Passwortangriffe mit spezialisierter Hardware (siehe Kapitel 2.5.8), wie zum Beispiel anhand einer ASIC oder FPGA, zu überwinden[15]. Das Verfahren beruht auf der Grundidee, Angreifer dazu zu zwingen möglichst große Mengen an Arbeitsspeicher für einen Angriff zu verwenden. Dies wird realisiert, indem scrypt pseudozufallsbasierte Zahlen erzeugt, die im Speicher der genutzten Hardware eines Angreifers gespeichert werden. Scrypt greift daraufhin mehrfach auf die abgelegten Zahlen zu, wodurch die Zahlen im Speicher gehalten werden, anstatt sie jedes Mal neu zu erzeugen. Dies fällt zulasten der Hashinggeschwindigkeit. Insbesondere spezialisierte Hardware, wie ASIC (2.5.8) oder eine FPGA (2.5.8), haben hierbei Probleme, da sie nicht über viel Speicher verfügen und somit nur schwer der hohen Speicheranforderung durch scrypt gerecht werden können. Zur Generierung eines scrypt-Hashwerts, muss

- ein Passwort,
- ein Salt,
- eine Ausgabelänge in Bit,
- sowie 3 Kostenparameter,
  - N: CPU- und Speicherfaktor
  - r: Faktor zu Kontrolle der Blockgröße

[14] https://de.wikipedia.org/wiki/Blowfish, abgerufen am 12.05.2018
[15] https://de.wikipedia.org/wiki/Scrypt, abgerufen am 08.05.2018

– p: Parallelfaktor

übergeben werden. Dürmuth und Kranz, 2014 haben bei einer Untersuchung darauf hingewiesen, dass scrypt trotz der, im Gegensatz zu bcrypt, erhöhten Speicheranforderung, bei niedrigen Werten der Parameter für GPUs angreifbarer als bcrypt ist.

2.4.3 Hashwert-Kollision

Bei digitalen Signaturen, vor allem im Bereich der Passwörter ist es wichtig, dass zu zwei unterschiedlichen Passwörtern nicht der gleiche Hashwert generiert werden kann. Laut Paar und Pelzl, 2016, S. 340 bedeutet dies, dass es rechentechnisch unmöglich sein muss, zwei unterschiedliche Nachrichten x1 $\neq$ x2 mit gleichem Hashwert $z1 = h(x1) = h(x2) = z2$ zu erzeugen. Hierbei wird unterschieden zwischen zwei Arten von Kollisionen. Bei der ersten Möglichkeit ist x1 (also ein Passwort) gegeben und es wird versucht, x2 (ein Passwort für das derselbe Hashwert wie für x1 generiert werden kann) zu finden. Wenn dies praktisch unmöglich ist, wird es als zweite Urbildresistenz oder schwache Kollisionsresistenz bezeichnet. Der zweite Fall liegt vor, wenn ein Angreifer sowohl x1 als auch x2 frei wählen kann. Wenn dies für ein Hashverfahren praktisch unmöglich ist, spricht man von einer starken Kollisionsresistenz. Der Unterschied zwischen einer schwachen und einer starken Kollisionsresistenz besteht in der Möglichkeit x1 und x2 frei wählen zu dürfen. Darüber hinaus kann eine Resistenz gegen eine sogenannte „Beinahe-Kollision“ (engl. near-collision resistance) gefordert werden. Dabei soll es praktisch unmöglich sein, zwei unterschiedliche Werte (Passwörter) für x1 und x2 zu finden, deren Hashwerte h(x1) und h(x2) sich nur in wenigen Bits unterscheiden[16].

2.4.4 Salt

Bevor ein Hashwert für das von einem Nutzer gewählte Passwort erzeugt wird, kann dem Passwort eine zufällige Zeichenkette, ein sogenannter „Salt“ hinzugefügt werden. Für jedes Passwort eines jeden Nutzers sollte ein einzigartiger Salt erzeugt werden. Hierdurch kann sichergestellt werden, dass für jeden Nutzer ein individueller Hashwert für das gewählte Passwort gespeichert wird, selbst wenn mehrere Nutzer das gleiche Passwort gewählt haben. Es wird empfohlen ausreichend lange Salts zu erzeugen, da sie, wenn sie an ein Passwort angehangen werden, einerseits die Sicherheit (Entropie, siehe Kapitel 2.3.3) eines Passworts weiter erhöhen und anderseits es schwerer machen Hashwerte im Voraus (offline) zu berechnen. Durch die Verwendung einzigartiger, ausreichend langer Salts, werden vorberechnete Lookup- oder Rainbow-Tables (siehe Kapitel 2.5.6) unbrauchbar (Provos und Mazières, 1999), da ein Angreifer nicht im Voraus weiß, welcher Nutzer welchen Salt erhalten hat, wodurch sich der ohnehin schon große Aufwand der Vorausberechnung aller möglichen Hashwerte und Passwörter vervielfacht.

2.4.5 Pepper

Neben einem Salt stellt „Pepper“ eine weitere Möglichkeit dar, die Sicherheit eines Hashwerts zu erhöhen. Genau wie ein Salt, ist Pepper ein weiterer Wert, der -

[16] https://de.wikipedia.org/wiki/Kryptographische_Hashfunktion, abgerufen am 03.05.2018

bevor ein Hashwert erzeugt wird - einem Passwort hinzugefügt wird. Salt und Pepper unterscheiden sich in sofern, dass ein Pepper-Wert nicht zusammen mit dem Hashwert gespeichert und für jeden Hashwert neu und zufällig generiert wird. Der String, aus dem der Pepper-Wert gebildet wurde, sollte an einem anderen Ort als dem Passwort-Hash gespeichert werden, wohingegen der Pepper-Wert selbst niemals gespeichert werden sollte, sondern immer zufällig aus einer Teilmenge von Werten („Pepper-String“) generiert werden. Bei einem Passwortabgleich wird der gespeicherte Hashwert mit dem übermittelten Hashwert, bestehend aus dem vorgeschlagenen Passwort, dem Salt und jedem möglichen Wert des Peppers, verglichen. Es werden so lange Vergleiche mit möglichen Werten des Peppers gemacht, bis ein Vergleich entweder erfolgreich ist oder fehlschlägt. Im Allgemeinen besteht der Grund für die Verwendung eines Peppers darin, die hinzugefügten Zeichen und Symbole dazu zu verwenden, ein schwaches Passwort sicherer zu machen. Hierdurch wird die Passwortlänge erhöht und die Komplexität (durch das Hinzufügen von Sonderzeichen und anderen Zeichenarten) verbessert. Weiterhin wird der resultierende Hashwert einzigartiger, wodurch Wörterbuchangriffe erschwert werden (LeBlanc und Messerschmidt, 2016).

2.5 Passwortangriffsverfahren

Im folgenden Abschnitt wird eine Auswahl von Verfahren und Techniken zum Angriff auf Passwörter vorgestellt und anhand von Beispielen beschrieben. Der Fokus liegt hierbei auf einer Auswahl der bekannteren Verfahren, da eine Beschreibung aller existierenden Verfahren den Umfang dieser Arbeit übersteigen würde.

2.5.1 Brute-Force-Methode

Bei einem Brute-Force-Angriff wird dem Namen entsprechend rohe Gewalt benutzt um ein Passwort zu ermitteln. Abhängig von der Zusammensetzung und Länge eines Passworts werden bei einem Brute-Force-Angriff alle Zeichenkombinationen (Groß-/Kleinbuchstaben, Zahlen und Sonderzeichen) für ein gesuchtes Passwort mit n-Stellen getestet (aaaa, aaab, aaac, ...). Der größte Nachteil dieser Angriffsart ist ihr Zeitaufwand. Je länger und komplexer ein gesuchtes Passwort ist, umso mehr Zeit wird benötigt um alle Kombinationen zu ermitteln (Schneier, 2000, S. 100). Beispielsweise enthält der Namensraum einer vierstelligen PIN, die nur Zahlen als mögliche Zeichen erlaubt, 10.000 Einträge („0000“ bis „9999“). Angenommen der gesuchte PIN wäre „9999“, so müssten zuvor 9.999 andere Kombinationen geprüft werden, bis das richtige Ergebnis gefunden wird (Carus, 2008, S. 250). Der Faktor Zeit ist hierbei stark abhängig von der Leistung des zum Errechnen der Kombinationen eingesetzten Computersystems. Je mehr Rechenleistung ein System hat, umso schneller kann ein gesuchtes Passwort gefunden werden. Abbildung 2.6 zeigt hierzu in einer tabellarischen Übersicht die Zeit, die ein Brute-Force-Angriff maximal benötigt um mit einem System, welches 1 Milliarde MD5-Hashwerte innerhalb einer Sekunde errechnen kann, ein Passwort zu ermitteln. Hierbei wird deutlich, dass die Länge und Zusammensetzung eines Passworts sich exponentiell auf die Zeit, die benötigt wird um das Passwort zu ermitteln, auswirkt. Ein 12-stelliges Passwort welches nur aus Zahlen besteht, kann laut Tabelle innerhalb von 17 Minuten ermittelt werden, wohingegen

ein 12-stelliges Passwort welches den gesamten Zeichenraum (96 Zeichen) ausnutzt, ungefähr 19 Millionen Jahre benötigt um ermittelt zu werden.

Maximale Rechenzeit eines Brute-Force-Angriffs bei 1 Milliarde Schlüsseln pro Sekunde

	Passwortlänge								
Zeichenraum	**4 Zeichen**	**5 Zeichen**	**6 Zeichen**	**7 Zeichen**	**8 Zeichen**	**9 Zeichen**	**10 Zeichen**	**11 Zeichen**	**12 Zeichen**
10 [0-9]	<1 ms	<1 ms	1 ms	10 ms	100 ms	1 Sekunde	10 Sekunden	2 Minuten	17 Minuten
26 [a-z]	<1 Sekunde	<1 Sekunde	<1 Sekunde	8 Sekunden	4 Minuten	2 Stunden	2 Tage	42 Tage	3 Jahre
52 [A-Z;a-z]	<1 Sekunde	<1 Sekunde	20 Sekunden	17 Minuten	15 Stunden	33 Tage	5 Jahre	238 Jahre	12.400 Jahre
62 [A-Z;a-z;0-9]	<1 Sekunde	<1 Sekunde	58 Sekunden	1 Stunde	3 Tage	159 Tage	27 Jahre	1.649 Jahre	102.000 Jahre
96 (+Sonderzeichen)	<1 Sekunde	8 Sekunden	13 Minuten	21 Stunden	84 Tage	22 Jahre	2.108 Jahre	202.000 Jahre	19 Mio Jahre

Abbildung 2.6: Brute-Force-Rechenzeit eines MD5-Hashwerts
(Quelle: https://de.wikipedia.org/wiki/Passwort)

2.5.2 Wörterbuchangriff

Die Basis eines Wörterbuchangriffs (engl. Dictionary Attack) ist eine Liste vorgegebener Passwörter. Ein Angreifer probiert hierbei jedes Benutzerkonto eines Zielsystems in Kombination mit jedem Passwort des gewählten Wörterbuchs aus und probiert somit sich erfolgreich an einem Zielsystem anzumelden. Wenn eines der Passwörter des Wörterbuchs mit einem der Passwörter der Benutzerkonten übereinstimmt, hat ein Angreifer einen gültigen Login gefunden und kann sich mit den ermittelten Nutzerdaten am Zielsystem anmelden (Carus, 2008, S. 250). Laut Raza et al., 2012, S. 2 stellt diese Art des Passwortangriffs, im Gegensatz zur Brute-Force-Methode, eine wesentlich intelligentere und schnellere Methode zur Ermittlung eines Passworts dar. Der deutsche Wortschatz kann bis zu 500.000 Wörter umfassen, die englische Sprache kennt ca. 600.000 Wörter[17]. Ausgehend davon, dass Passwörter häufig direkt oder mit leichten Variationen oder Erweiterungen aus Wörtern gebildet werden, die in einem Wörterbuch zu finden sind, ist die Zahl der in Frage kommenden Passwörter relativ gering. Studien belegen zudem, dass viele Angriffe auf Passwörter häufig mit dem Versuch beginnen ein Passwort beispielsweise im Bereich der Namen, Geburtstage oder auch Kosenamen zu erraten (Ur et al., 2015, S. 1). Beispielsweise ist die Zahl möglicher Geburtsdaten in ihrer vollen Darstellung als achtstellige Zahl im Schema „ttmmjjjj“ deutlich kleiner als die Zahl möglicher Nummernkombinationen. So kann durch geschickte Wahl möglicher Passwörter der Suchraum mit erheblichen Erfolgsaussichten auf tabellierte Wörter und Zahlenfolgen, sowie deren Kombination und einfache Abwandlungen, eingeschränkt werden.

2.5.3 Online-/Offline-Angriffe

Bei Brute-Force- aber auch Wörterbuchangriffen muss unterschieden werden zwischen Angriffen auf Anmeldemasken laufender Systeme wie beispielsweise ein Login-Fenster einer Webseite und Offline-Angriffen auf gestohlene Nutzerdaten (in Form von Hashwerten). Während bei Online-Angriffen weitere Sicherheitsmaßnahmen eines Systems zusätzlich Schutz bieten (z.B. limitierte Anzahl von Anmeldeversuchen, Wartezeit zwischen den Login-Versuchen („Ausbremsen“) oder Captchas), können Angreifer bei Offline-Angriffen beliebig viele Versuche unbeobachtet und ungehindert

[17] https://de.wikipedia.org/wiki/Wortschatz, abgerufen am 22.04.2018

durchführen (Weir, Aggarwal, Collins und Stern, 2010, S. 10). Bei einem Online-Angriff auf ein laufendes System wird daher häufig ein eher (schwaches) Passwort auf mehreren verschiedenen Accounts getestet, mit der Hoffnung, dass mindestens ein Nutzer ein unsicheres Passwort gewählt hat.

2.5.4 Lookup-Tables

Um das Performance-Problem von Brute-Force-Angriffen zu umgehen, können „Lookup-Tables“ (dt. Wertetabelle) genutzt werden. Eine Lookup-Table ist eine spezielle Datenstruktur, die vorberechnete Passwort-Hash-Pärchen enthält. Das Besondere an Lookup-Tables ist, dass sie selbst bei einem großen Datenbestand von mehreren Millionen Paaren immer noch mehrere hundert Abfragen pro Sekunde verarbeiten können. Das macht Lookup-Tables zwar schneller als einen reinen Brute-Force-Angriff, jedoch sind sie, ähnlich einem Wörterbuchangriff, durch die vorberechneten Werte auf eine vorbestimmte Menge an Eingaben (Passwörtern) begrenzt. Schneller wird ein solcher Angriff, wenn ein Angreifer zuvor Datensätze mit Nutzerdaten (Hashwerten von Passwörtern) gestohlen hat. In diesem Fall kann ein Angreifer einen gestohlenen Hashwert in einer passenden Lookup-Table einfach suchen und erhält, sofern der gesuchte Hashwert Teil der Lookup-Table ist, das dem Hashwert entsprechende Passwort. Der Aufwand bei einer Lookup-Table entsteht nicht während eines Angriffs, sondern im Vorhinein. Bevor eine Lookup-Table für einen Angriff auf ein oder mehrere Passwörter genutzt werden kann, muss diese zuerst erstellt werden. Dazu muss für jedes Passwort eines Wörterbuchs, welches während eines Angriffs getestet werden soll, zuvor einen Hashwert errechnet werden, der dann, zusammen mit dem entsprechenden Klartextpasswort, tabellarisch in einer Liste abgelegt wird. Entgegen dem großen Geschwindigkeitsvorteil während eines Angriffs, haben Lookup-Tables den Nachteil, dass sie sehr schnell sehr viel Speicherplatz auf einem System einnehmen können. Durch die Verwendung eines Salts (siehe Kapitel 2.4.4), während der Hashgenerierung, werden Lookup-Tables unbrauchbar, da der Aufwand, bei der Erstellung einer Lookup-Table jeden möglichen Salt-Wert zu beachten zu groß wäre.

2.5.5 Reverse-Lookup-Tables

Bei einem Angriff anhand einer „Reverse-Lookup-Table“ erstellt ein Angreifer zuerst eine Wertetabelle (engl. Lookup-Table), in der jeder Passwort-Hashwert einer kompromittierten Nutzerdatenbank einer Webseite dem entsprechenden Nutzernamen zugeordnet wird. Im Anschluss führt der Angreifer einen regulären Wörterbuch- oder Brute-Force-Angriff durch, bei dem die zuvor erstellte Wertetabelle aus Hashwerten und Nutzernamen referenziert wird. Das Resultat ist eine Liste von Nutzernamen und ihrer Klartextpasswörter. Diese Art des Angriffs ist besonders effektiv, da Nutzer häufig gleiche Passwörter verwenden, wodurch dem ermittelten Passwort eines Hashwerts mehrere Nutzernamen innerhalb der sogenannten „Reverse-Lookup-Table“ zugeordnet werden können. Der beste Schutz gegen diese Art von Angriff ist die Verwendung eines einzigartigen, ausreichend langen Salts (2.4.4) bei jedem Passwort eines Systems oder einer Webseite (LeBlanc und Messerschmidt, 2016, S. 40). Durch das Hinzufügen unterschiedlicher Salts zu jedem Passwort, bevor für ein Passwort ein Hashwert generiert wurde, wird sichergestellt, dass selbst für gleiche Passwörter unterschiedliche Hashwerte errechnet werden.

2.5.6 Rainbow-Tables

Während Brute-Force-Angriffe viel Zeit beziehungsweise Rechenleistung benötigen, werden Dictionary-Angriffe hauptsächlich vom verfügbaren Speicherplatz begrenzt. Einen Kompromiss aus beiden Angriffen (auch „time-memory trade-off“ genannt) stellen sogenannte „Rainbow-Tables“ dar. Mit ihnen können Passwörter zu bekannten Hashwerten in akzeptabler Zeit gefunden werden, ohne alle Hashwerte speichern zu müssen, wie beispielsweise bei einer Lookup-Table. Eine Rainbow-Table ist eine von Philippe Oechslin entwickelte Datenstruktur, die eine speichereffiziente und vor allem schnelle Suche nach einem Passwort für einen gegebenen Hashwert ermöglicht[18]. Um eine solche Tabelle zu erstellen, müssen zunächst lange Ketten aus miteinander korrespondierenden Kombinationen von Passwörtern und Hashwerten gebildet werden. Diese Ketten beginnen mit einem beliebig gewählten Passwort, für das ein Hashwert erzeugt wird. Aus diesem Hashwert entsteht durch eine sogenannte Reduktionsfunktion ein neues mögliches Passwort, auf das erneut die Hashfunktion angewendet wird. Dies wird so lange wiederholt, bis viele unterschiedliche, kollisionsfreie Ketten entstehen, zu denen jeweils mehrere tausend Elemente gehören. Gespeichert werden bei jeder Kette jeweils nur das erste und letzte Element (Oechslin, 2003, S. 5–6). Wird zu einem Hashwert das entsprechende Passwort gesucht, wird der Hashwert so lange mittels Reduktions- und Hash-Funktion umgewandelt, bis er dem letzten Element, dem „Ketten-Ende“, entspricht. Wenn so die richtige Kette gefunden wurde, muss diese nun erneut berechnet werden, um die richtige Kombination von Passwort und Hashwert zu erhalten. Durch den immensen Aufwand, der bei der Erstellung solcher Rainbow-Tables entsteht, lohnt ihr Einsatz nur, wenn sie für mehr als einen Angriff genutzt werden können. Die Verwendung eines Salts beim Erzeugen von Hashwerten ist die beste Verteidigung gegen eine solche Art des Angriffs. Eine Rainbow-Table die bei den vorberechneten Hashwerten Salts mit einbezieht, würde einerseits hohe zeitliche Kosten bei ihrer Erstellung verursachen und darüber hinaus eine immense Größe an Speicher belegen und andererseits würde, da entsprechend viele Ketten gebildet werden müssten, ein entsprechend großer zeitlicher Aufwand entstehen eine solche Rainbow-Table nach einer Kombination aus Hashwert und Passwort zu durchsuchen. Dies würde den ursprünglichen Vorteil des „time-memory trade-offs“ zunichte machen.

2.5.7 GPU-basierte Angriffe

Der eigentliche Zweck einer Graphics Processing Unit (GPU) ist es 3D-Berechnungen parallel auszuführen und diese so zu beschleunigen. Durch ihre, im Vergleich zu einer CPU, wesentlich höheren Anzahl an Kernen und die Möglichkeit parallelisierte Berechnungen durchzuführen eignet sich eine GPU sehr gut für die Berechnung von Hashingverfahren. Der Hauptunterschied zwischen CPU- und GPU-basierten Angriffen ist die Geschwindigkeit, in der Hashwerte berechnet werden können. Moderne GPUs sind bei einem Angriff auf ein Passwort mehr als 100-mal schneller als eine moderne CPU (Kasabov und van Kerkwijk, 2011). Die zuvor beschriebenen Angriffstechniken,

- Lookup-Tables / Reverse-Lookup-Tables (Kapitel 2.5.4 und 2.5.5),

[18] https://de.wikipedia.org/wiki/Rainbow_Table, abgerufen am 04.05.2018

- Rainbow-Tables (Kapitel 2.5.6),
- Brute-Force-Methode (Kapitel 2.5.1),
- und Wörterbuchangriffe (Kapitel 2.5.2)

können alle anhand eines GPU-Angriffs durchgeführt werden.

2.5.8 Angriffe anhand von spezialisierter Hardware

Durch den Einsatz von spezialisierter Hardware, wie ASICs (dt. Anwendungsspezifische Integrierte Schaltungen) oder auch FPGAs (engl. Field Programmable Gate Array), ist es möglich die Geschwindigkeit eines Angriffs, im Gegensatz zur Geschwindigkeit der Hashwert-Berechnung bei GPUs (und vor allem im Vergleich zu CPUs), massiv zu erhöhen. Darüber hinaus ist der Energieverbrauch solcher Hardware wesentlich effizienter als bei GPUs (Dürmuth und Kranz, 2014, S. 12).

FPGAs

FPGAs sind preiswerte, programmierbare integrierte Schaltkreise (ICs) in der Digitaltechnik, die viele Funktionalitäten direkt mitbringen und verhältnismäßig einfach durch den Nutzer selbst zu programmieren sind.

ASICs

Demgegenüber erfordern ASICs einen sehr hohen Entwicklungsaufwand mit entsprechend hohen Kosten. Jedoch arbeiten ASICs auch wesentlich effizienter (ca. 10-mal schneller) als FPGAs (Eckert, 2008).

2.6 Software hashcat

In diesem Abschnitt des Kapitels wird auf die Software `hashcat` eingegangen. Nach einer kurzen Einführung in die Software, bei der auf die Hauptfunktionsweise, sowie ihre Stärken und Schwächen eingegangen wird, folgt ein Abschnitt über die durch `hashcat` unterstützten Hashingverfahren. Zum Schluss folgt eine Beschreibung der in `hashcat` implementierten Funktionen zum Angreifen von Passwörtern.

2.6.1 Einführung in die Software

`Hashcat` ist eine unter der Massachusetts Institute of Technology (MIT) Lizenz lizensierte, frei verfügbare, OpenSource Software. Sie wird häufig als ein „Passwortwiederherstellungstool“ deklariert und laut der Webseite der Entwickler, ist es die aktuell schnellste Software dieser Art[19]. Allerdings ist `hashcat` im eigentlichen Sinne kein „Wiederherstellungstool“, da es nichts wiederherstellt. Egal, welche der durch `hashcat` unterstützen Funktionen ein Nutzer, zum „Ermitteln“ eines Passworts wählt, der Funktionsablauf bleibt immer derselbe. `Hashcat` muss, unabhängig von der gewählten Angriffsfunktion, immer ein Hashwert übergeben werden, dessen Passwort

[19] https://hashcat.net/hashcat/, abgerufen am 08.05.2018

dann durch `hashcat` ermittelt wird. Da allerdings kryptografische Hashfunktionen (siehe Kapitel 2.4.1) die Anforderung haben eine sogenannte „Einwegfunktion“ zu sein, kann aus einem erzeugten Hashwert nicht die ursprüngliche Information (das Passwort) zurück errechnet werden. Wie also ermittelt `hashcat` ein Passwort? `Hashcat` erzeugt, abhängig von der gewählten Angriffsfunktion, Passwörter und generiert deren Hashwerte. Die Hashwerte werden dann nacheinander mit dem Ziel-Hashwert verglichen, bis `hashcat` entweder ein Passwort gefunden hat, dessen Hashwert mit dem Ziel-Hashwert übereinstimmt oder keine weiteren Passwörter generieren kann. Das Problem hierbei ist, dass nicht nachgewiesen werden kann, ob es sich bei dem gefunden Passwort um das ursprüngliche Original-Passwort handelt. Wie in Kapitel 2.4.3 beschrieben, bringen Hashfunktionen die Gefahr einer Hashwert-Kollision mit sich. Auch wenn aktuelle kryptologische Hashverfahren möglichst kollisionsresistent sein sollten, ist es theoretisch möglich, dass `hashcat` bei einem Angriff auf ein Passwort nicht das ursprüngliche Passwort findet, sondern lediglich ein Passwort, welches den gleichen Hashwert wie der Ziel-Hashwert aufweist. Da in einem solchen Fall, bei einer Authentifikation an einem System, anhand des ermittelten Passworts, derselbe Hashwert generiert wird, wie der ursprüngliche Ziel-Hashwert, kann ein Nutzer erfolgreich an dem System angemeldet werden, unabhängig davon ob es sich bei dem übergebenen Passwort um das ursprüngliche Original-Passwort handelt oder nicht. Aus diesem Grund kann `hashcat` weniger als ein „Passwortwiederherstellungstool“, als mehr als ein „Passwort-Cracker-Tool“ bezeichnet werden. `Hashcat` benötigt immer einen angreifbaren Ziel-Hashwert, daher kann es ausschließlich für Offline-Angriffe (siehe Kapitel 2.5.3), verwendet werden. Hierbei, beziehungsweise in diesem Bereich, bringt es jedoch sehr viele Funktionen und Vorteile mit sich, unter anderem:

- `Hashcat` ist kompatibel zu allen großen Betriebssystemen (Windows, Mac Os und Linux)
- es ist „multi-thread-fähig“ und kann daher mehrere CPU's/GPU's gleichzeitig für einen Angriff nutzen
- es unterstützt die meisten gängigen Hashverfahren
- ein laufender Angriff kann pausiert und später fortgesetzt werden
- und noch vieles mehr

2.6.2 Unterstützte Hashingverfahren

`Hashcat` unterstützt aktuell über 200 verschiedene Hashverfahren, darunter diverse Algorithmen die verschiedene Hashverfahren miteinander kombinieren. Diese Kombinationen dienen dazu, den Angriff auf Hashwerte spezieller Umgebungen oder Anwendungen zu vereinfachen. Zu diesen Verfahren und Algorithmen gehören zum Beispiel:

- MD5
- SHA-1
- SHA-512
- SHA-3 (Keccak)

- scrypt
- WPA/WPA2
- bcrypt
- macOS v10.8+ (PBKDF2-SHA512)
- Atlassian (PBKDF2-HMAC-SHA1)
- WordPress (MD5)

Die Liste aller aktuell durch `hashcat` unterstützten Verfahren wird im Anhang B.1 noch einmal im Ganzen dargestellt[20].

Mögliche Konfigurationen

Die folgenden Abschnitte beschreiben `hashcat's` Angriffsfunktionen und deren möglichen Konfigurationen.

2.6.3 Dictionary-Attack

Die folgenden Informationen basieren einerseits auf den gegebenen Informationen der Entwickler auf der `hashcat`-Webseite [21] und andererseits auf im Zuge dieser Arbeit gewonnenen Erkenntnissen.
`Hashcat's` „Dictionary-Attack“ (dt. Wörterbuchangriff), wird in `hashcat` auch als „straight mode“ bezeichnet und entspricht dem in Kapitel 2.5.2 beschriebenen Wörterbuchangriff. Dementsprechend ist die Voraussetzung dieses Angriffs nur eine Textdatei, in der zeilenweise Passwörter aufgelistet sind. Wichtig hierbei ist, dass jede Zeile der Wörterbuchdatei nur jeweils ein Passwort listet, da `hashcat` während des Angriffs das Wörterbuch zeilenweise ausliest und somit alle Zeichen einer Zeile als ein Passwort erkennt. Weiterhin wird für den Angriff ein anzugreifender Ziel-Hashwert und das dazu passende Hashverfahren benötigt. Um eine Dictionary-Attack mit `hashcat` zu starten, müssen dem Programm die folgenden Parameter beim Start übergeben werden:

- -a 0
  - Der Parameter gibt `hashcat`, anhand des zum Parameter übergebenen Zahlenwerts, den Angriffsmodus vor. Der Wert „0“ steht hierbei für den straight mode oder auch Dictionary-Attack.
- -m [HashmodeNumber]
  - Anhand dieses Parameters wird `hashcat` das Hashverfahren des anzugreifenden Ziel-Hashwerts übergeben. Für den Platzhalter [HashmodeNumber] ist die dem Hashverfahren entsprechende Nummer anzugeben. Eine Tabelle, der durch `hashcat` unterstützten Hashverfahren und den den Verfahren zugeordneten Nummern kann über die Hilfeseite von `hashcat` aufgerufen

[20] https://hashcat.net/hashcat/, abgerufen am 08.05.2018
[21] https://hashcat.net/wiki/doku.php?id=dictionary_attack, abgerufen am 09.05.2018 f.

werden. Der Befehl hierzu lautet „hashcat -h“. Zusätzlich listet die Tabelle in Anlage B.1 alle unterstützten Hashverfahren und die ihnen entsprechenden Nummern auf.

- Pfad/zu/einer/Hashdatei
  - Hier wird `hashcat` entweder der Festplattenpfad zu einer Datei mit einem Hashwert übergeben oder ein einzelner Hashwert. Der Vorteil einer Hashdatei liegt darin, dass auch diese Datei von `hashcat` Zeilenweise abgearbeitet wird, wodurch es möglich ist nacheinander mehrere Hashwerte mit nur einem Angriff zu ermitteln.
- Pfad/zu/einer/Wörterbuchdatei
  - Zum Schluss muss `hashcat` ein Festplattenpfad zu einer Wörterbuchdatei übergeben werden.

Hieraus resultiert der folgende Befehl zum Start einer Dictionary-Attack in `hashcat`:

```
C:\Program Files (x86)\hashcat-4.0.1>hashcat64 -a 0 -m [HashmodeNumber]
   Pfad/zu/einer/Hashdatei Pfad/zu/einer/Wörterbuchdatei
```

2.6.4 Combinator-Attack

Die folgenden Informationen basieren einerseits auf den gegebenen Informationen der Entwickler, zu finden auf der `hashcat`-Webseite[22] und andererseits auf im Zuge dieser Arbeit gewonnen Erkenntnissen.
Die sogenannte „Combinator-Attack“ in `hashcat` unterscheidet sich im Grunde kaum von der zuvor beschriebenen Dictionary-Attack. Der einzige Unterschied liegt in dem Umstand, dass diese Angriffsart die Verwendung von zwei Wörterbüchern zulässt. Bei jedem Angriffsversuch wählt `hashcat` ein Passwort aus jedem der beiden Wörterbücher aus und kombiniert beide Wörter miteinander zu einem neuen Passwort. Hierzu muss `hashcat` anhand der folgenden Parameter gestartet werden:

- -a 1
  - für den Angriffsmodus wird `hashcat` der Wert „1“ übergeben, dies startet die Combinator-Attack
- -m [HashmodeNumber]
  - Übergabe der Nummer des Ziel-Hashwerts (Anlage B.1)
- Pfad/zu/einer/Hashdatei
  - Übergabe des Festplattenpfads zur Hashdatei
- Pfad/zur/Wörterbuchdatei1
  - Angabe des Pfades auf der Festplatte zur ersten Wörterbuchdatei

[22] https://hashcat.net/wiki/doku.php?id=combinator_attack, abgerufen am 09.05.2018 f.

- Pfad/zur/Wörterbuchdatei2
  - Angabe des Pfades auf der Festplatte zur zweiten Wörterbuchdatei

Der Startaufruf einer Combinator-Attack lautet dann:

```
C:\Program Files (x86)\hashcat-4.0.1>hashcat64 -a 1 -m [HashmodeNumber]
   Pfad/zu/einer/Hashdatei Pfad/zur/Wörterbuchdatei1
   Pfad/zur/Wörterbuchdatei2
```

Hierzu das folgende Beispiel, bei dem zwei Wörterbücher kombiniert werden:

- Wörterbuch 1
  - Gelbes
  - Blaues
  - Rotes
  - Schwarzes
- Wörterbuch 2
  - Auto
  - Fahrad

Wenn diese beiden Wörterbücher nun für eine Combinator-Attack verwendet werden, entstehen dabei die folgenden Passwörter:

- Passwörter
  - GelbesAuto
  - BlauesAuto
  - RotesAuto
  - SchwarzesAuto
  - GelbesFahrrad
  - BlauesFahrrad
  - RotesFahrrad
  - SchwarzesFahrrad

Darüber hinaus kann jedem der Wörter eine sogenannte Regel übergeben werden, um ein Sonderzeichen an jedes der zu kombinierenden Wörter zu hängen. Die Parameter hierfür lauten:

```
 -j, --rule-left=Rule  Regel wird auf jedes Wort des linken Wörterbuchs
                       angewendet

 -k, --rule-right=Rule Regel wird auf jedes Wort des rechten
                       Wörterbuchs angewendet
```

Wenn diese Parameter auf beide Wörterbücher angewendet werden um beispielsweise einen Bindestrich (-) und ein Ausrufezeichen (!) an die Passwörter anzuhängen, so lautet die entsprechende Befehlssyntax

```
C:\Program Files (x86)\hashcat-4.0.1>hashcat64 -a 1 -m [HashmodeNumber]
    Pfad/zu/einer/Hashdatei Pfad/zur/Wörterbuchdatei1
    Pfad/zur/Wörterbuchdatei2 -j '$-' -k '$!'
```

und die daraus entstehenden Passwörter lauten:

- Passwörter
  - Gelbes-Auto!
  - Blaues-Auto!
  - Rotes-Auto!
  - Schwarzes-Auto!
  - Gelbes-Fahrrad!
  - Blaues-Fahrrad!
  - Rotes-Fahrrad!
  - Schwarzes-Fahrrad!

Die Combinator-Attack ist ein einfaches Werkzeug um die Wörter von zwei Wörterbüchern zu möglichen Passwortkandidaten zu kombinieren, sie setzt somit da an, wo ein regulärer Wörterbuchangriff an seine Grenzen stößt. Wie in Kapitel 2.3.2 beschrieben, werden Passwörter mehr und mehr aus mehr als einem Wort gebildet, weshalb ein einfacher Wörterbuchangriff mit Passwörtern bestehend aus einem Wort sinnlos wäre. Einem Angreifer bleiben in einem solchen Fall nur zwei Möglichkeiten. Entweder muss ein Wörterbuch erzeugt werden, welches auch Passwörter enthält, die aus mehr als einem Wort bestehen (was bei der Kombination von entsprechend vielen Wörtern, sehr schnell, sehr viel Festplattenspeicherplatz einnehmen kann), oder ein Angriff muss so konzipiert werden, dass Wörter aus mehr als einem Wörterbuch miteinander kombiniert werden. Daraus wird ein Nachteil der `hashcat`-Combinator-Attack offensichtlich, da der Angriff die Kombination der Wörter von nur exakt zwei Wörterbüchern zulässt. Bei einer größeren Passphrase (bestehend aus mehr als zwei Wörtern), wäre ein solcher Angriff unzureichend.

2.6.5 Brute-Force-Methode / Mask-Attack

Im nachfolgenden Abschnitt wird `hashcat's` „Mask-Attack“ beschrieben, die Informationen hierzu basieren einerseits auf den Informationen der entsprechenden Webseite der `hashcat` Entwickler[23] und andererseits auf im Zuge dieser Arbeit gewonnen Erkenntnissen.
Die `hashcat`-eigene Version der Brute-Force-Methode nennt sich „Mask-Attack“. Wie in Kapitel 2.5.1 beschrieben ist die Brute-Force-Methode nichts anderes als das Durchprobieren aller möglichen Zeichenkombinationen, bis die richtige Kombination und somit das gesuchte Passwort gefunden wird. Auch wenn diese Art des Angriffs immer

[23] https://hashcat.net/wiki/doku.php?id=mask_attack, abgerufen am 09.05.2018 f.

(irgendwann) das gesuchte Passwort findet, so ist sie doch sehr zeit- und rechenaufwendig. `Hashcat's` Version der Brute-Force-Methode, die Mask-Attack, testet zwar ebenfalls alle möglichen Zeichenkombinationen durch um ein Passwort zu finden, verfolgt dabei jedoch einen anderen Ansatz. Mit der Mask-Attack ist es möglich übliche Passwortschemata anhand einer Maske abzubilden, um so die Menge an zu überprüfenden Kombinationen drastisch zu reduzieren. Wenn zum Beispiel das Passwort „Julia1984“ ermittelt werden soll, würden bei der Brute-Force-Methode alle möglichen Kombinationen von Zeichen durchprobiert werden. Durch die Mask-Attack kann das Schema des Passworts definiert werden, um beispielsweise den ersten Buchstaben als Großbuchstaben und die restlichen Buchstaben als Kleinbuchstaben zu definieren. Weiterhin ist es möglich festzulegen, dass ein Teil des Passworts aus Zahlen besteht (wie im angegebenen Beispielpasswort). Auf diese Art und Weise können eine große Menge an Kombinationen ausgeschlossen und die Rechenzeit zum Ermitteln des Passworts erheblich verkürzt werden.
Der Aufbau einer solchen Maske besteht aus Fragezeichen in Kombination mit statischen Zeichen (angenommen man weiß, dass der erste Buchstabe ein „R“ ist) oder aber sogenannten Zeichensatzvariablen, über die der jeweiligen Zeichenposition verschiedene Zeichen übergeben werden können. Die Anzahl der Fragezeichen definiert hierbei gleichzeitig die Länge eines gesuchten Passworts, sodass beispielsweise die Maske eines Passworts mit einer Länge von acht Zeichen aus entsprechend vielen Fragezeichen besteht „????????“. `Hashcat` gibt einem Nutzer für die Definition einer Maske die Option aus acht vordefinierten Zeichensatzvariablen zu wählen oder aber bis zu vier Zeichensatzvariablen selbst zu definieren. Die über `hashcat` vordefinierten Zeichensatzvariablen sind:

```
?l = abcdefghijklmnopqrstuvwxyz
?u = ABCDEFGHIJKLMNOPQRSTUVWXYZ
?d = 0123456789
?h = 0123456789abcdef
?H = 0123456789ABCDEF
?s = <<space>>!"#$%&'()*+,-./:;<=>?@[\]^_`{|}~
?a = ?l?u?d?s
?b = 0x00 - 0xff
```

Dementsprechend ist die Maske des Passworts „Julia1984“ nach dem folgenden Schema „?u?l?l?l?l?d?d?d?d“ aufgebaut. Anhand dieses Schemas wird klar definiert an welcher Postion im Passwort Großbuchstaben (?u), Kleinbuchstaben (?l) oder Zahlen (?d) erwartet werden. Um anhand dieses Beispiels eine Mask-Attack über `hashcat` zu starten, muss `hashcat` mit den folgenden Parametern aufgerufen werden:

- -a 3
  - für den Angriffsmodus wird `hashcat` der Wert „3“ übergeben, dies ist das Signal für `hashcat` eine Mask-Attack zu starten
- -m [HashmodeNumber]
  - Übergabe der Nummer des Ziel-Hashwerts (Anlage B.1)
- ?u?l?l?l?l?d?d?d?d

  - Definition einer Maske, die einerseits die Länge aber auch den Aufbau bzw. die Zusammensetzung des gesuchten Passworts abbildet

- Pfad/zu/einer/Hashdatei
  - Übergabe des Festplattenpfads zur Hashdatei

Der Startbefehl der Mask-Attack lautet:

```
C:\Program Files (x86)\hashcat-4.0.1>hashcat64 -a 3 -m [HashmodeNumber]
   ?u?l?l?l?l?d?d?d?d Pfad/zu/einer/Hashdatei
```

Wie bereits erwähnt, kann ein Nutzer ebenso selbst eine Zeichensatzvariable definieren. Hierzu muss anhand der Variablen „-1“, „-2“, „-3“ oder „-4“, eine benutzerdefinierte Zeichensatzemenge gebildet werden. Dies kann entweder durch das direkte Angeben von Zeichen geschehen, oder durch das Kombinieren der durch `hashcat` vordefinierten Zeichensatzvariablen wie z.B.:

```
-1 abcdefghijklmnopqrstuvwxyz0123456789
-1 abcdefghijklmnopqrstuvwxyz?d
-1 ?l0123456789
-1 ?l?d
```

In diesem Fall sieht der Startbefehl der Mask-Attack wie folgt aus:

```
C:\Program Files (x86)\hashcat-4.0.1>hashcat64 -a 3 -m [HashmodeNumber] -1
   abcdefghijklmnopqrstuvwxyz0123456789 ?1?1?1?1?1?1?1?1?1
   Pfad/zu/einer/Hashdatei
```

Wenn einem Angreifer die Länge eines anzugreifenden Passworts nicht bekannt ist, kann für eine Mask-Attack der „–increment“-Parameter an den Startbefehl angehangen werden. Dies hat zur Folge, dass bei einer definierten Maske mit einer Länge von acht Zeichen, `hashcat` (die Mask-Attack) mit einer einstelligen Maske beginnt und den Angriff mit einer achtstelligen Maske beendet (siehe Listing 2.1).

Listing 2.1: Beispiel des Inkrement-Parameters

```
?d
?d?d
?d?d?d
?d?d?d?d
?d?d?d?d?d
?d?d?d?d?d?d
?d?d?d?d?d?d?d
?d?d?d?d?d?d?d?d
```

Der Hintergrund dieser Funktion ist, dass `hashcat` ohne den Inkrement-Parameter immer nur nach Passwörtern sucht, die exakt der Länge der definierten Maske entsprechen. Wenn nun beispielsweise eine achtstellige Maske definiert wurde, würde ein Passwort was nur sieben Stellen hat, nicht gefunden werden. Wichtig hierbei ist, dass auch wenn der Inkrement-Parameter angegeben wurde, die Länge der definierten Maske immer noch den limitierenden Faktor angibt. So wird bei einer Maske mit

acht Stellen niemals nach einem Passwortkandidat mit mehr als acht Stellen gesucht werden. Abschließend ist zu sagen, dass die Mask-Attack eine intelligente Alternative zur in die Jahre gekommenen Brute-Force-Methode darstellt.

2.6.6 Hybrid-Attack

Die folgenden Informationen zu `hashcat's` „Hybrid-Attack“ basieren einerseits auf den Informationen der entsprechenden Webseite der `hashcat` Entwickler[24] und andererseits auf im Zuge dieser Arbeit gewonnen Erkenntnissen.
Der `hashcat`-eigene Hybrid-Angriff ähnelt sehr der Combinator-Attack. Allerdings werden hier nicht zwei Wörterbücher kombiniert, sondern zwei Angriffsarten. Der Hybrid-Angriff kombiniert einen Wörterbuchangriff mit einer Mask-Attack. So ist es beispielsweise möglich jedes Passwort eines Wörterbuchs mit jeder Zahl zwischen 0 und 9.999 zu kombinieren. Um einen Hybridangriff über `hashcat` zu starten, müssen die folgenden Parameter beim Start von `hashcat` übergeben werden:

- -a 6 / -a 7
  - Der Hybrid-Angriff unterscheidet sich von den anderen Angriffsmethoden in `hashcat` in sofern, dass es nicht einen Hybridangriffsmodus sondern zwei gibt. Die Idee hierbei ist es entscheiden zu können, was zuerst kommt, die Passwörter des Wörterbuchs oder die Passwörter der definierten Maske. Beim Aufruf der Funktion über „-a 6“ wird ein Wörterbuch mit einer Maske kombiniert, beim Aufruf der Funktion über „-a 7“ wird eine Maske mit einem Wörterbuch kombiniert.
- -m [HashmodeNumber]
  - Übergabe der Nummer des Ziel-Hashwerts (Anlage B.1)
- Pfad/zu/einer/Hashdatei
  - Festplattenpfads zur Hashdatei
- Pfad/zu/einer/Wörterbuchdatei
  - Festplattenpfad zu einer Wörterbuchdatei
- ?d?d?d?d
  - Definition einer Maske

Hier raus resultiert folgender Aufruf von `hashcat`:

```
C:\Program Files (x86)\hashcat-4.0.1>hashcat64 -a 6 -m [HashmodeNumber]
   Pfad/zu/einer/Hashdatei Pfad/zu/einer/Wörterbuchdatei ?d?d?d?d
```

[24] https://hashcat.net/wiki/doku.php?id=hybrid_attack, abgerufen am 09.05.2018 f.

2.6.7 Rule-based-Attack

Die Informationen zu `hashcat's` „Rule-based-Attack“ basieren einerseits auf den Informationen der entsprechenden Webseite der `hashcat` Entwickler[25] und andererseits auf im Zuge dieser Arbeit gewonnen Erkenntnissen.
Die „Rule-based-Attack“ ist weniger ein eigenständiger Angriff, als mehr eine Erweiterung des `hashcat`-eigenen Wörterbuchangriffs. Hierbei werden die Passwörter eines Wörterbuchs anhand von Regeln und Funktionen modifiziert, um so neue zusätzliche Passwortkandidaten zu erzeugen. Diese Art des Angriffs wird in `hashcat` als die komplexeste Angriffsart angesehen, die `hashcat`-Entwickler beschreiben sie auch als Programmiersprache, die für die Generierung von Passwortkandidaten entwickelt wurde. Durch ihre Funktionen zum Ändern, Kürzen oder Verlängern von Passwörtern beschreiben die `hashcat`-Entwickler sie als die flexibelste und effizienteste Methode zur Ermittlung von Passwörtern. Darüber hinaus sind die meisten der Funktionen zum Erstellen von Regeln kompatibel zu anderen Programmen beziehungsweise Tools wie „John the Ripper[26]“ oder „PasswordsPro[27]“ und umgekehrt. Die für einen solchen Angriff definierten Regeln müssen in einer separaten Datei, einer Regeldatei, gespeichert werden. Hierbei muss beachtet werden, dass `hashcat` diese Datei zeilenweise ausliest und alle Funktionen einer Zeile auf alle Passwörter des übergebenen Wörterbuchs anwendet. Dementsprechend kann jede Zeile der Regeldatei, mit ihren Funktionen, als eine Regel angesehen werden. Zu den Funktionen zum Modifizieren von Passwörtern gehört zum Beispiel:

- die Funktion „TN“, durch die ein Buchstabe innerhalb eines Passworts an einer definierten Stelle N in einen Groß- oder Kleinbuchstaben umgewandelt werden kann,
- die Funktion „c“, durch die alle Worte eines Wörterbuchs so modifiziert werden, dass jedes Wort mit einem großen Anfangsbuchstaben beginnt und der Rest des Wortes kleingeschrieben wird,
- oder auch die Funktion „$X“, durch die am Ende eines Passworts ein beliebiges Zeichen X angehangen wird.

Eine komplette Auflistung aller Funktionen und deren Beschreibung befindet sich im Anhang B.2 in den Abbildungen B.1, B.2 und B.3.
Der Aufruf eines regelbasierten Angriffs unterscheidet sich nur in einem Punkt vom Aufruf eines regulären Wörterbuchangriffs. Am Ende des in Kapitel 2.6.3 beschriebenen Aufrufs eines Wörterbuchangriffs muss ein zusätzlicher Parameter „-r /Pfad/-zu/einer/Regeldatei“ angehangen werden. Sodass der Aufruf des Wörterbuchangriffs den folgenden Aufbau aufweist:

```
C:\Program Files (x86)\hashcat-4.0.1>hashcat64 -a 0 -m [HashmodeNumber]
    Pfad/zu/einer/Hashdatei Pfad/zu/einer/Wörterbuchdatei -r
    /Pfad/zu/einer/Regeldatei
```

Listing 2.2 zeigt ein Beispiel einer solchen Regeldatei, bei der durch die Funktion „$X“ ein Zeichen an die Passwörter eines Wörterbuchs angehangen wird.

[25] https://hashcat.net/wiki/doku.php?id=rule_based_attack, abgerufen am 09.05.2018 f.
[26] http://www.openwall.com/john/, abgerufen am 10.05.2018
[27] http://www.insidepro.com/eng/passwordspro.shtml, abgerufen am 10.05.2018

Listing 2.2: Beispiel einer Regeldatei

```
$1$a
$1$b
$1$c
$2$a
$2$b
$2$c
$3$a
$3$b
$3$c
```

Die dargestellten Regeln zum Anhängen einer Zahlen-Buchstaben-Kombination werden nun auf das in Listing 2.3 dargestellte Passwort angewendet.

Listing 2.3: Wörterbuch mit nur einem Passwort

```
Passwort
```

Das Ergebnis sind die in Listing 2.4 dargestellten Passwortkandidaten.

Listing 2.4: Rule-based-Attack - Passwortkandidaten

```
Passwort1a
Passwort1b
Passwort1c
Passwort2a
Passwort2b
Passwort2c
Passwort3a
Passwort3b
Passwort3c
```

Neben selbsterstellter Regeln (in Regeldateien) bietet `hashcat` einem Nutzer über den Parameter „-g NUM“ die Möglichkeit zufallsbasierte Regeln, während eines Angriffs automatisiert zu erzeugen (wobei NUM die Menge der zu generierenden Regeln darstellt). Es sollte allerdings beachtet werden, dass die angegebene Anzahl an zu erzeugenden Regeln, die Anzahl der Regeln einer angegebenen Regeldatei, mit einschließt. Wenn also beispielsweise der Parameter „-g 300“ angegeben wird und eine Regeldatei übergeben wurde, die 200 Regeln enthält, dann erzeugt `hashcat` selbst nur 100 weitere zufällige Regeln, da die Gesamtanzahl von 300 Regeln somit erreicht wurde. Darüber hinaus ist die `hashcat`-Funktion „Saving Matched Rules“ (dt. abspeichern zutreffender Regeln), vor allem bei automatisch generierten Regeln, von Vorteil. Diese bietet die Möglichkeit anhand der Parameter „–debug-mode=1 –debug-file=matched.rule“ Regeln, die bei einem Angriff zu einem erfolgreich ermittelten Passwort geführt haben, in einer neuen Regeldatei zu speichern. Dies macht es möglich die Effizienz der verwendeten Regeldateien immer weiter zu verbessern.
Seit `hashcat` Version 0.07 ist es möglich beliebig oft den Parameter „-r /Pfad/zu/einer/Regeldatei“ an einen Wörterbuchangriff anzuhängen, um so mehrere verschiedene Regeldateien auf ein Wörterbuch anzuwenden. Hierbei wird jede Regel mit jeder angegebenen Regeldatei miteinander kombiniert. Der Aufruf eines solchen regelbasierten Angriffs könnte folgendermaßen aussehen:

```
C:\Program Files (x86)\hashcat-4.0.1>hashcat64 -r 123.rule -r abc.rule
    wordlist
```

`Hashcat's` regelbasierter Angriff bietet einem Nutzer extrem viel Flexibilität und eine Menge Werkzeuge um Passwörter auf jede nur noch so erdenkliche Art und Weise zu modifizieren. Hierdurch ist es einem erfahrenen Nutzer möglich komplett neue, eigene Angriffstechniken anhand von Regeln in `hashcat` zu realisieren. Allerdings können große Regeldateien mit Hunderten von Regeln auch schnell unübersichtig werden. Beispielsweise lässt sich über die Funktion „$1$2$3“ sehr einfach die Zahl 123 an die Passwörter eines Wörterbuchs anhängen. Wenn nun jedoch ein größerer Zahlenraum angehangen werden soll, wie zum Beispiel alle Zahlen zwischen 0 und 9.999, so müsste für jede der anzuhängenden Zahlen eine eigene Regel definiert werden. Dies stellt einerseits einen sehr großen Aufwand dar und andererseits macht es die Regeldatei unnötig komplex.

Kapitel 3

Angriffsszenarien und Optimierungen

Das Kapitel beschreibt die Konzeption und den Test von Passwortangriffsszenarien unter Einbeziehung des Faktors Mensch. Um eine repräsentative und vor allem überschaubare Übersicht über mögliche Angriffsszenarien für passwortgeschützte Systeme zu erhalten, wurde die Anzahl der Szenarien auf vier beschränkt. Jedes der Szenarien behandelt ein anderes Passwortschema, anhand dessen es in einem späteren Test überprüft wird. Zunächst wird auf den Aufbau der Szenarien eingegangen, währenddessen Bezug auf die allgemeinen Rahmenbedingungen, die allen Szenarien gleichermaßen zu Grunde liegen, genommen wird. Daraufhin folgt die Konzeption der einzelnen Szenarien, wobei jedes dieser Szenarien den zuvor vorgestellten Aufbaukriterien unterliegt. Nach der Konzeption werden für jedes Szenario zwei Tests mit der in Kapitel 2.6 vorgestellten Software `hashcat` durchgeführt. Im Anschluss an diese Tests werden innerhalb einer Analyse die Testergebnisse bewertet, auf mögliche Probleme des Szenarios beziehungsweise der verwendeten Software eingegangen und Lösungsansätze inklusive deren möglicher Umsetzung aufgezeigt.

3.1 Konzeption der Angriffsszenarien

Die folgenden Szenarien werden in einem späteren praktischen Test als Angriffsszenarien dienen, um auf Basis eines Hashwerts das dem Hashwert entsprechende Passwort zu ermitteln. Die Szenarien orientieren sich dabei zum Teil an bekannten Verfahren zur Passworterstellung, aber auch an Vorgaben offizieller Stellen wie dem Bundesamt für Sicherheit in der Informationstechnik (BSI). Es wird auf die Art und Weise eingegangen, wie Menschen Passwörter erstellen beziehungsweise entwickeln, um möglichst effizient das jeweilige Passwort anhand von `hashcat` erraten zu können. Jedes der Szenarien kann anhand eines Namens eindeutig identifiziert werden und entspricht dem folgenden Aufbau:

- Kurzbeschreibung
  - Das Szenario wird beschrieben und es wird der Untersuchungsgegenstand definiert. Weiterhin wird darauf eingegangen, inwiefern menschliches Verhalten bei dem Szenario eine Rolle spielt.
- Regelwerk
  - Das Regelwerk definiert nach welchem Schema innerhalb des Szenarios Passwörter gebildet und modifiziert werden können.
- Rahmenbedingungen
  - Zu den Rahmenbedingungen zählen alle Aspekte, die für das Szenario als gegeben betrachtet werden. Einige der Rahmenbedingungen gelten gleichermaßen für alle Szenarien, hierzu zählt zum Beispiel das Computersystem mit dem anhand von `hashcat` Passwörter erraten werden. Da die Zeit beziehungsweise die Geschwindigkeit, in der anhand von `hashcat` Passwörter ermittelt werden können, maßgeblich von der verwendeten Computerhardware abhängt und somit von System zu System stark variieren kann, möchten wir darauf hinweisen, dass alle Tests mit der folgenden Hardware durchgeführt werden:
    - Prozessor: AMD Phenom II X4 965 3,41 GHz
    - RAM: 8GB
    - Grafikkarte: NVIDIA GeForce GTX 560 Ti

    Weiterhin wird innerhalb der Tests ausschließlich die Grafikkarte zum Errechnen der Hashwerte genutzt. Jedes der konzipierten Angriffsszenarien wird sowohl mit dem Hashingverfahren SHA-512 (siehe Kapitel 2.4.1) als auch dem passwortbasierten Hashingverfahren scrypt (siehe Kapitel 2.4.2) getestet. Die Passwort-Hashwerte (SHA-512 und scrypt) der Szenarien können beispielsweise über eine Webseite, die Hashwerte erzeugt, generiert werden. Für die folgenden Tests wird die Webseite „www.browserling.com“ verwendet, welche ein Tool zum Erzeugen von SHA-512[1]- und scrypt[2]-Hashwerten anbietet. Die Webseite nutzt zum Erzeugen der SHA-512-Hashwerte die Javascript-Bibliothek `crypto-js`, welche von Jeff Mott 2009 entwickelt wurde[3] und zum Erzeugen der scrypt-Hashwerte die Javascript-Bibliothek `js-scrypt`, entwickelt 2013 von Tony Garnock-Jones[4]. Damit anhand der Hardware des Testsystems scrypt-Hashwerte in einer angemessenen Zeit ermittelt werden können, werden die entsprechenden Parameter (N,R,P) (siehe Kapitel 2.4.2) mit ihren Minimalwerten N=2, R=1 und P=1 bei einer Schlüssellänge von 32 Bit erzeugt.
- Entropie
  - In diesem Abschnitt wird auf den Grad der Entropie, die das Szenario erreicht beziehungsweise erreichen kann, eingegangen.

[1] https://www.browserling.com/tools/sha512-hash, abgerufen am 01.03.2018
[2] https://www.browserling.com/tools/scrypt, abgerufen am 01.03.2018
[3] https://code.google.com/archive/p/crypto-js/, abgerufen am 01.03.2018
[4] https://github.com/tonyg/js-scrypt, abgerufen am 01.03.2018

- Faktor Mensch
  - Da jedes der Szenarien sich explizit mit dem Faktor Mensch beschäftigt, wird in diesem Abschnitt darauf eingegangen, inwieweit sich der Faktor Mensch auf das jeweilige Szenario auswirkt.
- Durchführung
  - Hier wird der genaue Ablauf des Szenarios Schritt für Schritt bis hin zum Start von `hashcat` mit den entsprechenden Parametern beschrieben.
- Hypothese
  - Zum Schluss wird durch eine Hypothese das zu erwartende Ergebnis definiert.

3.1.1 Szenario 1 - Name + Zahl

Kurzbeschreibung

Im ersten Szenario wird auf ein Passwortschema eingegangen, welches aufgrund seines einfachen Aufbaus häufig verwendet wird (Florencio und Herley, 2006, S. 9). Dieses Szenario bildet ein laut Ur et al., 2015 weit verbreitetes Passwortschema ab, welches beschreibt, dass Menschen oft dazu tendieren beispielsweise ihnen bekannte Namen von Menschen oder Tieren, den Namen der Ehefrau, des eigenen Kindes oder auch des ersten Hundes als Passwort zu benutzen. Weiterhin gehen wir davon aus, dass ein solcher Name oft mit einer Zahl wie zum Beispiel einer vierstelligen Jahreszahl kombiniert wird. Diese Zahl hat eine direkte Verbindung zu dem vorangegangenen Namen oder der Person, von der das Passwort stammt. Beispiele hierfür sind das eigene Geburtsjahr oder auch eine Jahreszahl, die eine wichtige Bedeutung für die betreffende Person hat. Darüber hinaus nehmen wir an, dass neben dem Anhängen einer Jahreszahl Passwörter dieses Schemas häufig nur in der Art und Weise wie sie geschrieben werden, variieren. Daher werden in diesem Szenario die, wie wir vermuten, häufigsten beiden Varianten, wie Menschen Wörter innerhalb ihrer Passwörter schreiben, untersucht. Die erste Variante ist das Kleinschreiben aller Passwortbuchstaben, wohingegen die zweite Variante das Großschreiben des ersten und das Kleinschreiben aller übrigen Buchstaben des Passworts abbildet. Um diese Art von Passwort möglichst effizient und schnell mit `hashcat` zu ermitteln, wird ein sogenannter „Regelbasierter Angriff“ (siehe 2.6.7) durchgeführt.

Regelwerk

Für dieses Szenario wird eine Wörterbuchdatei benötigt, die eine möglichst große Anzahl an männlichen wie weiblichen Vornamen enthält und als Grundlage der zu erzeugenden Passwörter dient. Die Liste der Vornamen stammt hierbei von der Webseite www.albertmartin.de[5]. Die Webseite stellt eine freie Liste mit derzeit 19.030 deutschen, englischen und anderen internationalen Vornamen für statistische Auswertungen zur Verfügung. Nach dem Download der Liste kann sie unter dem Namen „names.dict“ gespeichert werden. Ferner muss für einen „Regelbasierten Angriff“ in

[5]http://www.albertmartin.de/vornamen/, abgerufen am 02.03.2018

`hashcat` eine Datei mit Regeln erstellt werden (siehe 2.6.7), durch die die Namen aus dem Wörterbuch modifiziert werden können, um Passwörter mit verschiedenen Schreibweisen zu bilden. Folgende Regeln müssen dafür definiert werden:

- Der „l“-Parameter, damit Hashwerte für alle Namen innerhalb des definierten Wörterbuchs mit Kleinbuchstaben errechnet werden.
- Der „c“-Parameter, damit Hashwerte für alle Namen innerhalb des definierten Wörterbuchs mit einem großen Anfangsbuchstaben und ansonsten kleinen Buchstaben errechnet werden.
- Um die Namen des Wörterbuchs mit Jahreszahlen von 1900 bis 2017 zu kombinieren, muss durch ein $-Zeichen vor jeder Zahl ($1 $9 $0 $0) für jedes Jahr eine Regel definiert werden. Diese Regeln zum Anhängen von Zahlen an einen Namen des Wörterbuchs werden mit dem „l“- und „c“-Parameter kombiniert.

Im Folgenden ein Auszug der ersten Zeilen der Regeldatei:

l
c
l$1$9$0$0
l$1$9$0$1
l$1$9$0$2
l$1$9$0$3
l$1$9$0$4
l$1$9$0$5

Die fertige Regeldatei wird unter dem Namen „szenario1.rule“ abgespeichert.

Rahmenbedingungen

Außer den für alle Szenarien geltenden Rahmenbedingungen (siehe 3.1) hat dieses Szenario keine weiteren Voraussetzungen.

Entropie

Wie in Kapitel 2.3.3 bereits beschrieben, geht es bei der Entropie eines Passworts um das Maß für die Zufälligkeit beziehungsweise den Informationsgehalt des Passworts. Durch die Szenariovorgaben des Regelwerks wird der Zeichenraum, also die Menge der verfügbaren Zeichen zur Bildung eines Passworts, definiert. Der Zeichenraum des Szenarios beinhaltet

- alle Großbuchstaben des Alphabets (26 Zeichen),
- alle Kleinbuchstaben des Alphabets (26 Zeichen),
- sowie die Zahlen von 0 bis 9 (10 Zeichen).

Somit ergibt sich für die Menge an verfügbaren beziehungsweise zulässigen Zeichen ein Zeichenraum von 62 Zeichen (N=62). Schwieriger ist die Bestimmung der Länge der möglichen Passwörter. Da die Länge des (für das Passwort) gewählten Namens nicht für jeden Namen gleich ist, muss mit einem Durchschnittswert gerechnet werden. Dementgegen hat die Jahreszahl, die mit dem Namen kombiniert wird, immer eine Länge von vier. Um die durchschnittliche Länge der Namen zu ermitteln, wurden die Namen des im Regelwerk beschriebenen Wörterbuchs in eine Liste der Microsoft Tabellenkalkulationssoftware Excel[6] kopiert. Hier wurde anhand der Funktion „LÄNGE()“ für jeden der Namen die Länge errechnet (die Funktion wurde auf alle Namen des Wörterbuchs angewendet). Im Anschluss wurde über die Funktion „MITTELWERT(B:B)“ (welche auf alle Werte der Spalte B angewendet wurde) der Durchschnittswert aller Namenslängen bestimmt. So ergibt sich eine durchschnittliche Länge der Namen in diesem Szenario von 6,3 Zeichen. Dieser Wert (abgerundet auf 6) addiert mit den vier Zahlen am Ende des Passworts, ergibt eine Passwortgesamtlänge von 10 Zeichen (L=10). Anhand der Werte für „L“ und „N“ kann nun anhand der Formel

$$H = L * \log(N)/\log(2) = 10 * \log(62)/\log(2)$$

der durchschnittliche Entropiewert der Passwörter berechnet werden. Hieraus ergibt sich ein Durchschnittswert von 60 Bit für die (anhand des erstellten Wörterbuchs) möglichen Passwörter.

Faktor Mensch

Im Abschnitt Entropie dieses Szenarios wurde deutlich, dass die Passwörter, die anhand dieses Szenarios gebildet werden, einen durchschnittlichen Entropiewert von 60 Bit aufweisen. Dies ist für sich genommen ein guter Wert, allerdings nur unter der Voraussetzung, dass das Passwort anhand eines Brute-Force-Angriffs (Kapitel 2.5.1) versucht wird zu ermitteln. Dieses Szenario hingegen bezieht durch die Annahme, dass Menschen dazu tendieren ihnen bekannte Namen und Daten für Passwörter zu verwenden, den Faktor Mensch (Kapitel 2.3.5) mit ein. Hierdurch ist es möglich einen Passwortangriff so anzupassen, dass das Ermitteln eines Passworts, welches wenn man es anhand eines Brute-Force-Angriffs angreifen würde, Stunden oder Tage benötigen würde, nur noch wenige Sekunden benötigt. Dies verdeutlicht, dass die Art und Weise wie ein Passwort gebildet wird, mindestens ebenso wichtig ist, wie die Stärke des Passworts auf Basis seines Entropiewerts.

Durchführung

Dem Szenario entsprechend muss ein Passwort bestehend aus einem männlichen oder weiblichen Vornamen kombiniert mit einer Jahreszahl zwischen 1900 und 2017 erstellt werden. Für das erstellte Passwort wird anschließend anhand des SHA-512- und des scrypt-Verfahrens der dem Passwort zugehörige Hashwert erzeugt und in der Datei „szenario1_sha512.hash“ beziehungsweise „szenario1_scrypt.hash“ abgespeichert. Nun wird auf dem für den Test vorgesehenen Computersystem auf der Kommandozeile `hashcat` gestartet. Beim Start von `hashcat` müssen diverse Parame-

[6] https://de.wikipedia.org/wiki/Microsoft_Excel, abgerufen am 10.03.2018

ter übergeben werden, um den SHA-512- beziehungsweise den scrypt-Hashwert von `hashcat` überprüfen zu lassen. Hierbei übergeben wir folgende Parameter:

- den Angriffsmodus „-a 0“, der für einen Wörterbuchangriff steht und bei `hashcat` auch als „Straight“-Modus bezeichnet wird (siehe 2.6.3)
- das anzugreifende Hashverfahren, anhand des Parameters „-m 1700“ für SHA-512 oder „-m 8900“ für scrypt
- die Datei, die den Hashwert des Passworts enthält (szenario1_sha512.hash oder szenario1_scrypt.hash)
- den Pfad zur Wörterbuchdatei (szenario1.dict)
- die Regeldatei mit den zuvor unter Regelwerk angesprochenen Regeln (szenario1.rule)

Zusammengefasst lautet die Eingabe zum Starten von `hashcat` in der Kommandozeile beispielhaft für den SHA-512-Hashwert wie folgt:

hashcat64 -a 0 -m 1700 szenario1_sha512.hash dictionary/szenario1.dict -r rules/szenario1.rule

Hypothese

`Hashcat` wird mit einem Wörterbuch, welches speziell auf das beschriebene Passwortschema ausgerichtet ist, und einer Regelsammlung, welche die Vornamen des Wörterbuchs entsprechend modifiziert, wesentlich schneller ein Passwort erraten können als mit einem allgemeinen Wörterbuch oder gar einem „Brute-Force-Angriff“ (siehe 2.5.1). Sofern der im Passwort verwendete Name im definierten Wörterbuch vorkommt, wird `hashcat` auf dem Testsystem innerhalb weniger Sekunden bis Minuten das gesuchte Passwort ermittelt haben.

3.1.2 Szenario 2 - bekannte, schwache Passwörter

Kurzbeschreibung

Dieses Szenario basiert auf einem direkten Wörterbuchangriff (siehe Kapitel 2.5.2) auf häufig verwendete, bekannt gewordene und schwache Passwörter. Beim Vergleich veröffentlichter Passwortlisten von Instituten und Sicherheitsfirmen, beziehungsweise im Internet öffentlich gewordenen Kundenpasswortlisten von Unternehmen, wird deutlich, dass viele Menschen immer noch schwache Passwörter wie „123456“ oder „Passwort“ verwenden. Diesen Umstand macht sich Angriffsszenario 2 zu Nutze. Hierbei wird ein Passwort einer im Jahr 2012 veröffentlichten Liste schwacher Passwörter ausgewählt. Es wird daraufhin versucht, das ausgewählte Passwort mit Hilfe eines zu diesem Zweck erstellten Wörterbuchs, welches aus Passwörtern aktuell veröffentlichter Passwortlisten besteht, zu ermitteln. Dabei werden Hashwerte für diese schwachen und immer noch häufig genutzten Passwörter errechnet, um möglichst schnell und effizient Passwörter dieser Art zu erraten.

Regelwerk

Das Passwort für das Szenario wird von einer im Jahr 2012 veröffentlichten Passwortliste ausgewählt. Das für das Szenario benötigte Wörterbuch wird anhand eines Abgleichs veröffentlichter Passwortlisten erstellt. Die Idee hierbei ist nicht ein möglichst großes und umfangreiches Wörterbuch anhand diverser Quellen zu erstellen, sondern eine Liste mit Fokus auf den weitverbreitetsten Passwörtern zu generieren. Hierzu werden die Passwörter verschiedener Top-100- oder auch Top-10-Listen miteinander abgeglichen, um so eine gemeinsame Liste der aktuell meist verwendeten Passwörter zu erhalten. Da beim Abgleich dieser Listen Duplikate ausgeschlossen werden, enthält die Ergebnisliste nur 167 Einträge. Die erste Quelle ist eine Liste der Top-10 der meist verwendeten deutschen Passwörter im Jahr 2017, welche durch das Hasso Plattner Institut (HPI)[7] veröffentlicht wurde. Die nächste Liste ist die durch das Sicherheitsunternehmen Splashdata[8] jährlich veröffentlichte Liste der Top-100 der schlechtesten Passwörter des Vorjahres (in diesem Fall handelt es sich um die Liste der schlechtesten Passwörter des Jahres 2017). Die letzte Liste ist nach einem Hackerangriff gegen das Unternehmen Adobe im Oktober 2013 (siehe Adobe Passwort Hack 2013[9]) öffentlich geworden. Sie enthält die Top-100, der durch den Hackerangriff bekannt gewordenen Passwörter. Die aus diesen drei Quellen resultierende Liste wird in der Datei „szenario2.dict“ abgespeichert. Da speziell diese häufig verwendeten Passwörter angegriffen werden, werden neben dem Wörterbuch keine Regeln zur Modifikation der Passwörter des Wörterbuchs benötigt.

Rahmenbedingungen

Außer den für alle Szenarien geltenden Rahmenbedingungen hat dieses Szenario keine weiteren Voraussetzungen.

Entropie

Da der Aufbau und die Variation der Passwörter dieses Szenarios stark voneinander abweichen, ist es nicht möglich eine einheitliche Aussage über den Entropiewert der Passwörter zu treffen. Es lässt sich lediglich mutmaßen, dass der Entropiewert nicht allzu groß sein kann, da Passwörter der veröffentlichten Passwortlisten teilweise nur aus Zahlen, Kleinbuchstaben oder der Kombination dieser bestehen. Dies schränkt den Zeichenraum der Passwörter erheblich ein, wodurch selbst ein 10-stelliges Passwort relativ schnell ermittelt werden kann.

Faktor Mensch

Dem Szenario geht die Annahme voraus, dass viele Menschen sich immer noch nur wenige Gedanken um die Stärke ihrer Passwörter machen und daher Passwörter benutzen, die nicht nur leicht in Erinnerung behalten werden können, sondern auch

[7] https://hpi.de/pressemitteilungen/2017/die-top-ten-deutscher-passwoerter.html, abgerufen am 11.03.2018

[8] https://13639-presscdn-0-80-pagely.netdna-ssl.com/wp-content/uploads/2017/12/Top-100-Worst-Passwords-of-2017a.pdf, abgerufen am 11.03.2018

[9] http://www.pc-magazin.de/news/adobe-hack-passwoerter-top-100-zu-simpel-1893649.html, abgerufen am 11.03.2018

ein hohes Maß an Komfortabilität bei der Eingabe aufweisen. Beispiele hierfür sind Passwörter wie „123456“ oder „abcdefg“, welche auf einfachen Buchstaben- oder Zahlenreihen basieren. Hinzu kommt, dass viele dieser sehr schwachen Passwörter immer wieder in den Medien und im Internet veröffentlicht werden, was deutlich macht, dass diese Passwörter, obwohl immer wieder vor ihrer Nutzung gewarnt wird, weiterhin von vielen Menschen benutzt werden. Diese „Sorglosigkeit“ können sich Angreifer zu Nutze machen, in dem sie die Passwortlisten, die veröffentlicht werden, um die Menschen vor der Nutzung schwacher Passwörter zu warnen, für wörterbuchbasierte Angriffe auf eben jene Passwörter verwenden.

Durchführung

Als erstes muss ein Passwort aus einer der unter „Regelwerk“ angegebenen Quellen ausgewählt werden. Für das ausgewählte Passwort muss ein SHA-512- und ein scrypt-Hashwert errechnet und jeweils in den Dateien szenario2_sha512.hash und szenario2_scrypt.hash gespeichert werden. Wie zuvor im Regelwerk beschrieben, wird für dieses Szenario keine `hashcat`-Regeldatei benötigt, da es sich um reinen Wörterbuchangriff ohne zusätzliche Modifikation der Passwörter durch `hashcat`-Regeln handelt. Als nächstes kann auf dem Testsystem `hashcat` über die Kommandozeile gestartet werden. Folgende Parameter werden beim Start übergeben:

- der Angriffsmodus „-a 0“, der für einen Wörterbuchangriff steht
- der Parameter „-m 1700“ für SHA-512 beziehungsweise den Parameter „-m 8900“ für scrypt
- der gespeicherte Hashwert des Passworts (szenario2_sha512.hash oder szenario2_scrypt.hash)
- der Pfad zur Wörterbuchdatei (szenario2.dict)

Daraus ergibt sich im Fall von script der folgende `hashcat` Startbefehl:

`hashcat`64 -a 0 -m 8900 szenario2_scrypt.hash dictionary/szenario2.dict

Hypothese

Bei einem gezielten Angriff auf bekannte Passwörter dieses Schemas wird `hashcat` das gesuchte Passwort innerhalb von Sekunden, wenn nicht sogar augenblicklich (ohne messbare Zeitspanne) erraten. Wir nehmen an, dass es immer noch möglich sein wird, Passwörter die bereits vor Jahren als schlecht definiert wurden, mit diesem Angriff (weitestgehend bestehend aus aktuell als schlecht definierten Passwörtern) zu erraten.

3.1.3 Szenario 3 - Passwortvorgaben des BSI

Kurzbeschreibung

Das Szenario beschäftigt sich mit den Passwortvorgaben offizieller Stellen, wie dem BSI, und wie Menschen unter der Berücksichtigung solcher Vorgaben Passwörter erzeugen. Speziell geht es um Passwörter die zwar den grundlegenden Vorgaben folgen, allerdings anhand bestimmter Schemata aufgebaut sind, welche durch einen Angreifer

identifiziert werden können. Solche Passwortschemata können anhand eines maskenbasierten Angriffs über `hashcat` (siehe 2.6.2) angegriffen werden. Dies werden wir versuchen in diesem Szenario beispielhaft durchzuführen.

Regelwerk

Die Passwörter dieses Szenarios unterliegen soweit möglich den Passwortvorgaben des BSI. Das BSI gibt folgende Hinweise[10] zum Bilden eines Passworts:

- Es sollte mindestens acht Zeichen lang sein, je länger desto besser.
- Es sollte aus Groß- und Kleinbuchstaben sowie Sonderzeichen und Ziffern (?!%...) bestehen.
- Tabu sind Namen von Familienmitgliedern, des Haustieres, des besten Freundes, des Lieblingsstars oder deren Geburtsdaten.
- Wenn möglich sollte es nicht in Wörterbüchern vorkommen.
- Es soll nicht aus gängigen Varianten und Wiederholungs- oder Tastaturmustern bestehen, also nicht asdfgh oder 1234abcd.
- Einfache Ziffern am Ende des Passworts anzuhängen oder eines der üblichen Sonderzeichen $! ? # am Anfang oder Ende eines ansonsten simplen Passwortes zu ergänzen ist nicht empfehlenswert.

Weiterhin muss das anzugreifende Passwort des Szenarios einem in `hashcat` abbildbaren Passwortschema entsprechen. Ur et al., 2015, S. 10, untersuchten zu diesem Thema das Passwortverhalten von 49 Versuchsteilnehmern. Sie erfuhren durch Befragungen, dass einige der Teilnehmer davon ausgingen, dass ein Passwort durch das Anhängen einer Kombination aus einer Zahl und einem Sonderzeichen am Ende des eigentlichen Passworts ein ausreichendes Maß an Sicherheit erhält. Darüber hinaus weisen Sie darauf hin, dass viele Menschen Passwörter häufig komplett oder mit vorhersehbaren Modifikationen wiederverwenden. Diesen Umstand wird sich dieses Szenario zu Nutze machen. Für das Passwortschema wird in diesem Szenario die Schneier- oder auch Akronym-Methode verwendet. Wie in Kapitel 2.2.1 bereits beschrieben, geht es bei dieser Methode darum ein Passwort anhand eines Schlüsselsatzes zu bilden, wobei die Anfangsbuchstaben eines jeden Wortes das eigentliche Passwort bilden. Um daraus ein angreifbares Passwortschema zu machen, muss das Passwort Konstanten haben, die bei jedem auf diese Art und Weise erzeugten Passwort wiedererkennbar sind. Daher definieren wir, dass das Passwort des Szenarios immer anhand desselben Schlüsselsatzes gebildet wird. Der Ersteller des Passworts ändert jedes Jahr nur die Zahl am Ende des Passworts beziehungsweise Schlüsselsatzes auf das aktuelle Jahr und hängt an diese Zahl ein „!“ an. Wir gehen in diesem Szenario davon aus, dass ein Angreifer den Schlüsselsatz nicht kennt, er jedoch das Passwortschema, welches den Aufbau des Passworts definiert, identifizieren konnte. Der Aufbau von Passwörtern des identifizierten Passwortschemas lautet hierbei wie folgt:

- eine Gesamtlänge von 8 Zeichen

[10] https://www.bsi-fuer-buerger.de/BSIFB/DE/Empfehlungen/Passwoerter/passwoerter_node.html, aufgerufen am 13.03.2018

- an jedes Passwort wird die aktuelle Jahreszahl im zweistelligen Format angehängt (Beispiel aus 2018 wird 18)
- das letzte Zeichen eines Passworts ist ein „!“
- alle übrigen Zeichen des Passworts sind immer gleich und bestehen nur aus Groß- und Kleinbuchstaben

Somit ergibt sich folgendes Schema: [Kombination aus Buchstaben][Jahreszahl][!]

Rahmenbedingungen

Außer den für alle Szenarien geltenden Rahmenbedingungen, hat dieses Szenario keine weiteren Rahmenbedingungen.

Entropie

Da dieses Szenario den Aufbau der Passwörter genau vorgibt und die Passwörter daher alle gleich aufgebaut sind, ist es möglich eine sehr genaue Aussage über die Entropiewerte der Passwörter dieses Szenarios zu treffen. Die Regeln des Szenarios geben genaue Angaben bezüglich der Länge und des Aufbaus der Passwörter. Passwörter des Szenarios haben

- eine Länge von 8 Zeichen
- und bestehen aus Klein- sowie Großbuchstaben
- und Sonderzeichen.

Hieraus ergibt sich folgende Berechnung für die Entropiwerte der Passwörter:

$$N = 94$$
$$L = 8$$
$$\text{Formel: } H = 8 * \log(94)/\log(2)$$

Somit hat jedes so erzeugte Passwort einen Entropiewert von 52 Bit.

Faktor Mensch

Wenn die Passwortvorgaben des BSI zu hundert Prozent erfüllt würden, wäre das Ergebnis eine vollkommen zusammenhangslose Kombination aus Zahlen, Zeichen und Sonderzeichen. Derartige Passwörter werden meistens mithilfe von Passwortgeneratoren erzeugt. Bei einem solchen Passwort hat der Faktor Mensch im Grunde keinen Einfluss auf die Generierung des Passworts, jedoch kann er beeinflussen auf welche Art und Weise mit dem Passwort umgegangen wird. Beispielsweise ist ein 16-stelliges Passwort aus wahllos kombinierten Zeichen, welches anhand eines Passwortgenerators erstellt wurde, als sehr sicher anzusehen. Es birgt allerdings das große Risiko, dass der Nutzer sich das Passwort nicht merken kann und es zum Beispiel auf einen Zettel schreibt, der im oder am Schreibtisch aufbewahrt wird. Das wiederum beeinhaltet das Risiko, dass das Passwort leicht anderen Menschen in die Hände fallen kann. In diesem Szenario wurden jedoch nicht alle Vorgaben des BSI eingehalten. Zwar wurde

Anhand der Akronym-Methode (2.2.1) eine scheinbare wahllose Kombination an Zeichen geschaffen, allerdings hat der Ersteller des Passworts durch die leicht abbildbare Weise wie Zahlen und Sonderzeichen mit dem Rest des Passworts kombiniert wurden, das Passwort angreifbar gemacht.

Durchführung

Als erstes muss ein Passwort gebildet werden, welches dem definierten Passwortschema entspricht. Daraufhin wird für das erstellte Passwort jeweils der SHA-512- und scrypt-Hashwert berechnet. Die Hashwerte werden anschließend jeweils in den Dateien szenario3_sha512.hash und szenario3_scrypt.hash abgespeichert. Im Anschluss werden folgende Parameter an `hashcat` beim Start übergeben:

- der Parameter „-m 1700“ für SHA-512 oder der Parameter „-m 8900“ für scrypt
- der Angriffsmodus „-a 3“, der für einen Passwortangriff mit der Mask-Methode steht
- den zu verwendenden charset (siehe 2.6.5) mit der Anzahl der Platzhalter, welcher die maximale Länge des Passworts festlegt (in diesem Szenario wird ein charset bestehend aus Klein- und Großbuchstaben benutzt: „-1 ?l?u“)
- der Pfad zur Datei mit dem gespeicherten Hashwert des Passworts (szenario3_sha512.hash oder szenario3_scrypt.hash)
- die Maske (siehe 2.6.5) die der Länge des Passworts entspricht, mit der aktuellen (zweistelligen) Jahreszahl und einem „!“als letztes Zeichen am Ende: „?1?1?1?1?118!“

Bei einem Angriff auf den SHA-512-Hashwert sieht die Eingabe dann folgendermaßen aus:

hashcat64 -m 1700 -a 3 -1 ?l?u szenario3_sha512.hash ?1?1?1?1?118!

Hypothese

Da dem Angreifer das Passwortschema des anzugreifenden Passworts bekannt ist und der Angreifer daher die Gesamtlänge des Passworts sowie die letzten drei Zeichen des Passworts kennt, muss der maskenbasierte Angriff nur noch alle Kombinationen an Groß- und Kleinbuchstaben der übrigen fünf Zeichen durchprobieren. Das Testsystem wird, abhängig vom gewählten Hashverfahren, das gesuchte Passwort innerhalb von wenigen Minuten bis Stunden ermitteln. Da es sich beim definierten Angriffsszenario um einen maskenbasierten Angriff mit `hashcat` handelt, wird das gesuchte Passwort definitiv gefunden werden.

3.1.4 Szenario 4 - Diceware

Kurzbeschreibung

Bei diesem Szenario untersuchen wir die Möglichkeit Passwörter beziehungsweise Passwortsätze, die mit der Diceware-Methode (siehe 2.2.5) erstellt wurden, möglichst

effizient zu erraten. Wie in Kapitel 2.2.5 bereits erklärt, geht es bei der Diceware-Methode darum, lange möglichst sichere Passwortsätze zu bilden, die leicht in Erinnerung behalten werden können. Für ein solches Passwortschema ist der Wörterbuchangriff eine gute Angriffsmethode, da das für diese Art von Angriff benötigte Wörterbuch von der Webseite des Erfinders der Diceware-Methode heruntergeladen werden kann. Normalerweise bleibt bei einem herkömmlichen Wörterbuchangriff auf ein Passwort immer das Risiko, dass das gesuchte Passwort nicht Teil des verwendeten Wörterbuchs ist. Bei der Diceware-Methode hingegen werden (wie in 2.2.5 bereits beschrieben) Passwortsätze anhand der (von der Webseite des Diceware-Erfinders beziehbaren) Diceware-Liste gebildet. Somit eignet sich dieselbe Liste bestens für einen wörterbuchbasierten Angriff. Das Problem hierbei ist, dass es sich bei Diceware-Passwörtern in der Regel nicht nur um ein einziges Wort, sondern um mehrere Wörter handelt, wodurch ein herkömmlicher Wörterbuchangriff nicht funktioniert, da auf diese Art und Weise der Hashwert eines einzelnen Wortes mit dem Hashwert eines Passwortsatzes (zwei oder mehr Wörter) verglichen werden würde. `Hashcat` bietet hierfür über einen sogenannten Kombinationsangriff die Möglichkeit einen Wörterbuchangriff anhand mehrerer Wörterbücher durchzuführen. Hierbei werden die Wörter aus zwei Wörterbüchern miteinander kombiniert. Diese Methode werden wir in diesem Szenario anwenden.

Regelwerk

Diceware-Passwörter werden (wie in 2.2.5 beschrieben) mit Hilfe einer sogenannten Diceware-Liste und einem sechsseitigen Würfel gebildet. Die Diceware-Liste enthält 7.776 Wörter, jedem der Wörter ist eine fünfstellige Zahl zugeordnet. Um einen Diceware-Passwortsatz (engl. „diceware passphrase“) zu bilden, muss für jedes Wort des Passwortsatzes fünfmal mit dem Würfel gewürfelt werden. Die daraus resultierende fünfstellige Zahl kann dann dem entsprechenden Wort der Diceware-Liste zugeordnet werden.

Rahmenbedingungen

Außer den für alle Szenarien geltenden Rahmenbedingungen, wird weiterhin ein sechseitiger Würfel sowie die deutsche Version der Diceware-Wörterliste benötigt (diese kann von der Webseite des Diceware Entwicklers Arnold G. Reinhold[11] heruntergeladen werden). Wir gehen in diesem Szenario davon aus, dass dem Angreifer das verwendete Passwortschema und somit die Verwendung von Diceware zur Erstellung des Passworts, sowie dass die Anzahl der verwendeten Diceware-Wörter des Passwortsatzes zwei ist, bekannt ist. Weiterhin weiß der Angreifer, dass weder Leerzeichen noch Sonderzeichen bei der Erstellung des Passwortsatzes verwendet wurden.

Entropie

Laut der Webseite[12] des Entwicklers der Diceware-Methode hat jedes Wort einer Diceware-Liste genau 12,9 Bit Entropie. Die Diceware-Liste enthält 7.776 Wörter. Das heißt, dass ein Diceware-Passwort bestehend aus zwei Wörtern eine Entropie

[11] http://world.std.com/~reinhold/diceware_german.txt,abgerufen am 20.03.2018
[12] http://world.std.com/~reinhold/diceware.html, abgerufen am 20.03.2018

von 25,8 aufweist. Dies lässt sich anhand der Formel zur Berechnung der Entropie eines Passworts nachweisen:

$$N = 7776$$
$$L = 2$$
$$\text{Formel: } H = 2 * \log(7776)/\log(2) = 25,8$$

Wie hier zu sehen, hat ein Passwortsatz mit zwei Wörtern, bei einem Zeichenraum von 7.776 Elementen, einen Entropiewert von 25,8 Bit.

Faktor Mensch

Sofern die Diceware-Methode anhand eines sechsseitigen Würfels benutzt wird, ist der Einfluss, den ein Mensch auf die Erstellung des Passworts nehmen kann, als indirekt anzusehen. Indirekt insofern, dass ein Mensch das Würfelergebnis zwar nicht direkt bewusst beeinflussen kann, jedoch seine Wurftechnik eine direkte Auswirkung auf das aus dem Wurf resultierende Ergebnis hat. Allerdings hat der Faktor Mensch einen direkten Einfluss darauf, in welcher Art und Weise mit der entstandenen Diceware-Passphrase umgegangen wird. Wenn beispielsweise ein Nutzer eine Passphrase aufschreibt, statt sie sich zu merken, besteht die Möglichkeit, dass andere Personen unberechtigten Zugriff auf die Passphrase und die durch die Passphrase geschützten Daten erlangen.

Durchführung

Als erstes muss anhand der Diceware-Liste und dem Würfel das Diceware-Passwort ermittelt werden. Hierzu muss für jedes der Wörter fünfmal gewürfelt werden um eine fünfstellige Zahl zu ermitteln, die im Anschluss in der Diceware-Liste nachgeschlagen werden kann. Die resultierenden Wörter können auf diverse Arten miteinander kombiniert werden um einen Passwortsatz zu bilden. Nachfolgend möchten wir drei der möglichen Kombinationen aufzeigen:

1. Wort1[Leerzeichen]Wort2
2. Wort1Wort2
3. Wort1[Sonderzeichen]Wort2 (Beispiel: Wort1-Wort2, Wort1@Wort2)

Zum Kombinieren der Wörter entscheiden wir uns für das zweite Beispiel. Wenn beide Wörter kombiniert wurden, muss der dem Passwortsatz zugehörige Hashwert für SHA-512 und scrypt errechnet werden. Die Hashwerte werden dafür jeweils in den Dateien szenario4_sha512.hash und szenario4_scrypt.hash abgespeichert. Um einen kombinationsbasierten Angriff mit hashcat durchzuführen, werden zwei Wörterbücher benötigt. Da es sich bei Diceware-Passwortsätzen um Wörter aus demselben Wörterbuch (die Diceware-Liste) handelt, muss von der heruntergeladenen Diceware-Liste eine Kopie erstellt werden. Nun müssen beim Start von `hashcat` folgende Parameter übergeben werden:

- der Parameter „-m 1700“ für SHA-512 oder der Parameter „-m 8900“ für scrypt

- der Angriffsmodus „-a 1“, für einen Passwortangriff mit der Kombinationsmethode
- der gespeicherte Hashwert des Passworts (szenario4_sha512.hash oder szenario4_scrypt.hash)
- der Pfad zum ersten Wörterbuch (Diceware Liste 1)
- der Pfad zum zweiten Wörterbuch (Diceware Liste 2)

Bei einem Angriff auf den SHA-512-Hashwert lautet der Aufruf von `hashcat` dann wie folgt:

`hashcat`64 -m 1700 -a 1 szenario4_sha512.hash dictionaries/diceware_de1.dict dictionaries/diceware_de2.dict

Hypothese

Ein Angreifer wird mit dieser Vorgehensweise definitiv das gesuchte Passwort finden. Im Gegensatz zu einem maskenbasierten Angriff wird durch diese Methodik wesentlich schneller und effizienter das gesuchte Passwort gefunden werden, da (sofern das gesuchte Passwort Teil des Wörterbuchs ist, was bei einem Angriff auf einen Diceware-Passwortsatz eine gegebene Voraussetzung ist) bei einem Wörterbuchangriff immer ganze Wörter und nicht wie bei einem maskenbasierten Angriff Kombinationen von Zeichen getestet werden.

3.2 Test der Szenarien, Analyse der Ergebnisse

Im Folgenden werden die zuvor beschriebenen Angriffsszenarien getestet und die daraus resultierenden Ergebnisse analysiert. Während des Tests wird die dem Szenario zuzuordnende Programmausgabe dokumentiert und anschließend in Hinblick auf die zuvor definierte Hypothese überprüft. Weiterhin werden alle beim Durchführen des Szenarios entstanden Probleme oder unnötigen Aufwände festgehalten und bewertet.

3.2.1 Szenario 1 - Name + Zahl

Test

Den Vorgaben des Szenarios entsprechend wurde die Vornamensliste heruntergeladen und als Wörterbuch unter dem Namen „szenario1.dict“ abgespeichert. Darüber hinaus wurde eine Regeldatei erstellt, um die Vornamen des Wörterbuchs entsprechend den Vorgaben des Szenarios zu modifizieren. Anschließend wurde ein zufällig gewählter Vorname mit einer Jahreszahl zwischen 1900 und 2017 kombiniert und der daraus resultierende Hashwert (sowohl als SHA-512- als auch als scrypt-Hashwert) zusammen mit den übrigen vorgegebenen Parametern des Szenarios (der Wörterbuchdatei mit Vornamen, sowie der Regeldatei mit Parametern zur Passwortmodifikation) beim Start der Software `hashcat` übergeben. Darauf hin wurde (als der gesuchte Hashwert gefunden wurde) folgende Ausgabe erzeugt:

Ausgabe des Tests des SHA-512-Hashwerts:

Listing 3.1: Szenario 1 - Test 1

```
C:\Program Files (x86)\hashcat-4.0.1>hashcat64 -a 0 -m 1700
    szenario1_sha512.hash dictionaries\szenario1.dict -r
    rules\szenario1.rule
hashcat (v4.0.1) starting...

9026131eacfa760c0134e3e1af501c28ca08130a2597981...01f6a5c179524:Kevin1994

Session..........: hashcat
Status...........: Cracked
Hash.Type........: SHA-512
Hash.Target......:
    9026131eacfa760c0134e3e1af501c28ca08130a2597981ed36...179524
Time.Started.....: Wed Mar 14 18:16:59 2018 (0 secs)
Time.Estimated...: Wed Mar 14 18:16:59 2018 (0 secs)
Guess.Base.......: File (dictionaries\szenario1.dict)
Guess.Mod........: Rules (rules\szenario1.rule)
Guess.Queue......: 1/1 (100.00%)
Speed.Dev.#1.....: 29550.6 kH/s (10.40ms)
Recovered........: 1/1 (100.00%) Digests, 1/1 (100.00%) Salts
Progress.........: 3883008/4529140 (85.73%)
Rejected.........: 0/3883008 (0.00%)
Restore.Point....: 8192/19030 (43.05%)
Candidates.#1....: Isebella1957 -> Tanhya2015
HWMon.Dev.#1.....: Temp: 33c Fan: 40%

Started: Wed Mar 14 18:16:55 2018
Stopped: Wed Mar 14 18:17:00 2018
```

Ausgabe des Tests des scrypt-Hashwerts:

Listing 3.2: Szenario 1 - Test 2

```
c:\Program Files (x86)\hashcat-4.0.1>hashcat64 -a 0 -m 8900
    szenario1_scrypt.hash dictionary/szenario1.dict -r rules/szenario1.rule
hashcat (v4.0.1) starting...

SCRYPT:1024:1:1:MTIzNDU2:hANNhySzs/TfyI7O7aVlObfSv1J/LfNi5GbTG4xPtIg=:Kevin1994

Session..........: hashcat
Status...........: Cracked
Hash.Type........: scrypt
Hash.Target......:
    SCRYPT:2:1:1:MTIzNDU2:hANNhySzs/TfyI7O7aVlObfSv1...xPtIg=
Time.Started.....: Sun Mar 11 13:11:55 2018 (58 secs)
Time.Estimated...: Sun Mar 11 13:12:53 2018 (0 secs)
Guess.Base.......: File (dictionaries\szenario1.dict)
```

```
13 Guess.Mod........: Rules (rules\szenario1.rule)
14 Guess.Queue......: 1/1 (100.00%)
15 Speed.Dev.#1.....:  40976 H/s (46.76ms)
16 Recovered........: 1/1 (100.00%) Digests, 1/1 (100.00%) Salts
17 Progress.........: 2390016/4529140 (52.77%)
18 Rejected.........: 0/2390016 (0.00%)
19 Restore.Point....: 8192/19030 (43.05%)
20 Candidates.#1....: Isebella1994 -> Laure1994
21 HWMon.Dev.#1.....: Temp: 70c Fan: 50%
22
23 Started: Sun Mar 11 13:11:45 2018
24 Stopped: Sun Mar 11 13:12:54 2018
```

Analyse

Das Ergebnis der Tests bestätigt die vorangegangene Hypothese in sofern, dass `hashcat` das gesuchte Passwort (Zeile 4) „Kevin1994“ anhand des vordefinierten Passwortschemas innerhalb weniger Sekunden (58 Sekunden - scrypt-Testergebnis) beziehungsweise unmittelbar (ohne messbare Zeitspanne - SHA-512-Testergebnis) erraten hat. Es wurde jedoch deutlich, dass es sehr aufwendig sein kann eine Regeldatei für `hashcat` zu erstellen. Um regelbasiert Zahlen an ein Passwort zu hängen, ist es erforderlich, für jede der angehängten Zahlen eine eigene Regel zu definieren. Somit war es auch in diesem Szenario nötig für jede Jahreszahl zwischen 1900 und 2017 eine eigene Regel zu formulieren. Im Fall dieses Szenarios war der Aufwand noch überschaubar, da die Zahlen auf einen bestimmten Jahreszeitraum begrenzt waren. Wenn dies nicht der Fall gewesen wäre, sondern beispielsweise alle vierstelligen Zahlen relevant gewesen wären um möglichst alle Kombinationen abzudecken, so hätte sich der Aufwand zur Erstellung aller Regeln um einiges erhöht. Weiterhin fiel auf, dass durch all diese Regeln die Komplexität der Regeldatei um einiges zugenommen hat, wodurch sie weniger überschaubar wurde. Zwar haben Recherchen gezeigt, dass es diverse Software-Tools[13] im Internet gibt um automatisiert Regeln für `hashcat` zu erstellen, allerdings würde die Benutzung lediglich das Problem der Erstellung der Regeln lösen, jedoch nicht die zunehmende Komplexität einer Regeldatei verhindern.

3.2.2 Szenario 2 - bekannte Passwörter

Test

Entsprechend den Regeln und Vorgaben des Szenarios wurden die Passwortlisten der drei im Szenario angegebenen Quellen zu einer Liste zusammengefasst und in der Datei „szenario2.dict“ gespeichert. Das Passwort wurde aus der Liste der 2012 gestohlenen LinkedIn-Passwörter (veröffentlicht auf der Webseite „fortune.com“[14]) ausgewählt. Hierbei war es die Idee zu überprüfen, ob ein im Jahr 2012 als schlecht definiertes Passwort mit einem Wörterbuch aus aktuell als schlecht definierten Passwörtern

[13] https://github.com/sc0tfree/mentalist, abgerufen am 27.03.2018

[14] http://fortune.com/2016/05/18/linkedin-breach-passwords-most-common/, abgerufen am 27.03.2018

immer noch erraten werden kann. Dies würde bedeuten, dass die im Szenario beschriebene Hypothese zutreffend wäre. Für das gewählte Passwort wurde ein SHA-512- und scrypt-Hashwert errechnet. Die errechneten Hashwerte wurden, dem Szenario entsprechend, in den Dateien „szenario2_sha512.hash“ „szenario2_scrypt.hash“ gespeichert. Im Anschluss wurde für jeden Test eine Hashdatei sowie das erzeugte Wörterbuch an `hashcat` als Startparameter übergeben. Nachfolgend die Ausgabe von `hashcat`, nachdem das zu den Hashwerten gehörige Passwort erraten wurde:

Ausgabe des Tests des SHA-512-Hashwerts:

Listing 3.3: Szenario 2 - Test 1

```
C:\Program Files (x86)\hashcat-4.0.1>hashcat64 -a 0 -m 1700
    tests/szenario2_sha512.hash tests/dictionaries/szenario2.dict
hashcat (v4.0.1) starting...

ba3253876aed6bc22d4a6ff53d8406c6ad864195ed144ab5c87...a0bab413:123456

Session..........: hashcat
Status...........: Cracked
Hash.Type........: SHA-512
Hash.Target......:
    ba3253876aed6bc22d4a6ff53d8406c6ad864195ed144ab5c87...bab413
Time.Started.....: Sat Mar 17 01:38:20 2018 (0 secs)
Time.Estimated...: Sat Mar 17 01:38:20 2018 (0 secs)
Guess.Base.......: File (tests/dictionaries/szenario2.dict)
Guess.Queue......: 1/1 (100.00%)
Speed.Dev.#1.....: 455.3 kH/s (0.12ms)
Recovered........: 1/1 (100.00%) Digests, 1/1 (100.00%) Salts
Progress.........: 167/167 (100.00%)
Rejected.........: 0/167 (0.00%)
Restore.Point....: 0/167 (0.00%)
Candidates.#1....: 123456 -> thunder
HWMon.Dev.#1.....: Temp: 33c Fan: 40%

Started: Sat Mar 17 01:38:06 2018
Stopped: Sat Mar 17 01:38:21 2018
```

Ausgabe des Tests des scrypt-Hashwerts:

Listing 3.4: Szenario 2 - Test 2

```
C:\Program Files (x86)\hashcat-4.0.1>hashcat64 -a 0 -m 8900
    tests/szenario2_scrypt.hash tests/dictionaries/szenario2.dict
hashcat (v4.0.1) starting...

SCRYPT:2:1:1:MTIzNDU2:7ICqt9XQ/QySifPvO+j2WCchxN9Er1GZP2yfgIRB4CA=:123456

Session..........: hashcat
Status...........: Cracked
```

```
8  Hash.Type........: scrypt
9  Hash.Target......:
       SCRYPT:2:1:1:MTIzNDU2:7ICqt9XQ/QySifPvO+j2WCchxN9Er...RB4CA=
10 Time.Started.....: Sat Mar 17 01:46:41 2018 (0 secs)
11 Time.Estimated...: Sat Mar 17 01:46:41 2018 (0 secs)
12 Guess.Base.......: File (tests/dictionaries/szenario2.dict)
13 Guess.Queue......: 1/1 (100.00%)
14 Speed.Dev.#1.....:  62800 H/s (0.10ms)
15 Recovered........: 1/1 (100.00%) Digests, 1/1 (100.00%) Salts
16 Progress.........: 167/167 (100.00%)
17 Rejected.........: 0/167 (0.00%)
18 Restore.Point....: 0/167 (0.00%)
19 Candidates.#1....: 123456 -> thunder
20 HWMon.Dev.#1.....: Temp: 33c Fan: 40%
21
22 Started: Sat Mar 17 01:46:31 2018
23 Stopped: Sat Mar 17 01:46:43 2018
```

Analyse

Beide Tests bestätigen die Hypothese des Szenarios. Wie zuvor angenommen konnte das gesuchte Passwort „123456“ (Zeile 4) in beiden Tests innerhalb von Millisekunden erraten werden. Durch das erfolgreiche Erraten des richtigen Passworts wurde auch der zweite Teil der Hypothese bestätigt. Dabei ging es darum, zu testen, ob Passwörter, die vor Jahren (2012) als schlecht definiert und veröffentlicht wurden, heute immer noch verwendet werden. Dieses Ergebnis macht sehr deutlich, wie sorglos und einfach es sich viele Menschen bei der Auswahl ihres Passworts machen.

3.2.3 Szenario 3 - Passwortvorgaben des BSI

Test

In diesem Szenario wurde mithilfe des definierten Passwortschemas ein Passwort anhand des Schlüsselsatzes „I will reach my goals 2018!“ gebildet, woraus das Passwort „Iwrmg18!“ resultierte. Hierfür konnte nachfolgend der entsprechende SHA-512- beziehungsweise scrypt-Hashwert erzeugt und unter den im Szenario definierten Dateinamen abgespeichert werden. Zum Schluss wurde `hashcat`, die im Szenario definierten Parameter, übergeben. Nachfolgend die Ergebnisse des Tests mit dem SHA-512-Hashwert, sowie des Tests mit dem scrypt-Hashwert.

Ausgabe des Tests des SHA-512-Hashwerts:

Listing 3.5: Szenario 3 - Test 1

```
1 C:\Program Files (x86)\hashcat-4.0.1>hashcat64 -a 3 -m 1700 -1 ?l?u
      tests\szenario3_sha512.hash ?1?1?1?1?118!
2 hashcat (v4.0.1) starting...
3
4 4d48ea51151506faa8aa14957387581f30e99454c509e1bd257306...df4909f290ec:Iwrmg18!
```

```
5
6  Session..........: hashcat
7  Status...........: Cracked
8  Hash.Type........: SHA-512
9  Hash.Target......:
       4d48ea51151506faa8aa14957387581f30e99454c509e1bd257...f290ec
10 Time.Started.....: Mon Mar 19 23:23:11 2018 (4 secs)
11 Time.Estimated...: Mon Mar 19 23:23:15 2018 (0 secs)
12 Guess.Mask.......: ?1?1?1?1?118! [8]
13 Guess.Charset....: -1 ?l?u, -2 Undefined, -3 Undefined, -4 Undefined
14 Guess.Queue......: 1/1 (100.00%)
15 Speed.Dev.#1.....: 28735.5 kH/s (10.35ms)
16 Recovered........: 1/1 (100.00%) Digests, 1/1 (100.00%) Salts
17 Progress.........: 113178624/380204032 (29.77%)
18 Rejected.........: 0/113178624 (0.00%)
19 Restore.Point....: 0/2704 (0.00%)
20 Candidates.#1....: SNJer18! -> YDBQg18!
21 HWMon.Dev.#1.....: Temp: 35c Fan: 40%
22
23 Started: Mon Mar 19 23:23:03 2018
24 Stopped: Mon Mar 19 23:23:16 2018
```

Ausgabe des Tests des scrypt-Hashwerts:

Listing 3.6: Szenario 3 - Test 2

```
1  C:\Program Files (x86)\hashcat-4.0.1>hashcat64 -a 3 -m 8900 -1 ?l?u
       tests\szenario3_3_scrypt.hash ?1?1?1?1?118!
2  hashcat (v4.0.1) starting...
3
4  SCRYPT:1024:1:1:MTIzNDU2:j/rMYvFKsgjvSWrXc6WR7oMm2f+C1awA9dxJaHzHs88=:Iwrmg18!
5
6  Session..........: hashcat
7  Status...........: Cracked
8  Hash.Type........: scrypt
9  Hash.Target......:
       SCRYPT:2:1:1:MTIzNDU2:j/rMYvFKsgjvSWrXc6WR7oMm2f...zHs88=
10 Time.Started.....: Sun Mar 18 21:11:51 2018 (1 hour, 11 mins)
11 Time.Estimated...: Sun Mar 18 22:23:31 2018 (0 secs)
12 Guess.Mask.......: ?1?1?1?1?118! [8]
13 Guess.Charset....: -1 ?l?u, -2 Undefined, -3 Undefined, -4 Undefined
14 Guess.Queue......: 1/1 (100.00%)
15 Speed.Dev.#1.....:  32772 H/s (46.75ms)
16 Recovered........: 1/1 (100.00%) Digests, 1/1 (100.00%) Salts
17 Progress.........: 140875776/380204032 (37.05%)
18 Rejected.........: 0/140875776 (0.00%)
19 Restore.Point....: 2707456/7311616 (37.03%)
20 Candidates.#1....: Ivnjg18! -> IWAKJ18!
21 HWMon.Dev.#1.....: Temp: 74c Fan: 40%
22
23 Started: Sun Mar 18 21:11:41 2018
```

```
24 Stopped: Sun Mar 18 22:23:32 2018
```

Analyse

Wie durch die Hypothese des Szenarios angenommen, konnte das Passwort „Iwrmg18!“ (Zeile 4) bei beiden Tests gefunden werden. Weiterhin ist ein klarer zeitlicher Unterschied beim Ermitteln der Passwörter zwischen dem Test des SHA-512-Hashwerts (4 Sekunden) und dem Test des scrypt-Hashwerts (eine Stunde und elf Minuten) festzustellen. Dies macht die Überlegenheit des scrypt-Verfahrens gegenüber dem SHA-512-Verfahren allzu deutlich. Deweiteren hat der Test verdeutlicht, wie wichtig Informationen über das anzugreifende Passwortschema sind. Zwar würde ein maskenbasierter Angriff mit `hashcat` immer irgendwann das gesuchte Passwort ermitteln, allerdings wächst die Zeit, die `hashcat` benötigt um das gesucht Passwort zu finden, exponentiell an umso weniger Informationen dem Angreifer über das anzugreifende Passwortschema bekannt sind. Der Vorteil des maskenbasierten Angriffs ist seine Anpassbarkeit an diverse Passwortschemata. Je mehr der Angriff an das anzugreifende Passwort angepasst wird, um so geringer fällt die Zeit zum Ermitteln des gesuchten Passworts aus.

3.2.4 Szenario 4 - Diceware

Test

Zu Beginn des Tests wurde die deutsche Version der Diceware-Liste von der Diceware-Entwickler-Webseite heruntergeladen. Da wir für den Kombinationsangriff mit `hashcat` zwei Wörterbücher benötigen, wurde von der heruntergeladenen Diceware-Liste eine Kopie erstellt. Als nächstes wurde anhand der Beschreibung des Szenarios ein Diceware-Passwortsatz gebildet. Für diesen Passwortsatz wurden sowohl der SHA-512- als auch der scrypt-Hashwert ermittelt und unter den durch das Szenario vorgegebenen Dateinamen gespeichert. Anschließend wurden `hashcat` alle im Szenario definierten Parameter beim Start übergeben. Unter Berücksichtigung der gesuchten Hashwerte (SHA-512 und scrypt) wurde folgende Ausgabe erzeugt:

Ausgabe des Tests des SHA-512-Hashwerts:

Listing 3.7: Szenario 4 - Test 1

```
1 C:\Program Files (x86)\hashcat-4.0.1>hashcat64 -m 1700 -a 1
     tests/szenario4_sha512.hash tests/dictionaries/diceware_de.dict
     tests/dictionaries/diceware_de2.dict
2 hashcat (v4.0.1) starting...
3
4 adce218c7a6fca0061275193d3c168191b5fe9c063e06b501f741....373a062:fbfalte
5
6 Session..........: hashcat
7 Status...........: Cracked
8 Hash.Type........: SHA-512
```

```
Hash.Target......:
    adce218c7a6fca0061275193d3c168191b5fe9c063e06b501f7...73a062
Time.Started.....: Sat Mar 17 02:39:30 2018 (1 sec)
Time.Estimated...: Sat Mar 17 02:39:31 2018 (0 secs)
Guess.Base.......: File (tests/dictionaries/diceware_de.dict), Left Side
Guess.Mod........: File (tests/dictionaries/diceware_de2.dict), Right Side
Speed.Dev.#1.....: 17472.3 kH/s (21.19ms)
Recovered........: 1/1 (100.00%) Digests, 1/1 (100.00%) Salts
Progress.........: 14929920/60466176 (24.69%)
Rejected.........: 0/14929920 (0.00%)
Restore.Point....: 0/7776 (0.00%)
Candidates.#1....: 0ez -> zzzzfase
HWMon.Dev.#1.....: Temp: 34c Fan: 40%

Started: Sat Mar 17 02:39:26 2018
Stopped: Sat Mar 17 02:39:31 2018
```

Ausgabe des Tests des scrypt-Hashwerts:

Listing 3.8: Szenario 4 - Test 2

```
C:\Program Files (x86)\hashcat-4.0.1>hashcat64 -m 8900 -a 1
    tests/szenario4_scrypt.hash tests/dictionaries/diceware_de.dict
    tests/dictionaries/diceware_de2.dict
hashcat (v4.0.1) starting...

SCRYPT:2:1:1:MTIzNDU2:1WFIHqhLYflqBCo8d3lP5wVdWhrL7Pagupq7+vvmiGA=:fbfalte

Session..........: hashcat
Status...........: Cracked
Hash.Type........: scrypt
Hash.Target......:
    SCRYPT:2:1:1:MTIzNDU2:1WFIHqhLYflqBCo8d3lP5wVdWhrL7...vmiGA=
Time.Started.....: Sat Mar 17 02:40:24 2018 (3 secs)
Time.Estimated...: Sat Mar 17 02:40:27 2018 (0 secs)
Guess.Base.......: File (tests/dictionaries/diceware_de.dict), Left Side
Guess.Mod........: File (tests/dictionaries/diceware_de2.dict), Right Side
Speed.Dev.#1.....: 1483.1 kH/s (0.14ms)
Recovered........: 1/1 (100.00%) Digests, 1/1 (100.00%) Salts
Progress.........: 3883008/60466176 (6.42%)
Rejected.........: 0/3883008 (0.00%)
Restore.Point....: 0/7776 (0.00%)
Candidates.#1....: 0falte -> finitefalte
HWMon.Dev.#1.....: Temp: 36c Fan: 40%

Started: Sat Mar 17 02:40:10 2018
Stopped: Sat Mar 17 02:40:28 2018
```

Analyse

`Hashcat` konnte, wie in der Hypothese des Szenarios bereits vermutet, den Passwortsatz den Hashwerten zuordnen und das gesuchte Passwort „fbfalte“ sehr schnell (1-3 Sekunden) erraten. Allerdings konnten wir feststellen, dass der `hashcat`-eigene Kombinationsangriff nur Passwörter von maximal zwei Wörterbüchern miteinander kombiniert. Das naheliegende Problem hierbei ist, dass Passwortsätze, in unserem Fall ein Diceware-Passwortsatz, in den seltensten Fällen aus nur zwei Wörtern bestehen. Bei einem kleinen Wörterbuch (wie dem von Diceware) können Zweiwort-Passwortsätze innerhalb weniger Sekunden erraten werden. Für einen Passwortsatz, der aus mehr als zwei Wörtern besteht, bietet `hashcat` (abgesehen von einem Brute-Force- bzw. Maskenangriff) nicht die passende Funktion. Auch wenn ein Maskenangriff irgendwann den gesuchten Passwortsatz erraten würde, so würde es jedoch - wie auch schon Nielsen et al., 2014 nachweisen konnten - eine erhebliche Menge an Zeit in Anspruch nehmen. Daher ist bei längeren Passwortsätzen von dieser Angriffsart abzuraten. Somit bietet `hashcat` keine effiziente Möglichkeit einen Diceware-Passwortsatz mit mehr als zwei Wörtern anhand eines Wörterbuchangriffs zu erraten.

3.3 Empfehlungen auf Basis der Testergebnisse

Die Testergebnisse der Angriffsszenarien haben deutlich gemacht, dass es für jede Art von Passwort eine andere Herangehensweise gibt, um den Hashwert eines Passworts mit `hashcat` anzugreifen. `Hashcat` unterstützt den Angreifer hierbei mit einer Auswahl von verschiedenen Angriffsmethoden. Die Testergebnisse haben gezeigt, dass Hintergrundinformationen zum anzugreifenden Hashwert hierbei maßgeblich sind. Je mehr Informationen ein Angreifer über den anzugreifenden Hashwert im vorhinein herausfindet, umso besser kann `hashcat` für den entsprechenden Angriff konfiguriert werden. Manche Informationen sind hierbei zwingend vor einem Angriff in Erfahrung zu bringen, wie zum Beispiel das verwendete Hashingverfahren. Ohne das Verfahren, anhand dessen der anzugreifende Hashwert gebildet wurde, zu kennen, kann `hashcat` keinen Angriff durchführen. Weiterhin hilft das Wissen um das eingesetzte Hashingverfahren um vor einem Angriff abschätzen zu können, welcher zeitliche Aufwand zum Ermitteln des Passworts entsteht. Die Zeit, die `hashcat` benötigt um ein und dasselbe Passwort anhand verschiedener Hashingverfahren zu ermitteln, variiert sehr stark. Während der Tests konnte das Passwort eines SHA-512-Hashwerts teilweise innerhalb weniger Sekunden bis Minuten ermittelt werden, wohingegen `hashcat` für dasselbe Passwort eines scrypt-Hashwerts zum Teil etliche Stunden oder sogar Tage benötigt wurden. Neben dem verwendeten Hashingverfahren hängt die Zeit zum Ermitteln des Passworts auch stark von der Menge an Informationen ab, die ein Angreifer über die Person, die das Passwort erstellt hat, in Erfahrung bringen konnte. Wenn ein Angreifer beispielsweise anhand bereits durch `hashcat` ermittelter Passwörter das Schema kennt, mit welchem die Passwörter entworfen wurden, sowie die durchschnittliche Länge der Passwörter identifizieren konnte, so kann der Angreifer die identifizierten Passwortschemata auf die Angriffsmethoden in `hashcat` übertragen. Angenommen, ein Angreifer weiß, dass das gesuchte Passwort eine Verbindung aus einer Wort- oder Buchstabenkombination, einem Sonderzeichen, sowie einer zwei- bis vierstelligen Zahl ist und in den meisten Fällen eine Passwortlänge von acht Zeichen aufweist, so kann er dieses Wissen auf die maskenbasierten Angriffsmethoden in `hashcat` anwenden. Auf

diese Art und Weise muss `hashcat` nicht alle möglichen Kombinationen von Zahlen, Zeichen und Sonderzeichen durchprobieren (siehe 2.5.1), sondern kann jeden Bereich des Passworts gezielt anhand des definierten Schemas angreifen.

3.3.1 Hashcat-Entscheidungsdiagramm

Im Folgenden werden die verschiedenen Konfigurationsmöglichkeiten von `hashcat` (unter Einbeziehung der gewonnenen Erkenntnisse der durchgeführten Tests) anhand eines Entscheidungsdiagramms dargestellt.

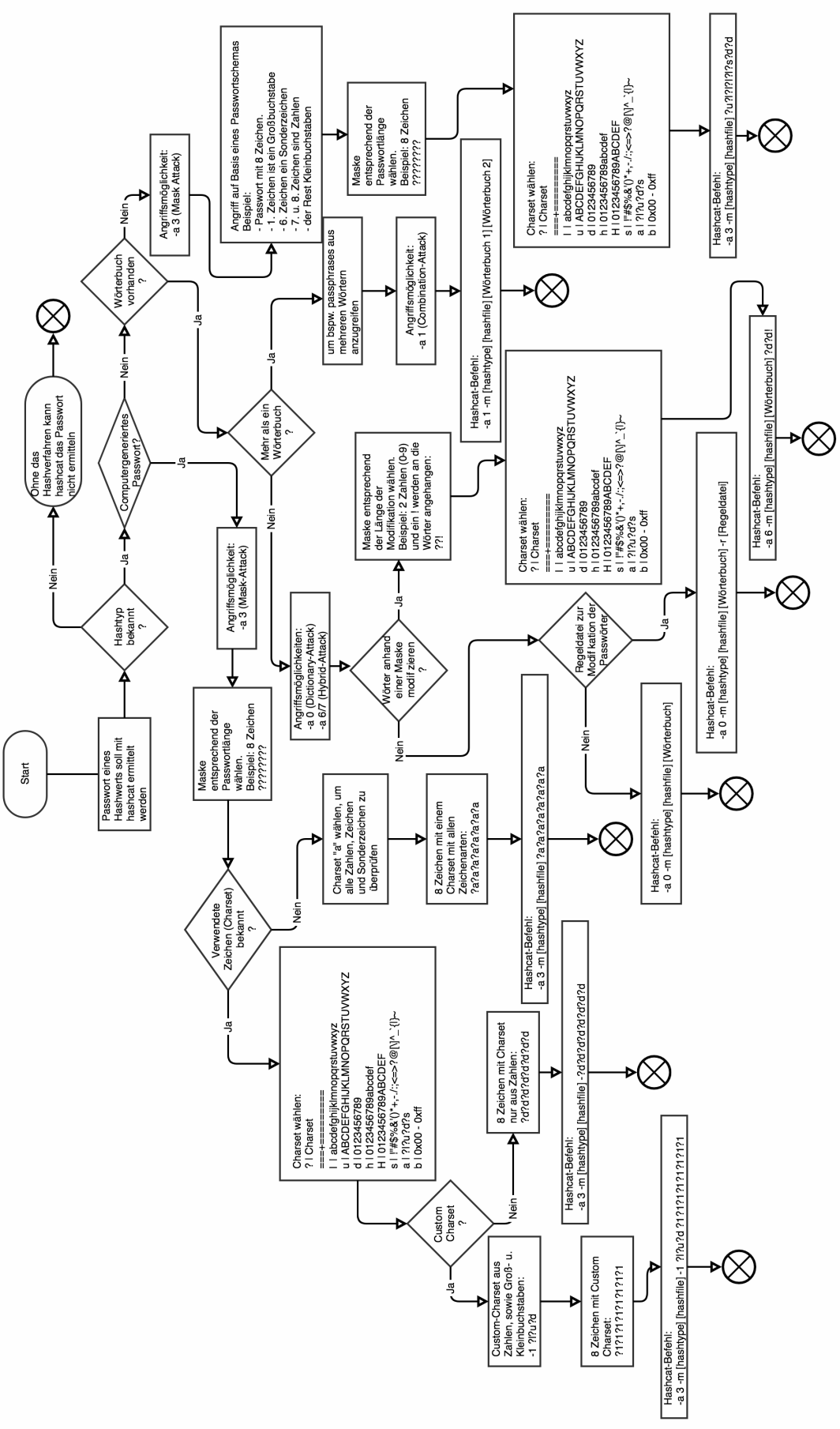

Abbildung 3.1: Hashcat-Entscheidungsdiagramm

3.4 Schwachstellen der Software

Die Testergebnisse zeigen, dass `hashcat` teilweise nur unter großem Aufwand den Anforderungen der Angriffsszenarien gerecht werden konnte. Durch den Test von Szenario 1 wurde deutlich, dass abhängig von den Anforderungen an eine Regeldatei, Regeldateien in `hashcat` schnell sehr komplex und unübersichtlich werden können. Weiterhin hat der Test von Szenario 4 gezeigt, dass der Kombinationsangriff von `hashcat` zum Angriff auf Passwörter, die aus mehr als zwei Wörterbüchern zusammengesetzt werden (sogenannte Passwortsätze, engl. „passphrases“), unzureichend ist, da bei der Konfiguration des Kombinationsangriffs in `hashcat` maximal zwei Wörterbücher angegeben werden können. `Hashcat` selbst bietet in diesem Fall die Möglichkeit über ein Zusatztool namens „combinator“ zwei Wörterbücher zu einem zu vereinen. Hierbei wird jedes Wort aus dem einen mit jedem Wort aus dem anderen Wörterbuch kombiniert um so ein neues Wörterbuch zu erschaffen. So könnte man bei einem Angriff auf einen Diceware-Passwortsatz, das Diceware-Wörterbuch entsprechend der vermuteten Anzahl an verwendeten Wörtern oft genug mit sich selbst kombinieren, um dann mit einem einfachen Wörterbuchangriff (keinem Kombinationsangriff) den Diceware-Passwortsatz anzugreifen. Hierbei entsteht allerdings sehr schnell ein neues Problem. Der Speicherplatz, den beispielsweise ein einzelnes Diceware-Wörterbuch auf einer Festplatte benötigt, liegt bei ca. 46 KB. Wenn man dieses Wörterbuch einmal mit sich selbst kombiniert, sodass jeder Eintrag des Wörterbuchs aus zwei Wörtern besteht, so beträgt der neue benötigte Speicherplatz auf der Festplatte (für das resultierende Wörterbuch) ca. 647,2 MB. Dies macht einen exponentiellen prozentualen Anstieg des Wörterbuch-Speicherplatzes von ungefähr 1.406.856,52% aus. Es wird deutlich, dass die Kombination von Wörterbüchern, die für sich allein schon etliche Tausend Einträge besitzen (siehe Diceware-Wörterbuch, 7.776 Einträge), unter Umständen schnell zu einem Speicherplatzproblem führen kann.

3.4.1 Konzipierung einer Lösung

Für die in den Szenarien 1 und 4 in `hashcat` identifizierten Schwachstellen, werden im Folgenden Lösungsansätze entwickelt. In erster Linie geht es darum, eine abstrakte Lösung für beide der aufgeführten Problemstellungen zu definieren, um festzustellen welche Funktionen `hashcat` bereitstellen müsste um die Problemstellungen zu lösen.

3.4.2 Lösungsansatz Szenario 1

Das zuvor bei Szenario 1 beschriebene Problem bestand darin, dass durch das anhand einer `hashcat`-Regeldatei einfache Anhängen mehrerer Zahlen an die Wörter eines Wörterbuchs, ein verhältnismäßig großer Aufwand entstanden ist. Für jede anzuhängende Zahl musste eine eigene Regel definiert werden. Dies führt bei einer größeren Anzahl anzuhängender Zahlen einerseits zu einem recht großen Aufwand und andererseits zu sehr viel Inhalt innerhalb der Regeldatei.
Ein denkbarer Lösungsansatz wäre die Möglichkeit durch `hashcat` einen Zahlenbereich definieren zu können. `Hashcat` würde dann selbstständig alle Zahlen dieses Bereichs (beispielsweise alle Zahlen zwischen 0 und 9.999) an ein Passwort anhängen. Dies würde den Aufwand zur Erstellung, sowie die Menge an Zeilen innerhalb der Regeldatei, massiv reduzieren. Leider ist eine solche Möglichkeit über eine Regeldatei in `hashcat` aktuell nicht abbildbar. Allerdings bietet `hashcat` eine derartige Option

über den sogenannten „Hybrid-Angriff“ an. Hierbei wird ein maskenbasierter Angriff mit einem Wörterbuchangriff kombiniert. Ein Angreifer kann durch eine entsprechende Maske einen Angriff definieren, bei dem beispielsweise alle Zahlen von 0 bis 9.999 automatisiert an ein Passwort eines Wörterbuchs angehängt werden. Ein Beispiel für einen derartigen Hybrid-Angriff sieht folgendermaßen aus:

hashcat64 -a 6 -m 1700 -1 ?d Pfad/zur/Hashdatei /Pfad/zum/Wörterbuch ?1?1?1?1 –increment

Hierbei würden an jedes Passwort des definierten Wörterbuchs die Zahlen von 0 bis 9.999 angehängt werden. Durch den „–increment“-Befehl wird `hashcat` angewiesen mit einstelligen Zahlen zu beginnen um zum Schluss bei vierstelligen Zahlen aufzuhören. Ohne den „–increment“-Befehl würde `hashcat` alle nicht genutzten Stellen mit einer 0 belegen, wodurch allen einstelligen Zahlen drei Nullen vorausgehen würden. Allerdings ist es nicht möglich einem Hybrid-Angriff eine Regeldatei zur weiteren Modifikation der Passwörter des Wörterbuchs zu übergeben. Daher wäre es ein Lösungsansatz des identifizierten Problems, `hashcat` um eine weitere Angriffsmethode, eine Art „Dreifach-Angriff“, die einen Hybrid-Angriff mit einem regelbasierten Angriff kombiniert, zu erweitern.

3.4.3 Lösungsansatz Szenario 4

Der Test von Szenario 4 hat deutlich gemacht, dass anhand des `hashcat`-eigenen Kombinationsangriffs nur Passwortsätze mit maximal zwei Wörtern angegriffen werden können. `Hashcat` bietet dem Nutzer zwar die Möglichkeit Wörterbücher anhand des `hashcat`-Tools „combinator“ miteinander zu einem neuen Wörterbuch zu kombinieren, allerdings können hierdurch schnell sehr große Wörterbücher, die sehr viel Speicherplatz auf einer Festplatte belegen, entstehen. Der Lösungsansatz nimmt sich hier den `hashcat`-eigenen Kombinationsangriff zum Vorbild. Wie arbeitet der Kombinationsangriff? Weder erstellt `hashcat` während des Angriffs eine neue Wörterbuchdatei, in dem es die Kombinationen abspeichert, noch schreibt `hashcat` die gesamte Liste an kombinierten Wörtern in den Arbeitsspeicher des Computers, da ansonsten sehr schnell ein Großteil des Arbeitsspeichers belegt wäre und dem Computer keine Ressourcen mehr für den eigentlichen Angriff zur Verfügung stehen würden. `Hashcat` übergibt jeden kombinierten Passwortsatz einzeln an den Angriff. Für jeden erzeugten Passwortsatz wird direkt der entsprechende Hashwert errechnet und mit dem Ziel-Hashwert verglichen, wobei bei Nichtübereinstimmung der Passwortsatz verworfen und der nächste Passwortsatz überprüft wird. Auf diese Art und Weise muss `hashcat` weder viel Speicherplatz auf der Festplatte für entsprechend große Wörterbücher belegen, noch wird eine große Menge an Arbeitsspeicher für den Angriff benötigt. Da jede Kombination sofort überprüft wird, wird auch vor dem eigentlichen Angriff keine Zeit benötigt um eine Liste mit allen Hashwerten der bestehenden Kombinationsmöglichkeiten zu errechnen. Daraus ergibt sich die Frage, wie dieses Konzept auf viele Wörterbücher übertragen werden kann. Beim Start eines Angriffs werden alle Wörterbücher eines vordefinierten Verzeichnisses geöffnet, für jedes der Wörterbücher wird genau wie beim `hashcat`-eigenen Kombinationsangriff jeder Eintrag eines Wörterbuchs mit jedem anderen Wörterbuch kombiniert, sodass am Ende ein einzelner Passwortsatz, dessen Wortanzahl der Anzahl der zuvor geöffneten Wörterbücher entspricht, entsteht. Anhand jedes erzeugten Passwortsatzes wird sofort ein Angriff durchgeführt, indem wie zuvor beschrieben für den entsprechenden Passwortsatz ein Hashwert errechnet und mit dem Ziel-Hashwert verglichen wird. Auf diese Art und Weise können beliebig viele Wörterbücher miteinander kombiniert werden, ohne viel Speicherplatz auf der Festplatte eines Computers zu belegen.

3.4.4 Umsetzung der Lösungsansätze

Zur Umsetzung der beschriebenen Lösungsansätze untersuchen wir zwei mögliche Verfahren. Beim ersten der beiden Verfahren geht es darum, den `hashcat`-eigenen Programmcode so zu erweitern, dass für jeden der beschriebenen Lösungsansätze ein neuer Angriffsmodus in `hashcat` abgebildet wird. Dieses Vorgehen würde allerdings ein fortlaufendes Updaten des hinzugefügten Codes bedeuten; jedes Mal, wenn für `hashcat` ein Update veröffentlicht wird. Das zweite mögliche Verfahren ist das Programmieren eines sogenannten „Wrappers“. Der „Wrapper“ ist ein Programm, welches von außen auf `hashcat` zugreift um Werte an `hashcat` zu übergeben und von diesem zu empfangen. Die zuvor beschriebenen Lösungsansätze werden als Funktionen in separaten Modulen implementiert. Beim Ausführen einer der Methoden greift der „Wrapper“ auf `hashcat` zu und übergibt alle (für die jeweilige Angriffsmethode) benötigten Parameter und Werte an `hashcat` und bekommt im Gegenzug jedes errechnete Ergebnis zur Weiterverarbeitung zurück. Aufgrund der Losgelöstheit vom `hashcat`-Programmcode und der daraus resultierenden Kompatibilität zu anderen Programmiersprachen (als der Sprache mit der `hashcat` selbst programmiert wurde) wurde die Umsetzung anhand eines „Wrappers“ gewählt. Der „Wrapper“, der im Grunde als eine Erweiterung von `hashcat` anzusehen ist, wird im Weiteren als „`hashcatXT`“ (Extended Technology (XT), also eine Erweiterung der ursprünglichen Software) bezeichnet oder einfach Tool genannt.

Kapitel 4

Implementierung

Im folgenden Abschnitt wird auf die Entwicklung von „hashcatXT“ eingegangen. Zu Beginn werden der Aufbau und die Funktionen des Tools beschrieben. Hierbei werden wir einerseits auf die grundlegenden Systemanforderungen des Tools eingehen und andererseits die zur Entwicklung des Tools gewählte Programmiersprache, die verwendeten Programmbibliotheken, sowie die implementierten Funktionen beschreiben. Darauf aufbauend folgt eine Beschreibung der Benutzeroberfläche. Im Anschluss daran werden Funktionstests der fertigen Software durchgeführt, bei denen beispielhaft der Start von „`hashcatXT`“ anhand verschiedener Parameter demonstriert wird. Zum Schluss werden verschiedene Anwendungsfälle (engl. „use cases“) aufgezeigt.

4.1 Entwicklungsprozess

Die Entwicklung des Tools wurde weitestgehend auf Basis des aus der Softwareentwicklung bekannten „erweiterten Wasserfallmodells“ durchgeführt. Goll und Hommel, 2015, S. 57 beschreiben das Wasserfallmodell als erstes bedeutendes Entwicklungsmodell der Informatik. Bei einem Wasserfallmodell wird die Entwicklung eines Systems oder einer Software am Stück betrachtet. Die einzelnen Phasen des Vorgehensmodells entsprechen hierbei den Entwicklungsschritten der Software. Dieses Modell wurde gewählt um den Entwicklungsprozess übersichtlicher zu gestalten und die Komplexität des Projekts beherrschbar und vor allem planbar zu machen. Abbildung 4.1 zeigt die einzelnen Phasen des Wasserfallmodells. Das sogenannte „erweiterte Wasserfallmodell“ hat im Gegensatz zum normalen Wasserfallmodell iterative Aspekte (Wasserfallmodell mit Rücksprung). Hierdurch ist ein schrittweises „Aufwärtslaufen“ der einzelnen Phasen des Modells möglich, wenn in der aktuellen Phase etwas nicht nach Plan abgelaufen ist, kann der Fehler in der nächsthöheren Stufe behoben werden.

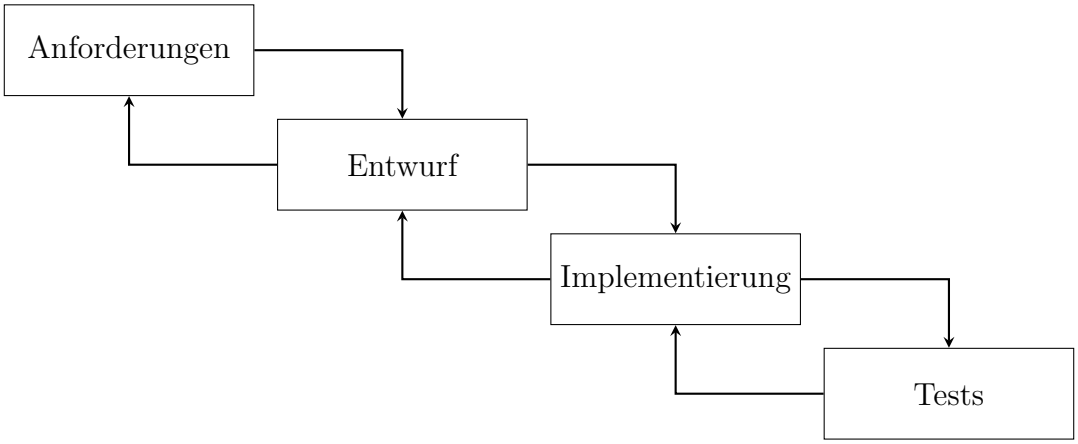

Abbildung 4.1: Wasserfallmodell zur Entwicklung von `hashcatXT`

4.1.1 Anforderungen an die Software

Die `Anforderungen` an die Software leiten sich hierbei aus den zuvor beschriebenen Lösungsansätzen (siehe Kapitel 3.4.1), sowie einer Recherche zur Machbarkeit ab. Bei dieser Recherche ging es um die Frage, inwiefern `hashcat` Möglichkeiten zum Senden und Empfangen von Informationen bietet. Auf der Webseite der Entwickler von `hashcat`[1] wird in einem Artikel[2] des webseiteneigenen „wikis“ beschrieben, dass `hashcat` das Einlesen von Informationen eines anderen Programms anhand des „stdin“-Parameters unterstützt. Auch beschreibt der Artikel die Möglichkeit die Programmausgabe von `hashcat` über den „stdout“-Parameter an ein anderes Programm zu senden.

4.1.2 Entwurf

Auf Basis der Anforderungen aus Kapitel 3.4.1 und den Informationen der Recherche wurde ein `Entwurf` der geplanten Funktionen entwickelt. Der Ablauf der einzelnen Funktionen wurde hierbei anhand eines Ablaufdiagramms schematisch dargestellt. Für Szenario 1 sieht der Ablauf der Funktion, die den Lösungsansatz des Szenarios abbildet (hier als „tripple_attack()“ bezeichnet), wie folgt aus:

[1] https://hashcat.net, abgerufen am 05.04.2018

[2] https://hashcat.net/wiki/doku.php?id=brute_force_in_oclhashcat_plus_original, abgerufen am 05.04.2018

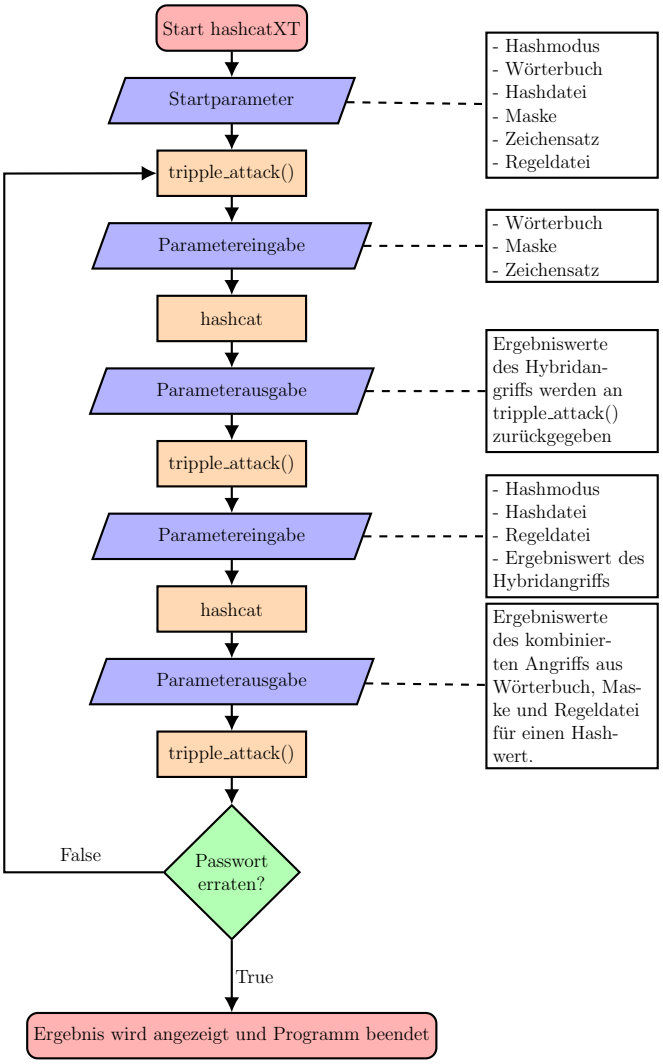

Abbildung 4.2: Ablaufdiagramm Tripple Attack

Der Ablauf der Funktion des zweiten Lösungsansatzes (zu Szenario 4), im Weiteren bezeichnet als „multi_combinator()“, ist wie folgt:

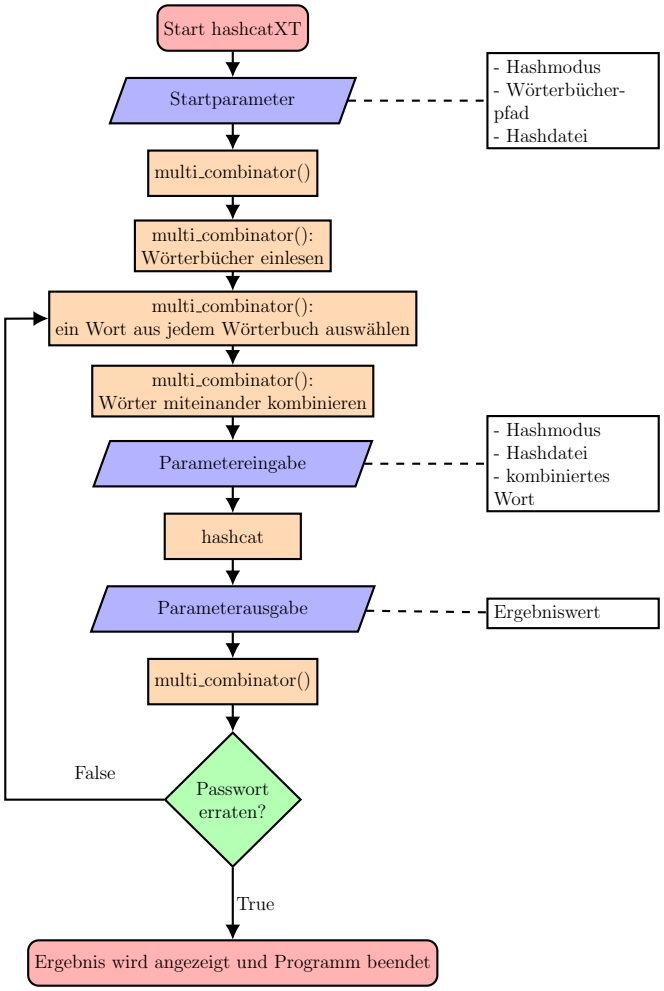

Abbildung 4.3: Ablaufdiagramm Multi-Combinator-Attack

4.1.3 Implementierung der Funktionen und Tests

Anhand der Ablaufdiagramme der zu implementierenden Funktionen begann nun die Entwicklung des Tools und die `Implementierung` der zuvor beschriebenen Funktionen. Die `Überprüfung` der Programmstruktur und der implementierten Funktionen erfolgte nach der erfolgreichen Implementierung anhand von Beispielwerten. Die Beispielwerte entsprachen den auf die entworfenen Angriffsmethoden ausgerichteten Passworthashwerten, sowie mehreren übersichtlichen (selbst erstellten) Wörterbüchern, zum Testen der Kombinationsfunktion des Tools. Entsprechend den Testergebnissen wurden im Anschluss in der `Wartungsphase` Anpassungen am Tool und dem jeweiligen Programmcode vorgenommen.

4.2 Systemanforderungen des Tools

Da das Tool selbst in der Programmiersprache Python (siehe Kapitel 4.3) programmiert wurde, ist das Tool weitestgehend plattformunabhängig, da Python für jedes gängige System und Betriebssystem verfügbar ist. Weitestgehend bedeutet in diesem Zusammenhang, dass zwar die Erweiterung `hashcatXT` aufgrund der gewählten Programmiersprache plattformunabhängig ist, jedoch gilt dies nicht in Bezug auf das Basisprogramm `hashcat`. Dieser Umstand ergibt sich aus der unterschiedlichen Aufrufsweise von `hashcat` unter den Betriebssystemen Mac Os und Linux (beispielsweise Ubuntu) im Gegensatz zu Microsoft

Windows. Da der Aufruf von `hashcat` unter Mac Os und Linux identisch ist, wurde diese erste Version von `hashcatXT` ausschließlich für Mac Os und Linux entwickelt, um Kompatibilitätsprobleme zu vermeiden. Im Bezug auf die verwendete Hardware gibt es keinerlei Einschränkungen, jedoch empfiehlt es sich, wie für `hashcat` auch, ein System mit einer entsprechend guten Hardware auszuwählen um eine entsprechend gute Performance beim Ermitteln der Passwörter zu erreichen. Die einzige systemseitige Voraussetzung ist eine Python-Installation. Bei Mac Os und Linux-Systemen ist Python systemseitig vorinstalliert. Da das Tool jedoch anhand des Python Software Development Kit (SDK) mit der Version 3.6 programmiert wurde und auf Mac Os und Linux-Systemen lediglich die Version 2.7 vorinstalliert ist, ist es nötig eine Installation von Python mit mindestens der Versionsnummer 3.6 durchzuführen. Weiterhin benötigt das Tool eine lauffähige Version der Software `hashcat`. Da „`hashcatXT`“ als „Wrapper“ für `hashcat` gedacht ist, können die Funktionen des Tools ohne eine funktionierende Version von `hashcat` nicht ausgeführt werden. Ansonsten erfordert das Tool dieselben Angaben beziehungsweise Parameter wie `hashcat` auch (siehe hierzu 2.6.2), da wie in Kapitel 3.4.4 bereits beschrieben, das Tool eine Erweiterung von `hashcat` darstellt.

4.3 Wahl der Programmiersprache

Zur Programmierung von „`hashcatXT`“ wurde die Sprache Python in Version 3.6.3[3] verwendet. Python ist eine sehr leicht zu erlernende Programmiersprache und laut Woyand, 2017, S. 1 besonders geeignet zur schnellen Entwicklung von Softwareprototypen. Weiterhin ist es plattformunabhängig, wodurch das programmierte Tool, bezogen auf die Programmiersprache, auf jedem gängigen Betriebssystem eingesetzt werden kann.

4.4 Verwendete Module

Die folgenden Module wurden bei der Entwicklung von „`hashcatXT`“ eingesetzt:

- sys
- os
- docopt
- subprocess
- itertools
- threading

4.4.1 Sys-Modul

Das `sys`-Modul ist Teil der Python-Standardbibliothek und kann immer verwendet werden. Es enthält systemspezifische Funktionalitäten und ermöglicht unter anderem den Zugriff auf Kommandozeilenparameter beim Start eines Python-Programms[4]. Bei `hashcatXT` wurden anhand des Befehls „sys.path.insert()“ Verzeichnisspfade zu verwendeten Modulen, aber auch zu den selbst entworfenen Modulen gespeichert. Das ermöglichte innerhalb des Programmcodes den direkten Zugriff auf die Dateien innerhalb dieser Verzeichnisse, ohne jedes Mal den gesamten Pfad auf der Festplatte angeben zu müssen.

[3] https://www.python.org/downloads/release/python-363/, abgerufen am 07.04.2018
[4] https://docs.python.org/3.6/library/sys.html, abgerufen am 07.04.2018

4.4.2 Os-Modul

Das `os`-Modul ist Teil der Python-Standardbibliothek und kann immer verwendet werden. Es dient der Interaktion mit dem Betriebssystem[5]. Mit ihm können beispielsweise aus Python heraus auf Prozesse des Betriebsystems oder auch des Dateisystems zugegriffen werden. Somit ist es möglich den Inhalt eines Verzeichnisses auszulesen, um beispielsweise festzustellen, wie viele beziehungsweise welche Dateien sich in einem Verzeichnis befinden. In `hashcatXT` wurde das `os`-Modul unter anderem benutzt um beim Definieren der Programmverzeichnisse (anhand des Befehls „sys.path.insert()“) keinen Pfad angeben zu müssen. Durch Verwendung des Befehls „os.getcwd()“ ist es möglich eine relative Pfadangabe auf Basis des Verzeichnisses, in dem sich `hashcatXT` befindet, durchzuführen. Weiterhin wurde das `os`-Modul innerhalb der Funktion „multi_combinator()“ benutzt, um einen Verzeichnispfad auszulesen und auf die in dem Verzeichnis befindlichen Wörterbücher zugreifen zu können.

4.4.3 Docopt-Modul

Das `docopt`-Modul ist ein externes Modul und nicht Teil der Python-Standardbibliothek. Es muss aus dem Internet heruntergeladen[6] und unter Angabe des Dateipfades in ein Python-Projekt importiert werden. Das `docopt`-Modul wurde von Vladimir Keleshev entwickelt. Die Entwicklerwebseite[7] beschreibt es als „Parser“ für Befehlszeilenparameter oder auch „Command-line interface description language“. Der Vorteil von `docopt` ist die Definition der Befehlszeilenparameter eines kommandozeilenbasierten Programms anhand der Beschreibung seiner Optionen und Argumente. Häufig wird die Dokumentation eines Programms und der mit dem Programm ausführbaren Befehle im Anschluss an die Entwicklung durchgeführt, um beispielsweise anhand von Hilfeseiten die Funktionen zu beschreiben. `Docopt` kehrt diesen Schritt um. Der Entwickler beschreibt zuerst die Optionen und Argumente, sowie die Syntax zum Aufruf von Funktionen des Programms und `docopt` interpretiert diese Beschreibung und definiert entsprechende Parameter um auf das Programm zuzugreifen. Die von `docopt` interpretierte Beschreibung muss dem sogenannten „Docstring“-Konzept[8] folgen. Dieses basiert auf den Gepflogenheiten zur Schreibweise für Hilfetexte und Handbuchseiten, wie sie seit Jahrzehnten unter Linux-Systemen üblich sind. Auf diese Art und Weise werden die Befehlszeilenparameter des Programms definiert und gleichzeitig wird das Programm und seine Funktionen dokumentiert. Nachfolgend ein Beispiel von der `docopt`-Entwicklerwebseite für den Einsatz von docopt:

Listing 4.1: Python docopt Beispiel

```
"""Naval Fate.

Usage:
  naval_fate.py ship new <name>...
  naval_fate.py ship <name> move <x> <y> [--speed=<kn>]
  naval_fate.py ship shoot <x> <y>
  naval_fate.py mine (set|remove) <x> <y> [--moored | --drifting]
  naval_fate.py (-h | --help)
  naval_fate.py --version

```

[5] https://docs.python.org/3.6/library/os.html, abgerufen am 09.04.2018
[6] https://github.com/docopt/docopt, abgerufen am 09.04.2018
[7] http://docopt.org/, abgerufen am 09.04.2018
[8] https://www.python.org/dev/peps/pep-0257/, abgerufen am 10.04.2018

```
11 Options:
12   -h --help    Show this screen.
13   --version    Show version.
14   --speed=<kn> Speed in knots [default: 10].
15   --moored     Moored (anchored) mine.
16   --drifting   Drifting mine.
17
18 """
19
20
21 from docopt import docopt
22
23 if __name__ == '__main__':
24     arguments = docopt(__doc__, version='Naval Fate 2.0')
25     print(arguments)
```

Die Zeilen 1 bis 18 vom Listing 4.1 zeigen die docstring-basierte Beschreibung. Diese beginnt und endet mit drei doppelten Anführungsstrichen, anhand derer `docopt` den zu interpretierenden Bereich identifizieren kann. Dieser Bereich ist unterteilt in einen „Usage“- und einen „Options“-Bereich. Im `Usage`-Bereich werden mögliche Befehle anhand von Befehlsmustern definiert. Einerseits werden diese Befehlsmuster dem Nutzer bei falschen Eingaben oder dem Aufruf der Hilfeseite über den „-h“-Parameter eingeblendet (um so den Nutzer bei der Bedienung des Programms zu unterstützen) und andererseits wird über die definierten Muster festgelegt, welche Befehle beim Aufruf des jeweiligen Programms zulässig sind. Jedes der Befehlsmuster beginnt mit dem Namen des Tools beziehungsweise dem Namen der aufgerufenen Moduldatei (Listing 4.1 Zeile 4: „naval_fate.py“), danach folgen zulässige `Optionen` und `Argumente`. `Optionen` sind Befehlszeilenparameter über die auf bestimmte Funktionen eines Programms zugegriffen werden können. Sie sind durch einen oder zwei Bindestriche gekennzeichnet. Hier wird zwischen der kurzen Form „-h“ und der langen Form „−−help“ unterschieden, wobei beide Varianten wie im Listing 4.1 Zeile 8 zu sehen ist, gleichermaßen eingesetzt werden können. Wenn eine `Option` namentlich mit mehr als einem Buchstaben dargestellt werden soll, müssen immer zwei Bindestriche verwendet werden, ansonsten unterscheiden sich beide Formen nicht voneinander. Die im `Usage`-Bereich benutzten Optionsparameter müssen hierbei immer den Parametern des `Options`-Bereichs entsprechen. Neben Optionsparametern gehören auch `Argumente` (dargestellt anhand von spitzen Klammern „<Argument>“) zum `Usage`-Bereich. `Argumente` dienen dazu dem Programm dynamische Inhalte zu übergeben. Sie können beim Aufruf eines Programms alleinstehend oder in Kombination mit einer vorher angegebenen `Option` übergeben werden. Bei der Angabe eines `Arguments` in Verbindung mit einer `Option`, wird der Wert des `Arguments` der Variable der `Option` zugeordnet. Beispielsweise wird in Listing 4.1 Zeile 5 das `Argument` <kn> definiert. Hier kann ein Nutzer einen frei gewählten Wert für <kn> angegeben, welcher beim Start des Programms der `Option` „−−speed“ zugeordnet wird. Zu beachten ist hier, dass die `Option` „−−speed“ und das `Argument` <kn> von eckigen Klammern [−−speed=<kn>] umschlossen sind. Im Usage-Bereich von eckigen Klammern umschlossene Elemente werden von `docopt` als optionale Parameter angesehen. Der Nutzer kann hier demnach selbst entscheiden, ob er die jeweilige `Option` oder das `Argument` beim Start des Programms angeben möchte oder nicht. Im `Options`-Bereich werden alle (nicht nur die im Usage-Bereich benutzten) Optionsparameter aufgelistet. Wenn für eine `Option` beide Formen (kurz und lang) definiert werden sollen, können beide Formen nebeneinander (durch ein Leerzeichen getrennt) gelistet werden. Rechts neben einer definierten `Option` kann (mit einem Mindestabstand von zwei Leerzeichen zu den Optionen) eine Beschrei-

bung der `Option` beziehungsweise ihrer Funktion angegeben werden. Bei Bedarf kann in der Beschreibung einer `Option` ein Standardwert, welcher der `Option`-Variable zugeordnet wird, wenn der Nutzer keinen Wert festgelegt hat, zugeordnet werden. In den Zeilen 21 bis 25 von Listing 4.1 wird dargestellt, wie `docopt` in ein Projekt eingebunden werden kann und wie auf die durch `docopt` erzeugten Parameter zugegriffen werden kann. In `hashcatXT` wurden die Kommandozeilenparameter und `Argumente` zum Aufruf des Programms anhand von `docopt` definiert. Ebenso wurde mit `docopt` die Dokumentation der über `hashcatXT` verfügbaren Funktionen und deren Syntax über eine entsprechende Hilfeseite abgebildet.

4.4.4 Subprocess-Modul

Das `subprocess`-Modul ist Teil der Python-Standardbibliothek und kann immer verwendet werden. Es ermöglicht den Aufruf von Programmen und Systemtools und baut eine Verbindung zu deren Dateneingabe, Datenausgabe, sowie bei der Ausführung auftretenden Fehlern auf[9]. In `hashcatXT` wird das `subprocess`-Modul benutzt um `hashcat` aufzurufen, Eingabewerte an `hashcat` zu übergeben und die Ausgabewerte von `hashcat` zu empfangen und in `hashcatXT` weiterzuverarbeiten.

4.4.5 Itertools-Modul

Das `itertools`-Modul ist Teil der Python-Standardbibliothek und wird mit jeder Python-Installation installiert. Mit `itertools` können diverse Typen von Iteratoren erzeugt und bearbeitet werden[10]. Ein Beispiel hierfür ist die Funktion „itertools.product()“, mit welcher es möglich ist das kartesische Produkt der Werte zweier Listen zu erzeugen. In `hashcatXT` werden mit dem `itertools`-Modul in der Funktion „multi_combinator()“ Wörter aus verschiedenen Wörterbüchern miteinander kombiniert um anhand der Kombinationen Passwortsätze zu bilden.

4.4.6 Threading-Modul

Das `threading`-Modul gehört zur Python-Standardbibliothek und ist Bestandteil jeder Python Installation. Mit dem `threading`-Modul können einzelne Funktionen in separaten Threads (Ausführungssträngen) ausgeführt werden, um beispielsweise zwei Funktionen zur selben Zeit, also parallel, auszuführen[11]. In `hashcatXT` wird das Modul benutzt um kontinuierlich während eines Angriffs die Ausgabe von `hashcat` beziehungsweise `hashcatXT` auf dem Bildschirm des Nutzers auszugeben. Ohne das `threading`-Modul würde der Nutzer erst, wenn der Angriff beendet ist, eine Ausgabe sehen können.

4.5 Implementierte Module und Funktionen

`HashcatXT` besteht aus drei einzelnen Modulen (der gesamte Programmcode von `hashcatXT` wurde in Anhang C an die Arbeit angehangen), die jeweils Funktionen abbilden. Die Module sind:

- hashcatXT.py
- tripple_attack.py

[9] https://docs.python.org/3.6/library/subprocess.html, abgerufen am 12.04.2018
[10] https://docs.python.org/3.6/library/itertools.html, abgerufen am 12.04.2018
[11] https://docs.python.org/3.6/library/threading.html, abgerufen am 12.04.2018

- multi_combinator.py

Jedes der Module greift auf unterschiedliche Funktionen von importierten Modulen aus der Python eigenen Standardbibliothek, sowie auch von externen Entwicklern zurück (Kapitel 4.4). Abbildung 4.4 veranschaulicht die Struktur von `hashcatXT` und zeigt, zu welchem Modul welche Funktionen gehören. Weiterhin wird auch deutlich, wo Module anderer Anbieter in `hashcatXT` eingebunden werden.

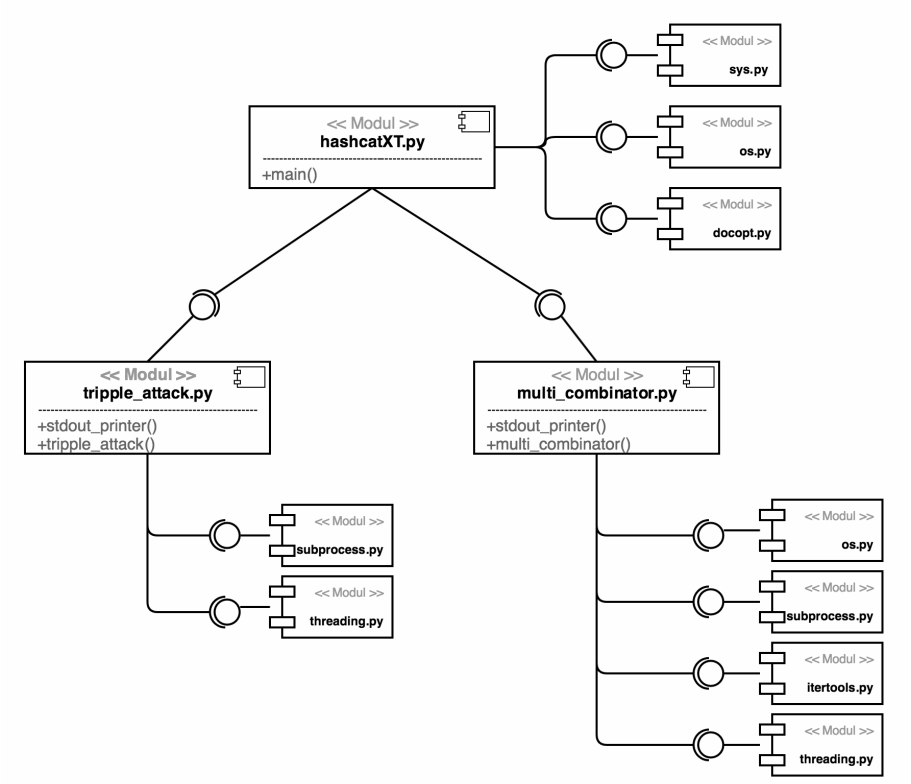

Abbildung 4.4: Komponentendiagramm, Module und Funktionen von hashcatXT

4.5.1 Modul 1: HashcatXT.py

Über das Modul hashcatXT.py wird anhand des docopt-Moduls die docstring-basierte Beschreibung des Tools, sowie die Definition der Befehlszeilenparameter abgebildet. Die Beschreibung besteht hierbei aus

- dem Usage-Bereich,
- einem Options-Bereich,
- sowie optionalen Beschreibungen.

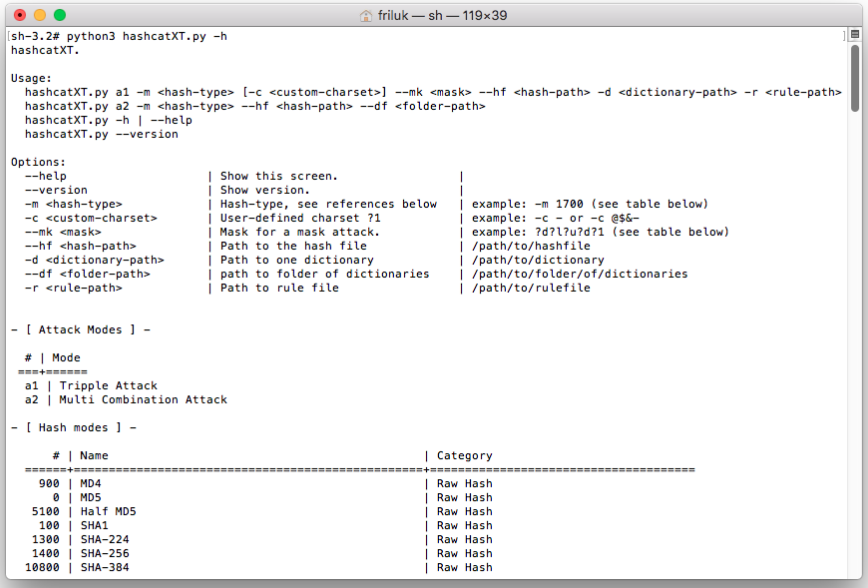

Abbildung 4.5: hashcatXT docopt Docstring

Wie in Abbildung 4.5 zu sehen, werden im Usage-Bereich vier Befehlsmuster definiert. Die ersten beiden Muster sind für die zwei implementierten Angriffsmethoden zuständig. Hierdurch sieht ein Nutzer welche `Optionen` und `Argumente` für welche Angriffsmethode benötigt werden. Das erste Befehlsmuster stellt den Aufruf der „Tripple Attack“-Funktion dar. Das Muster zeigt, dass für einen erfolgreichen Aufruf der Funktion folgende Parameter angegeben werden müssen:

- a1
- -m <hash-type>
- [-c <custom-charset>]
  - Die Option und das dazugehörige Argument wurden in eckige Klammern gesetzt, um sie als optional zu kennzeichnen.
- −−mk <mask>
- −−hf <hash-path>
- -d <dictionary-path>
- -r <rule-path>

Das zweite Befehlsmuster demonstriert den Aufruf der zweiten implementierten Angriffsmethode, der Funktion „multi_combinator()“. Die Parameter hierfür lauten:

- a2
- -m <hash-type>
- −−hf <hash-path>
- −−df <folder-path>

Wenn die Funktionen nicht entsprechend der Befehlsmuster aufgerufen werden, weil beispielsweise ein Parameter (`Option` oder `Argument`) vergessen wurde, wird der Befehl nicht ausgeführt und dem Nutzer werden stattdessen die im Usage-Bereich definierten Befehle angezeigt. Die letzten beiden Muster zeigen einem Nutzer, wie die Hilfeseite des Tools aufgerufen und wie die Versionsnummer des Tools angezeigt werden kann. Im `Options`-Bereich werden alle (nicht nur die in den Befehlsmustern des Usage-Bereichs benutzten) Optionparameter abgebildet. Für `hashcatXT` wurden folgende Optionparameter definiert:

- -h / −−help
  - Wenn dieser Parameter beim Aufruf des Programms angegeben wird, wird die Hilfeseite des Programms auf dem Bildschirm des Nutzers anzeigt. Diese beinhaltet den Usage-Bereich, den Options-Bereich sowie weitere Beschreibungen zum Tool.
- −−version
  - Der Parameter zeigt dem Nutzer die Versionsnummer der verwendeten `hashcatXT`-Version an.
- -m <hash-type>
  - Beim Start des Tools wird über diesen Parameter der Hashtyp (also das Verfahren mit dem der Hashwert erzeugt wurde) des anzugreifenden Hashwerts angegeben. Alle unterstützten „Hash-Modes“ können einer Tabelle der Hilfeseite von `hashcatXT` entnommen werden.
- -c <custom-charset>
  - Über den „Custom Charset“-Parameter wird der zu verwendende Zeichenraum für einen maskenbasierten Angriff definiert. Alle für den maskenbasierten Angriff zu verwendenden Zeichen können hier definiert werden.
- −−mk <mask>
  - Der Parameter legt die Länge der Angriffsmaske fest. Bei einem regulären maskenbasierten Angriff über `hashcat` würde die Maskenlänge der Länge des gesuchten Passworts entsprechen. Dementgegen bei einem `hashcat`-eigenen Hybridangriff oder, im Fall von `hashcatXT`, einem Tripple-Angriff, entspricht die Länge der Maske, der Menge an Zeichen die an ein Wort eines Wörterbuchs angehängt werden. Welche Zeichen, für eine Maske verwendet werden können, kann der Hilfeseite von `hashcatXT` entnommen werden.
- −−hf <hash-path>
  - Über diesen Parameter gibt der Nutzer einen Festplattenpfad zu einer Hashdatei an, die mit `hashcatXT` angegriffen werden soll.
- -d <dictionary-path>
  - Bei diesem Parameter kann der Nutzer einen Pfad auf der Festplatte zu einer Wörterbuchdatei, die für einen Angriff benutzt werden soll, angeben.
- −−df <folder-path>

  - Für einen Multi-Combination-Angriff kann ein Nutzer über diesen Parameter einen Festplattenpfad zu einem Verzeichnis, welches mehrere Wörterbücher enthält, angeben.

- -r <rule-path>

  - Anhand dieses Parameters kann ein Nutzer einen Festplattenpfad zu einer `hashcat`-Regeldatei angeben.

Nach dem `Options`bereich folgen weitere Beschreibungen und Informationen zur Nutzung von `hashcatXT`. Diese Beschreibungen sind optional und dienen nicht der Funktionalität von docopt, sondern haben den alleinigen Zweck dem Nutzer mehr Informationen zur Nutzung von `hashcatXT` zu geben.

Die Funktion main()

Neben der Definition der Befehlszeilenparameter in `docopt` wird anhand einer Bedingung innerhalb der „main()“-Funktion überprüft, an welche Funktion die beim Start des Programms übergebenen Parameter übergeben werden müssen (Listing 4.2 Zeile 3 bis 6). In Listing 4.2 Zeile 3 und 4 wird überprüft, ob der Befehlszeilenparameter „a1“ den Wert „True“ hat, wenn dem so ist, werden die beim Start des Programms vom Nutzer übergebenen Parameter an die Funktion „tripple_attack()“ des importierten Moduls „tripple_attack.py“ übergeben.

Listing 4.2: Modul hashcatXT.py main() Funktion und Aufruf

```
1 def main():
2     args = docopt(__doc__, version='hashcatXT 1.0')
3     if (args['a1'] == True):
4         tripple_attack.tripple_attack(args['-d'], args['--mk'], args['-c'],
              args['-m'], args['--hf'], args['-r'])
5     if (args['a2'] == True):
6         multi_combinator.multi_combinator(args['-m'], args['--hf'],
              args['--df'])
7
8 if __name__ == '__main__':
9     main()
```

In Zeile 5 und 6 wird überprüft, ob der Parameter „a2“ den Wert „True“ besitzt. Wenn dies zutrifft, werden alle beim Start von `hashcatXT` angegebenen Parameter an die Funktion „multi_combinator()“ des importierten Moduls „multi_combinator.py“ übergeben.
Die Zeilen 8 und 9 dienen dem Aufruf der „main()“-Funktion. Beim Start eines Python-Moduls wird dessen Laufzeitvariable „__name__“ der Wert „__main__“ zugewiesen, dies wird in Zeile 8 überprüft. Da `hashcatXT` immer über das „hashcatXT.py“-Modul gestartet werden muss, wird diese Bedingung immer zutreffen. Beim Zutreffen der Bedingung aus Zeile 8 wird in Zeile 9 die „main()“-Funktion aufgerufen.

4.5.2 Modul 2: tripple_attack.py

Das Modul `tripple_attack.py` bildet den Lösungsansatz aus Kapitel 3.4.2 ab. Dies bedeutet, dass das anhand der Parameter (welche an die Funktion tripple_attack() übergeben wurden) ein kombinierter Angriff aus einem Hybridangriff und einem regelbasierten Angriff (über `hashcat`) durchgeführt wird. Für die Umsetzung wurden die Module `subprocess.py`

(Kapitel 4.4.4) und `threading.py` (Kapitel 4.4.6) importiert um auf Funktionen dieser Module zugreifen zu können. Das `tripple_attack.py` Modul selbst besteht aus zwei Funktionen:

- tripple_attack()
- stdout_printer()

tripple_attack()

Die Funktion `tripple_attack()` (Listing 4.3) bildet die Hauptfunktion des Moduls ab. An diese werden die Startparameter von `hashcatXT` übergeben, sofern `hashcatXT` mit dem „a1"-Parameter gestartet wurde. Die Funktion startet `hashcat` hierbei als Prozess, um einen Hybridangriff sowie einen regelbasierten Angriff durchzuführen. Dabei werden anhand des Hybridangriffs lediglich mögliche Passwörter gebildet, welche dann (ohne einen Angriff durchzuführen) an `hashcatXT` zurückgegeben werden. `HashcatXT` startet anschließend den zweiten `hashcat`-Prozess, um einen regelbasierten Angriff durchzuführen. An diesen Angriff werden die zuvor durch den Hybridangriff erzeugten Passwörter übergeben. Die Passwörter werden daraufhin anhand der, für den regelbasierten Angriff, definierten Regeln weiter modifiziert und dann von `hashcat` für einen Angriff auf die (über `hashcatXT` übergebene) Hashdatei benutzt. Der Status, beziehungsweise das Ergebnis des Angriffs, wird hierbei fortlaufend auf dem Bildschirm des Nutzers ausgegeben. Dies passiert anhand eines parallel zum Angriff laufenden Threads, der mithilfe der Funktion stdout_printer() die Ergebnisse auf dem Bildschirm des Nutzers ausgibt.

Listing 4.3: Modul: tripple_attack.py Funktion: tripple_attack()

```
1  def tripple_attack(dictionary, mask, charset, hash_mode, hash_file,
       rule_file):
2      if charset is None:
3          hybrid_attack = Popen(['hashcat', '-a', '6', dictionary, mask,
               '-i', '--stdout'], stdout=PIPE)
4      else:
5          hybrid_attack = Popen(['hashcat', '-a', '6', dictionary, '-1',
               charset, mask, '-i', '--stdout'], stdout=PIPE)
6      rule_attack = Popen(['hashcat', '-a', '0', '-m', hash_mode, hash_file,
           '-r', rule_file, '-O', '--potfile-disable'],
7                           stdin=hybrid_attack.stdout, stdout=PIPE)
8      p = threading.Thread(target=stdout_printer, args=(rule_attack,))
9      p.start()
10     p.join()
```

stdout_printer()

Listing 4.4: Modul: tripple_attack.py Funktion: stdout_printer()

```
1  def stdout_printer(p):
2      for line in p.stdout:
3          print(line.rstrip().decode('utf-8'))
```

Die in Listing 4.4 dargestellte Funktion „stdout_printer()" wird anhand des in der Funktion „tripple_attack()" erzeugten Threads parallel zur Funktion „tripple_attack() " ausgeführt.

Über diese wird kontinuierlich die Datenausgabe der Funktion „tripple_attack()“ auf dem Bildschirm des Nutzers ausgegeben.

4.5.3 Modul 3: multi_combinator.py

Der in Kapitel 3.4.3 beschriebene Lösungsansatz wird anhand des Moduls „multi_combinator.py“ abgebildet. Wie in 3.4.3 beschrieben geht es darum, die Wörter mehrerer Wörterbücher zu einem Passwortsatz zu kombinieren um anhanddessen über `hashcat` einen Wörterbuchangriff durchzuführen. Für die Umsetzung des Moduls wurden die Module

- os.py (4.4.2),
- subprocess.py (4.4.4),
- itertools.py (4.4.5),
- sowie threading.py (4.4.6)

importiert, um auf deren Funktionen zugreifen zu können. Wie auch das Modul „tripple_attack.py“ besteht dieses Modul aus zwei Funktionen:

- multi_combinator()
- stdout_printer()

multi_combinator()

Die Funktion multi_combinator() (Listing 4.5) bildet die Hauptfunktion des Moduls ab, an sie werden (sofern beim Start von `hashcatXT` der Parameter „a2“ angegebenen wurde) die Startparameter von `hashcatXT` übergeben. Die Funktion liest das ihr über die Variable „directory“ übergebene Verzeichnis aus und ordnet die Wörterbücher innerhalb des Verzeichnisses einer Liste zu. Im Anschluss daran wird, anhand der Liste, jede Wörterbuchdatei geöffnet und deren Inhalt in eine weitere Liste geladen. Nun werden zuerst Permutationen für alle Wörter der geöffneten Wörterbücher erstellt (um die Wörter der Wörterbücher miteinander zu kombinieren) und anschließend wird für das Ergebnis das kartesische Produkt erzeugt, um so auch die Kombinationen zu erzeugen die durch die Permutationen nicht erzeugt werden konnten. Anschließend wird jede der Kombinationen (als Passwortkandidat) an einen über die Popen-Funktion (subprocess-Modul (4.4.4)) gestarteten `hashcat`-Prozess übergeben. `Hashcat` führt anhand der übergebenen Passwortkandidaten einen Wörterbuchangriff durch. Parallel hierzu ist ein Thread aktiv, der die Ergebnisse beziehungsweise den Status von `hashcat` kontinuierlich an die Funktion „stdout_printer()“ übergibt.

Listing 4.5: Modul: multi_combinator.py Funktion: multi_combinator()

```
1  def multi_combinator(hash_mode, hash_file, directory):
2      erg = Popen(['hashcat', '-a', '0', '-m', hash_mode, hash_file, '-O',
           '--potfile-disable'],
3                  stdin=PIPE,
4                  stdout=PIPE,
5                  stderr=STDOUT,
6                  universal_newlines=True)
7      t = threading.Thread(target=stdout_printer, args=(erg,))
8      t.start()
9      file = []
10     with os.scandir(directory) as listOfEntries:
```

```
11          for entry in listOfEntries:
12              if entry.is_file() and entry.name is not ".DS_Store":
13                  file.append(open(directory + entry.name).readlines())
14          file = list(itertools.permutations(file))
15          for b in range(0, len(file)):
16              for i in itertools.product(*file[b]):
17                  test = '\n'.join(i).replace("\n", "")
18                  # print(test + "\n")
19                  erg.stdin.write((test + "\n"))
20                  erg.stdin.flush()
21      erg.stdin.close()
22      t.join()
```

stdout_printer()

Listing 4.6 stellt die Funktion „stdout_printer()“ dar. Sie wird durch den in der Funktion „multi_combinator()“ erzeugten Thread parallel zur „multi_combinator()“ Funktion ausgeführt. Sie gibt kontinuierlich die durch die „multi_combinator()“ Funktion übergebenen Werte auf dem Bildschirm des Nutzers aus.

Listing 4.6: Modul: multi_combinator.py Funktion: stdout_printer()

```
1 def stdout_printer(p):
2     for line in p.stdout:
3         print(line.rstrip())
```

4.6 Die Benutzeroberfläche

Da `hashcatXT` ein konsolenbasiertes Programm ist, hat es keine grafische Oberfläche. Die Steuerung des Programms, sowie die Ausgabe der Ergebnisse geschieht komplett über die Konsole des Nutzers (in Textform). Die gesamte Konfiguration von `hashcatXT` erfolgt über die beim Start des Programms angegebenen Parameter. Nachdem das Programm gestartet wurde, müssen keine weiteren Eingaben durch den Nutzer gemacht werden. Die folgenden Abbildungen zeigen das Interface von `hashcatXT` anhand verschiedener Eingaben.

Abbildung 4.6: Start von `hashcatXT` ohne Parameter

Abbildung 4.6 zeigt die Ausgabe von `hashcatXT` beim Start des Programms ohne die Angabe von Parametern. Dem Nutzer werden für `hashcatXT` zulässige Eingaben angezeigt.

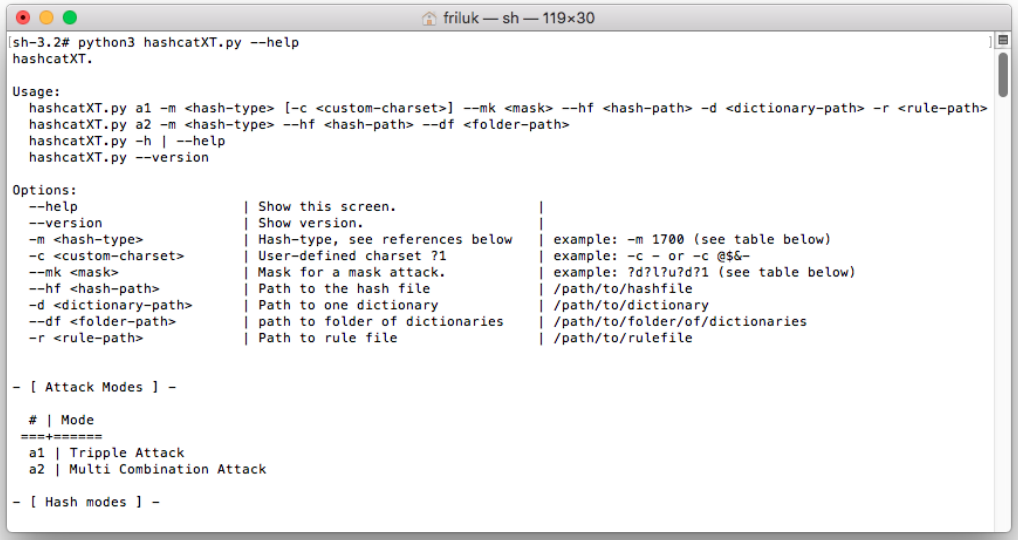

Abbildung 4.7: Aufruf der Hilfeseite von `hashcatXT`

Abbildung 4.7 zeigt den Aufruf der Hilfeseite von `hashcatXT` unter Angabe des Parameters „−−help“. Auf der Hilfeseite werden dem Nutzer Beispiele für Aufrufe des Programms, sowie eine Auflistung aller zulässigen Befehle angezeigt. Im Anschluss an die Auflistung der zulässigen Befehle folgen Tabellen, die dem Nutzer weitere Informationen für einen erfolgreichen Aufruf des Programms geben.

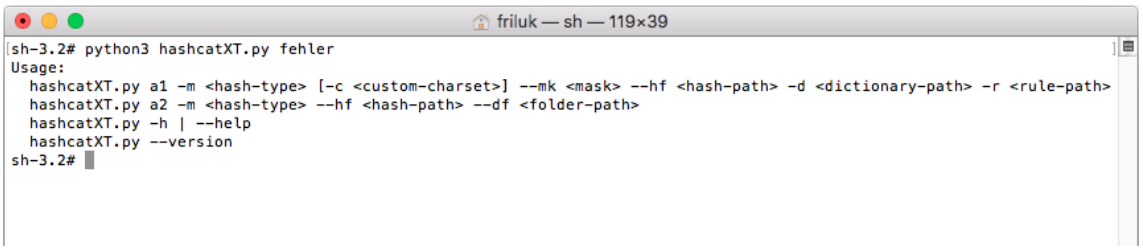

Abbildung 4.8: Falsche Syntax beim Start von `hashcatXT`

Abbildung 4.8 zeigt die Fehlerseite, welche dem Nutzer angezeigt wird, wenn beispielsweise ein falscher Parameter beim Start von `hashcatXT` angegeben wurde. Anhand der über das docopt-Modul (4.4.3) definierten Regeln zum Aufruf des Programms werden dem Nutzer hier zulässige Eingaben vorgeschlagen. Die angezeigte Fehlerseite entspricht inhaltlich der Ausgabe, die dem Nutzer angezeigt wird, wenn `hashcatXT` ohne einen Parameter gestartet wird.

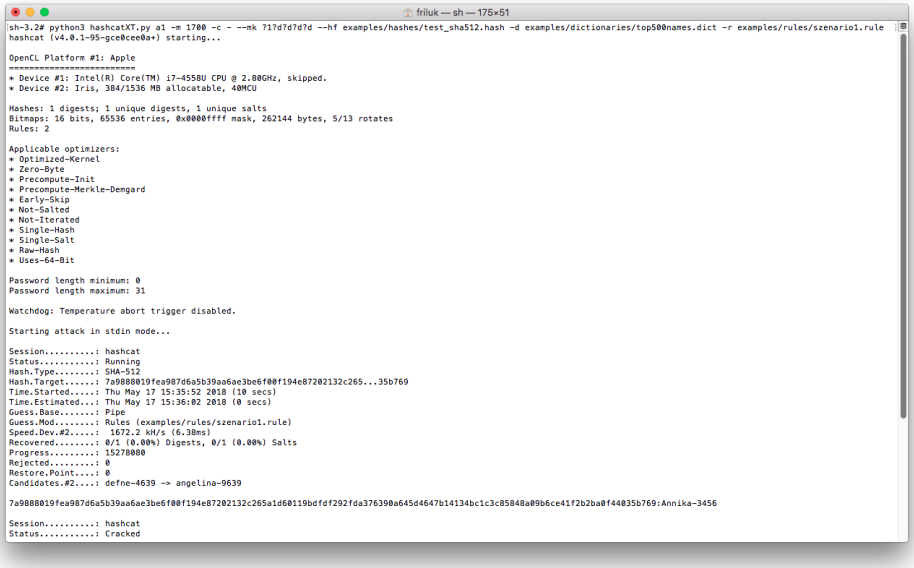

Abbildung 4.9: Starten eines Tripple-Angriffs

Abbildung 4.9 zeigt beispielhaft den Aufruf der Funktion „tripple_attack“ über `hashcatXT`. Der Zugriff auf die Funktion erfolgt durch die Angabe des Startparameters „a1“.

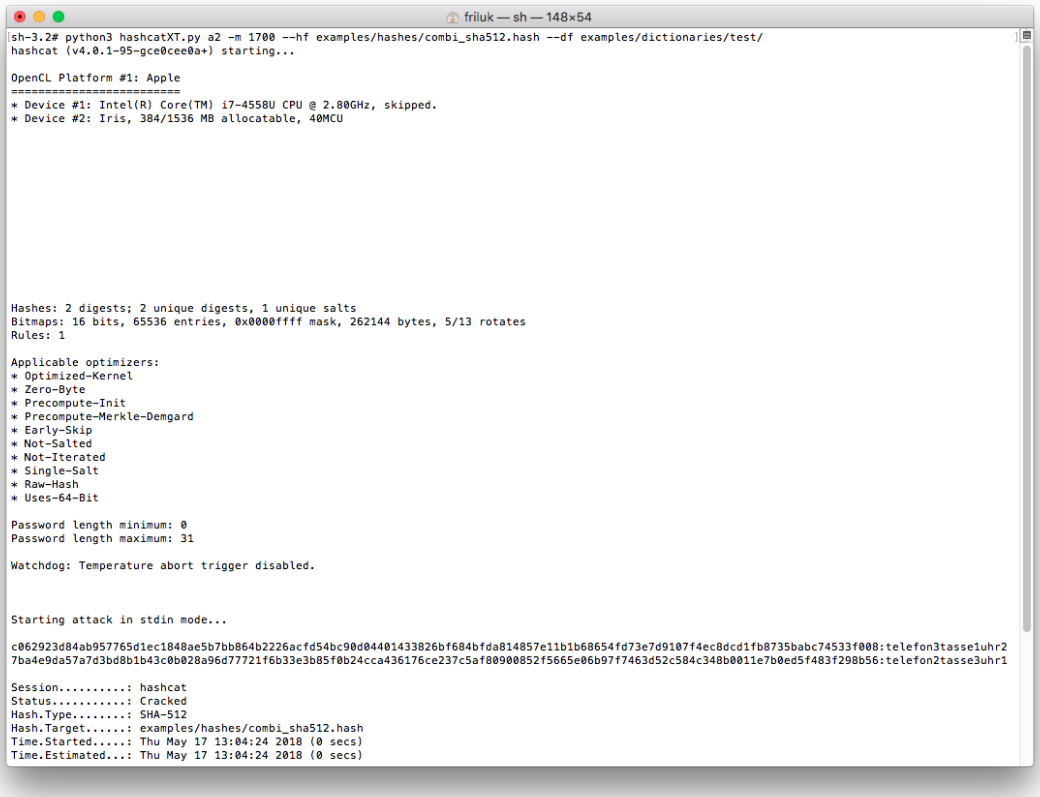

Abbildung 4.10: Starten eines Multi-Combinator-Angriffs

Abbildung 4.10 zeigt den Aufruf der Multi-Combinator-Angriffsmethode. Der Zugriff auf die Funktion erfolgt über die Angabe des Startparameters „a2“.

4.7 Funktionstests

In den nachfolgenden Funktionstests werden die fertiggestellten Funktionen „tripple_attack()“ und „multi_combinator()“ aus der Sicht des Benutzers geprüft. Die Tests werden funktionsorientiert durchgeführt und dienen dazu Abweichungen der Funktionalität gegenüber den definierten Anforderungen (Kapitel 3.4.2 und 3.4.3) aufzudecken.

4.7.1 Funktionstest 1

Um die Funktion „tripple_attack“ zu testen, wird `hashcatXT` anhand der folgenden Parameter gestartet:

- a1
  - Der Parameter a1 startet die Funktion tripple_attack und übergibt alle weiteren Parameter an die Funktion.
- -m 1700
  - Gibt das Hashverfahren (1700) an, anhanddessen `hashcatXT` beziehungsweise `hashcat` Hashwerte erzeugen soll. Der Wert 1700 erzeugt Hashwerte anhand des Hashverfahrens SHA-512.
- -c -
  - Erzeugt einen vom Benutzer gewählten Zeichensatz, der im Fall dieses Tests nur einen Bindestrich „-“ enthält. Innerhalb einer Maske wird über den Parameter $1 auf einen so erzeugten Zeichensatz zugegriffen.
- −−mk ?1?d?d?d?d
  - Definiert die Maske, die an ein Passwort angehangen wird. $1 übergibt für das erste Zeichen den benutzerdefinierten Zeichensatz, der anhand des Parameters „-c“ definiert wurde. $d übergibt für die restlichen Stellen der Maske die Zahlen von 0 bis 9.
- −−hf examples/hashes/test_sha512.hash
  - Übergibt den Festplattenpfad einer Hashdatei.
- -d examples/dictionaries/top500names.dict
  - Übergibt den Festplattenpfad einer Wörterbuchdatei.
- -r examples/rules/szenario1.rule
  - Übergibt den Festplattenpfad einer Regeldatei.

Hierdurch ergibt sich der folgende (vollständige) Funktionsaufruf:

python3 hashcatXT.py a1 -m 1700 -c - −−mk ?1?d?d?d?d −−hf examples/hashes/test_sha512.hash -d examples/dictionaries/top500names.dict -r examples/rules/szenario1.rule

Abbildung 4.11 zeigt den erfolgreichen Aufruf der „tripple_attack“-Funktion über `hashcatXT`. Wie in der Abbildung zu sehen, benötigt `HashcatXT` sechs Sekunden um das gesuchte Passwort zu ermitteln.

Abbildung 4.11: Funktionstest der Funktion tripple_attack

4.7.2 Funktionstest 2

Für den Test der Funktion „multi_combinator“ wurden drei Wörterbücher erstellt, deren Wörter anhand der Funktion kombiniert werden sollen. Die Wörterbücher enthalten alle dieselben Wörter, haben allerdings abhängig von dem jeweiligen Wörterbuch die Nummer 1, 2 oder 3 am Ende eines jeden Wortes. Dementsprechend sind die Wörter des ersten Wörterbuchs:

- telefon1
- musik1
- uhr1
- tasse1

Ein Beispiel für eine mögliche Kombination der Wörter der Wörterbücher ist „telefon3tasse1uhr2“. Entsprechend der Wörter der Wörterbücher wurde der Hashwert eines Passworts in einer Hashdatei gespeichert. Für den Test der Funktion wurden dann die folgenden Parameter an `hashcatXT` übergeben:

- a2
  - Der Parameter a2 startet die Funktion multi_combinator und übergibt alle weiteren Parameter an die Funktion.

- -m 1700
  - Gibt das Hashverfahren (1700) an, anhand dessen `hashcatXT` beziehungsweise `hashcat` Hashwerte erzeugen soll. Der Wert 1700 erzeugt Hashwerte anhand des Hashverfahrens SHA-512.
- −−hf examples/hashes/test_sha512.hash
  - Übergibt den Festplattenpfad einer Hashdatei.
- -df examples/dictionaries/top500names.dict
  - Übergibt den Festplattenpfad zu einem Verzeichnis mit mehreren Wörterbüchern.

Der komplette Funktionsaufruf über `hashcatXT` lautet:

python3 hashcatXT.py a2 -m 1700 −−hf examples/hashes/combi_sha512.hash −−df examples/dictionaries/test

Abbildung 4.12 zeigt den erfolgreichen Aufruf der Funktion „multi_combinator“ anhand der beschriebenen Parameter.

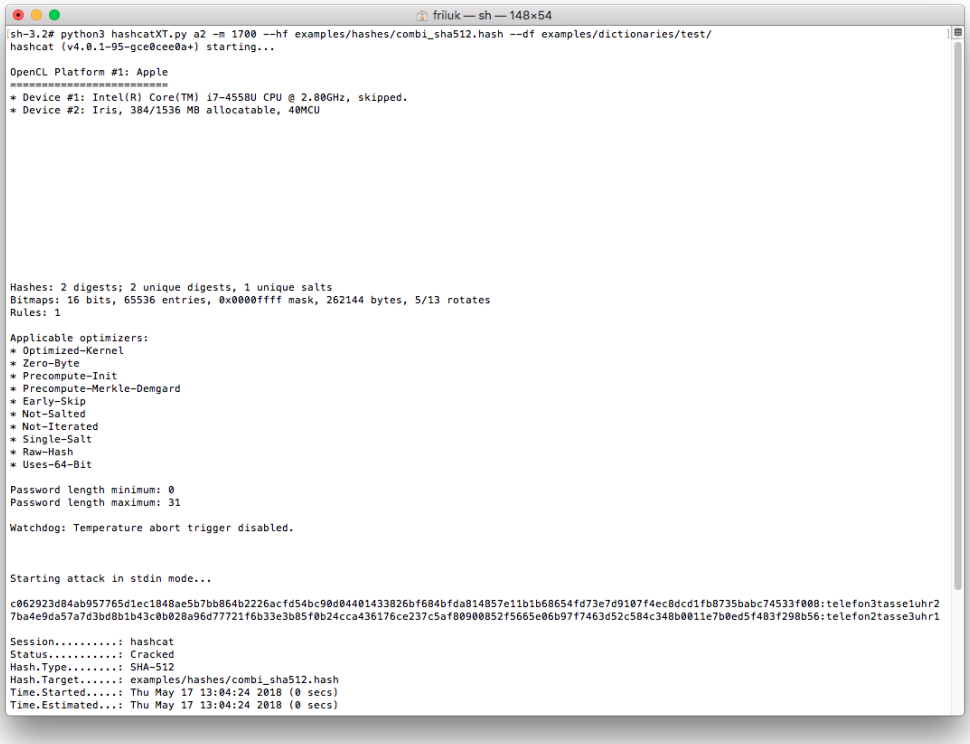

Abbildung 4.12: Starten eines Multi-Combinator-Angriffs

4.8 Anwendungsfälle

Anhang A.1 und A.2 beschreiben jeweils einen möglichen Anwendungsfall der Software `hashcatXT`. Jeder der beschriebenen Anwendungsfälle stellt in einer tabellarischen Schreibweise die essentiellen Schritte vom Start der Software bis zum gefunden Passwort dar.

Kapitel 5

Schlussfolgerung

Dieses letzte Kapitel der Arbeit bewertet in einem Fazit, die im Verlauf der Arbeit gewonnen Erkenntnisse und geht in einem Ausblick auf mögliche Nachfolgestudien ein. Zum Schluss fasst ein persönliches Schlusswort die, während der Entstehung dieser Arbeit, gewonnen Erfahrungen und Erkenntnisse zusammen.

5.1 Fazit

Zusammenfassend ist zu sagen, dass solange Passwörter von Menschen als Mittel zur Authentifikation genutzt werden, die Sicherheit dieser Passwörter durch den Faktor Mensch beeinflusst wird. Denn auch wenn bei der Erstellung eines Passworts kein direkter Einfluss durch einen Menschen genommen wird, besteht immer die Möglichkeit der Einflussnahme im Umgang und der Art und Weise wie ein Passwort verwahrt beziehungsweise gespeichert wird. Die Analyse der Szenarientests, aber auch andere Studien, haben gezeigt, dass viele der möglichen Fehler im Umgang mit Passwörtern, auf fehlendes Wissen der Nutzerseite zurückzuführen ist. Hier muss mehr Aufklärung zum Thema Passwortsicherheit geleistet werden. Passwörter und Verfahren zur Passworterstellung können noch so sicher sein, es wird immer Möglichkeiten geben, durch Präferenzen oder Gewohnheiten einzelner Individuen, die Regeln eines Passwortverfahrens zu ändern oder zu lockern. Der zu Szenario 2 durchgeführte Test, sowie verschiedene, veröffentlichte Studien zur Untersuchung häufig verwendeter Passwörter, wie die des HPI, zeigen, dass sehr viele Menschen noch immer zu schwache und vor allen Dingen zu kurze Passwörter verwenden. Dies zeigt, dass eine klare Abhängigkeit zwischen der Benutzbarkeit (engl. usability) eines Passworts und seiner Sicherheit existiert. Wenn die Ansprüche an die Passwortsicherheit gegenüber den Ansprüchen an die Benutzbarkeit eines Passwortverfahrens überwiegen, wird ein solches Passwortverfahren entweder keine bis wenig Anwendung finden oder seine Sicherheit wird durch das Verhalten Einzelner gemindert werden.
Die in der Einleitung dieser Arbeit infrage gestellten Alternativen, wie zum Beispiel moderne „Password Strength Meter“, die den Nutzer bei Erstellung guter und vor allem langlebiger Passwörter unterstützen, finden noch zu wenig Anwendung. Passwortverfahren, wie Diceware, versuchen hier einen Mittelweg zu gehen, indem dem Nutzer keine unnötige Komplexität bei der Anzahl der verwendbaren Zeichen „aufgezwungen“ wird, stattdessen gilt „Quantität vor Qualität“, da durch das Generieren sehr langer Passwörter (Passphrases), die gut durch Nutzer in Erinnerung behalten werden können, gleichzeitig eine entsprechende Sicherheit erreicht wird. Einen anderen Ansatz hierzu liefert das „Key-Stretching-Verfahren“ bei passwortbasierten Hashfunktionen (siehe Kapitel 2.4.2). Hierbei wird die Entropie eines Passworts, unabhängig von der Stärke, des durch den Nutzer gewählten Passworts, auf ein ausreichend sicheres Maß angehoben. Dies ist als guter Ansatz anzusehen, da versucht wird

den Faktor Mensch so weit wie möglich zu umgehen und die Verantwortung für ein sicheres und starkes Passwort nicht länger dem Nutzer allein zu überlassen. Im späteren Verlauf der Arbeit haben die Ergebnisse der Szenarientests gezeigt, dass es durchaus möglich ist, die Effizienz von passwortbasierten Angriffen unter Einbeziehung des Faktors Mensch zu steigern. Die hierbei festgestellten `hashcat`-Schwächen haben gezeigt, dass die Software neben den vielen Möglichkeiten, die sie bietet um Passwörter jeder Art anzugreifen, einerseits noch nicht ihr ganzes Potential nutzt, um beispielsweise die vorhandenen Funktionen effizienter miteinander zu kombinieren, um neue und individuelle Angriffstechniken zu ermöglichen und andererseits noch nicht die Möglichkeit bietet, Passphrases mit mehr als zwei Wörtern komfortabel anzugreifen.
Zusätzlich haben die Szenarientests, aber auch diverse andere Tests mit `hashcat`, gezeigt, dass es ohne ausreichende Erfahrung zum Teil schwierig sein kann, die für ein anzugreifendes Passwort geeignetsten Angriffsfunktionen mit den hierfür besten Optionen zu wählen. Das Resultat dieser Überlegungen und der Tests ist das an die Tests anschließende `hashcat`-Empfehlungsdiagramm. Es soll anderen Nutzern einen einfacheren Einstieg in `hashcat` und die Thematik des „Passwort crackens“ ermöglichen.
Die, auf Basis der aufgedeckten `hashcat`-Schwächen entwickelte Erweiterung von `hashcat`, hat gezeigt, dass es möglich ist, die Funktionalität von `hashcat` mit einfachen Mitteln zu erweitern und zu verbessern. Dies gibt programmierkundigen Nutzern die Möglichkeit, mit wenigen Mitteln eigene Angriffsfunktionen für `hashcat` zu entwickeln, ohne den eigentlichen `hashcat`-Programmcode verstehen zu müssen.
Abschließend ist zu sagen, dass weiterhin neue Ansätze für die Passwortauthentifikation anhand von Studien evaluiert werden müssen. Es sollte ein Weg gefunden werden, Menschen, schon möglichst früh, über den Wert ihrer eigenen Daten und die hieraus entstehende Notwendigkeit eines sicheren Passworts, aufzuklären, um sie somit frühzeitig für die Passwortsicherheit zu sensibilisieren und die Gefahr des Faktors Mensch, für die Passwortsicherheit, weiter zu reduzieren.

5.2 Ausblick

Man kann davon ausgehen, dass auch in Zukunft Passwörter aufgrund ihrer einfachen Umsetzbarkeit und der durch Nutzer verhältnismäßig einfachen Anwendung eine der meistverwendeten Authentifikationsmethoden sein werden. Deshalb sollten weitere Studien und Forschungen zum menschlichen Verhalten bei der Passworterstellung und der Art und Weise wie mit einem Passwort im Alltag umgegangen wird, durchgeführt werden. Hierbei sind verschiedene interessante Ansätze denkbar, wobei vor allem der psychologische Aspekt bei der Passwortwahl interessant ist. Ein Beispiel für eine solche Studie wäre eine anonyme Nutzergruppenbefragung, um anhand der Ergebnisse den Faktor Mensch besser im Vorhinein kalkulieren zu können. Bei dieser Befragung könnte beispielhaft die deutsche Bevölkerung repräsentativ abgebildet werden, sodass evaluiert werden kann, ob Faktoren wie zum Beispiel der Bildungsgrad, das Alter des Passworterstellers oder die regionale Herkunft einen Einfluss auf die Passwortkomplexität beziehungsweise Passwortsicherheit haben. Dies wäre vor allem für größere Projekte, bei denen die Nutzerauthentifikation eine wichtige Rolle spielt, sehr interessant. Denkbar wäre auch eine Vergleichsstudie, bei der ebenfalls anhand von repräsentativen Befragungen, Erkenntnisse darüber gewonnen werden können, ob die Probleme der Passwortsicherheit an mangelndem Wissen auf Nutzerseite liegen oder ob das Wissen sowohl bezüglich möglicher drohender Gefahren wie zum Beispiel Passwortdiebstähle (damit verbunden die Weitergabe personenbezogener Daten an unbefugte Dritte und den daraus resultierenden möglichen Konsequenzen) als auch der möglichen Passworterstellungsmethoden zwar vorhanden ist, aber die Priorität gegenüber den eigenen per-

sonenbezogenen Daten zu gering ist. Ebenfalls von Interesse wäre es zu erfahren, wie viel Aufwand ein Nutzer bereit ist auf sich zunehmen um die persönlichen Daten ausreichend zu schützen, denn der eigentliche Aufwand eines sicheren Passworts liegt oft nicht in der Erstellung sondern in dessen Verwendung.
Da im Zuge dieser Arbeit nur zwei Funktionen für `hashcatXT` entwickelt wurden, sollte die Funktionalität von `hashcatXT` weiter ausgebaut werden. Eine denkbare Option wäre die Möglichkeit angreifbare Hashwerte für Testzwecke direkt über das Programm generieren zu können, um so auf externe Tools und oder Webseiten verzichten zu können. Darüber hinaus sollte eine Kompatibilität zum Betriebssystem Microsoft Windows entwickelt werden.
Für die Zukunft wäre eine statistische Erhebung über die Häufigkeit von Passwortangriffsverfahren wünschenswert, um so evaluieren zu können wie häufig, welcher Typ von Angriffsart im Durchschnitt durchgeführt wird. Dies würde zum einen Rückschlüsse darüber geben, ob es Angriffsarten gibt, die nicht mehr benutzt werden, weil damit zu rechnen ist, dass aktuelle Systeme gegen einen solchen Angriff gewappnet sind und zum anderen würde es einen Überblick über die Gegenseite, nämlich die Anzahl der Systeme, geben, die nicht nur mit aktuellen, sondern auch veralteten Methoden angegriffen werden können. Die Ergebnisse könnten dann in einer Bewertungstabelle abgebildet werden.

5.3 Persönliches Schlusswort und Danksagung

Da ich während meines Studiums keine umfangreichen wissenschaftlichen Arbeiten schreiben musste, hatte ich sehr großen Respekt vor der Herausforderung diese Bachelorarbeit zu verfassen. Besonders spannend hierbei war der Umstand, dass sich die Arbeit Stück für Stück entwickelt hat, da während meiner Literaturrecherche der Umfang und die Ausrichtung der Arbeit noch nicht in Gänze klar definiert waren. Diese „klare“ Definition entstand im Grunde erst, als die ersten eigenen Tests mit `hashcat` abgeschlossen waren und ich Anforderungen an die Software gestellt habe, die durch `hashcat` nicht erfüllt werden konnten. Die anschließende Recherche und die Einarbeitung in Python zur Entwicklung von `hashcatXT` haben mir mehr Freude bereitet als ursprünglich angenommen, da ich im Laufe meines Studiums meinen Schwerpunkt nie in der Programmierung gesehen habe. Bei der weiteren Literaturrecherche zu den Grundlagenkapiteln war ich sehr überrascht, wie wenig deutschsprachige Literatur in Form von Büchern oder wissenschaftlichen Artikeln, vor allem in Bezug auf ältere Themen, zu finden war.
Auch sehr anspruchsvoll und herausfordernd war das Schreiben dieser Arbeit in LaTeX[1]. Ich habe zuvor noch nie mit Latex gearbeitet, weshalb zuerst eine Menge Zeit in die Einarbeitung geflossen ist. Als allerdings einmal ein Grundverständnis für den Aufbau und das Konzept von Latex vorhanden war, war ich sehr von den Möglichkeiten, wie zum Beispiel Funktionen zum Zeichnen von Diagrammen, beeindruckt. Letztendlich ist durch die Hilfe meines betreuenden Professors Dr.-Ing. Holger Schmidt, aber ebenso durch die Unterstützung meiner Partnerin und meiner Familie, eine sehr umfangreiche und aussagekräftige Bachelorarbeit entstanden. Hierfür möchte ich mich bei allen Personen herzlichst bedanken.

[1] https://www.latex-project.org, abgerufen am 17.05.2018

Anhang A

Anwendungsfälle

A.1 Anwendungsfall 1

Name:	Anhängen von Zahlen an Passwörter
Primärakteur:	Angreifer
Umfang:	Hacker, IT-Security-Experte
Ebene:	Ziel auf Anwenderebene
Interessen:	`Hacker` - ermittelt Passwörter anderer für illegale Aktivitäten `IT-Security-Experte` - Testet im Auftrag die Passwortsicherheit eines Systems
Vorbedingung:	Der Angreifer hat `hashcatXT` bereits installiert und hat ein Shell-Fenster zum Verzeichnis von `hashcatXT` geöffnet.
Invariante:	Der Angreifer weiß, dass das verwendete Passwort aus einem Wort sowie einer angehängten Zahl besteht. Es ist außerdem bekannt das SHA-512-Verfahren zum Generieren des Hashwerts benutzt wurde.
Nachbedingungen:	`HashcatXT` ermittelt das gesuchte Passwort.

Standardablauf:	1. Der Angreifer startet `hashcatXT` und übergibt Parameter für: - den Angriffsmodus - den Parameter für das Hashverfahren - einen Festplattenpfad zu einer Hashdatei die den/die Hashwert(e) enthält - eine Maske die die Anzahl der Stellen der an das Passwort anzuhängenden Zahl definiert. - einen Festplattenpfad für ein Wörterbuch - einen Festplattenpfad zu einer Regeldatei, anhand der die Passwörter des Wörterbuchs in Ihrer Schreibweise bzw. ihrem Aufbau modifiziert werden können. 2. `HashcatXT` übergibt die Parameter für das Wörterbuch und der Maske an `hashcat`. 3. `Hashcat` hängt an jedes Passwort des Wörterbuchs Zahlen entsprechend der Größe der Maske an. 4. `Hashcat` übergibt Passwortkandidaten an `hashcatXT`. 5. `HashcatXT` übergibt die Passwortkandidaten, zusammen mit der Pfadangabe der Hashdatei und der Regeldatei an `hashcat`. 6. `Hashcat` modifiziert die Passwortkandidaten entsprechend der in der Regeldatei definierten Regeln. 7. `Hashcat` erzeugt für jeden modifizierten Passwortkandidaten einen SHA512 Hashwert. 8. `Hashcat` vergleicht alle erzeugten Hashwerte mit dem Hashwert der übergebenen Hashdatei. 9. `Hashcat` gibt fortwährend den Status der Hashwertvergleiche an `hashcatXT` zurück. 10. `HashcatXT` gibt den von `hashcat` übermittelten Status auf dem Bildschirm des Nutzers aus. 11. Einer der durch `hashcat` erzeugten Hashwerte entspricht dem übergebenen Hashwert. 12. `Hashcat` übermittelt das Passwort des übereinstimmenden Hashwerts an `hashcatXT` 13. `HashcatXT` zeigt das Ergebnis, das ermittelte Passwort, auf dem Bildschirm des Nutzers an. 14. Sofern mehr als ein Hashwert in der übergebenen Hashdatei gespeichert war, beginnt der Prozess, mit dem nächsten Hashwert wieder bei Schritt 6. Wenn alle Hashwerte gefunden wurden, wird `hashcatXT` beendet.

Tabelle A.1: Anwendungsfall 1

A.2 Anwendungsfall 2

Name:	Ermitteln eines Passwortsatzes
Primärakteur:	Angreifer
Umfang:	Hacker, IT-Security Experte
Ebene:	Ziel auf Anwenderebene

Interessen:	`Hacker` - ermittelt Passwörter anderer für Illegale Aktivitäten `IT-Security Experte` - Testet im Auftrag die Passwortsicherheit eines Systems
Vorbedingung:	Der Angreifer hat `hashcatXT` bereits installiert und hat ein Shell Fenster zum Verzeichnis von `hashcatXT` geöffnet.
Invariante:	Der Angreifer weiß, dass die Zielperson Passwortsätze (passphrases), die aus vier Worten bestehen, zum Schutz seiner/Ihrer Daten verwendet. Weiterhin weiß er, dass das Scrypt Verfahren verwendet wurde
Nachbedingungen:	`HashcatXT` ermittelt den gesuchten Passwortsatz / die gesuchten Passwortsätze.
Standardablauf:	1. Der Angreifer startet `hashcatXT` und übergibt Parameter für: - den Angriffsmodus - den Parameter für das Hashverfahren - einen Festplattenpfad zu einer Hashdatei die den/die Hashwert(e) enthält - einen Festplattenpfad zu den Wörterbüchern 2. `HashcatXT` kombiniert die Worte der übergebenen Wörterbücher zu einem Passwortsatz. 3. `HashcatXT` übergibt nacheinander jeden möglichen Passwortkandidaten zusammen mit dem Hashwert der übergebenen Hashdatei und dem verwendeten Hashverfahren an `hashcat`. 4. `Hashcat` erzeugt für jeden übergebenen Passwortkandidaten, den entsprechenden Hashwert (anhand des übergebenen Hashverfahrens). 5. `Hashcat` vergleicht jeden erzeugten mit dem übergebenen Hashwert. 6. `Hashcat` gibt fortwährend den Status der Hashwertvergleiche an `hashcatXT` zurück. 7. `HashcatXT` gibt den von `hashcat` übermittelten Status auf dem Bildschirm des Nutzers aus. 8. Einer der durch `hashcat` erzeugten Hashwerte entspricht dem übergebenen Hashwert. 9. `Hashcat` übermittelt das Passwort / den Passwortsatz des übereinstimmenden Hashwerts an `hashcatXT`. 10. `HashcatXT` zeigt das Ergebnis, das ermittelte Passwort, auf dem Bildschirm des Nutzers an. 11. Sofern mehr als ein Hashwert in der übergebenen Hashdatei gespeichert war, beginnt der Prozess mit dem nächsten Hashwert wieder bei Schritt 3. Wenn alle Hashwerte gefunden wurden, wird `hashcatXT` beendet.

Tabelle A.2: Anwendungsfall 2

Anhang B

Hashcat

B.1 Unterstützte Algorithmen

```
- [ Hash modes ] -

   # | Name                                               |
======+==================================================+
  900 | MD4                                              |
    0 | MD5                                              |
 5100 | Half MD5                                         |
  100 | SHA1                                             |
 1300 | SHA-224                                          |
 1400 | SHA-256                                          |
10800 | SHA-384                                          |
 1700 | SHA-512                                          |
 5000 | SHA-3 (Keccak)                                   |
  600 | BLAKE2b-512                                      |
10100 | SipHash                                          |
 6000 | RIPEMD-160                                       |
 6100 | Whirlpool                                        |
 6900 | GOST R 34.11-94                                  |
11700 | GOST R 34.11-2012 (Streebog) 256-bit             |
11800 | GOST R 34.11-2012 (Streebog) 512-bit             |
   10 | md5($pass.$salt)                                 |
   20 | md5($salt.$pass)                                 |
   30 | md5(utf16le($pass).$salt)                        |
   40 | md5($salt.utf16le($pass))                        |
 3800 | md5($salt.$pass.$salt)                           |
 3710 | md5($salt.md5($pass))                            |
 4010 | md5($salt.md5($salt.$pass))                      |
 4110 | md5($salt.md5($pass.$salt))                      |
 2600 | md5(md5($pass))                                  |
 3910 | md5(md5($pass).md5($salt))                       |
 4300 | md5(strtoupper(md5($pass)))                      |
 4400 | md5(sha1($pass))                                 |
  110 | sha1($pass.$salt)                                |
```

```
   120 | sha1($salt.$pass)                                |
   130 | sha1(utf16le($pass).$salt)                       |
   140 | sha1($salt.utf16le($pass))                       |
  4500 | sha1(sha1($pass))                                |
  4520 | sha1($salt.sha1($pass))                          |
  4700 | sha1(md5($pass))                                 |
  4900 | sha1($salt.$pass.$salt)                          |
 14400 | sha1(CX)                                         |
  1410 | sha256($pass.$salt)                              |
  1420 | sha256($salt.$pass)                              |
  1430 | sha256(utf16le($pass).$salt)                     |
  1440 | sha256($salt.utf16le($pass))                     |
  1710 | sha512($pass.$salt)                              |
  1720 | sha512($salt.$pass)                              |
  1730 | sha512(utf16le($pass).$salt)                     |
  1740 | sha512($salt.utf16le($pass))                     |
    50 | HMAC-MD5 (key = $pass)                           |
    60 | HMAC-MD5 (key = $salt)                           |
   150 | HMAC-SHA1 (key = $pass)                          |
   160 | HMAC-SHA1 (key = $salt)                          |
  1450 | HMAC-SHA256 (key = $pass)                        |
  1460 | HMAC-SHA256 (key = $salt)                        |
  1750 | HMAC-SHA512 (key = $pass)                        |
  1760 | HMAC-SHA512 (key = $salt)                        |
 14000 | DES (PT = $salt, key = $pass)                    |
 14100 | 3DES (PT = $salt, key = $pass)                   |
 14900 | Skip32 (PT = $salt, key = $pass)                 |
 15400 | ChaCha20                                         |
   400 | phpass                                           |
  8900 | scrypt                                           |
 11900 | PBKDF2-HMAC-MD5                                  |
 12000 | PBKDF2-HMAC-SHA1                                 |
 10900 | PBKDF2-HMAC-SHA256                               |
 12100 | PBKDF2-HMAC-SHA512                               |
    23 | Skype                                            |
  2500 | WPA/WPA2                                         |
  2501 | WPA/WPA2 PMK                                     |
  4800 | iSCSI CHAP authentication, MD5(CHAP)             |
  5300 | IKE-PSK MD5                                      |
  5400 | IKE-PSK SHA1                                     |
  5500 | NetNTLMv1                                        |
  5500 | NetNTLMv1+ESS                                    |
  5600 | NetNTLMv2                                        |
  7300 | IPMI2 RAKP HMAC-SHA1                             |
  7500 | Kerberos 5 AS-REQ Pre-Auth etype 23              |
  8300 | DNSSEC (NSEC3)                                   |
 10200 | CRAM-MD5                                         |
 11100 | PostgreSQL CRAM (MD5)                            |
 11200 | MySQL CRAM (SHA1)                                |
 11400 | SIP digest authentication (MD5)                  |
 13100 | Kerberos 5 TGS-REP etype 23                      |
```

```
 16100 | TACACS+                                        |
 16500 | JWT (JSON Web Token)                           |
   121 | SMF (Simple Machines Forum) > v1.1             |
   400 | phpBB3 (MD5)                                   |
  2611 | vBulletin < v3.8.5                             |
  2711 | vBulletin >= v3.8.5                            |
  2811 | MyBB 1.2+                                      |
  2811 | IPB2+ (Invision Power Board)                   |
  8400 | WBB3 (Woltlab Burning Board)                   |
    11 | Joomla < 2.5.18                                |
   400 | Joomla >= 2.5.18 (MD5)                         |
   400 | WordPress (MD5)                                |
  2612 | PHPS                                           |
  7900 | Drupal7                                        |
    21 | osCommerce                                     |
    21 | xt:Commerce                                    |
 11000 | PrestaShop                                     |
   124 | Django (SHA-1)                                 |
 10000 | Django (PBKDF2-SHA256)                         |
 16000 | Tripcode                                       |
  3711 | MediaWiki B type                               |
 13900 | OpenCart                                       |
  4521 | Redmine                                        |
  4522 | PunBB                                          |
 12001 | Atlassian (PBKDF2-HMAC-SHA1)                   |
    12 | PostgreSQL                                     |
   131 | MSSQL (2000)                                   |
   132 | MSSQL (2005)                                   |
  1731 | MSSQL (2012, 2014)                             |
   200 | MySQL323                                       |
   300 | MySQL4.1/MySQL5                                |
  3100 | Oracle H: Type (Oracle 7+)                     |
   112 | Oracle S: Type (Oracle 11+)                    |
 12300 | Oracle T: Type (Oracle 12+)                    |
  8000 | Sybase ASE                                     |
   141 | Episerver 6.x < .NET 4                         |
  1441 | Episerver 6.x >= .NET 4                        |
  1600 | Apache $apr1$ MD5, md5apr1, MD5 (APR)          |
 12600 | ColdFusion 10+                                 |
  1421 | hMailServer                                    |
   101 | nsldap, SHA-1(Base64), Netscape LDAP SHA       |
   111 | nsldaps, SSHA-1(Base64), Netscape LDAP SSHA    |
  1411 | SSHA-256(Base64), LDAP {SSHA256}               |
  1711 | SSHA-512(Base64), LDAP {SSHA512}               |
 16400 | CRAM-MD5 Dovecot                               |
 15000 | FileZilla Server >= 0.9.55                     |
 11500 | CRC32                                          |
  3000 | LM                                             |
  1000 | NTLM                                           |
  1100 | Domain Cached Credentials (DCC), MS Cache      |
  2100 | Domain Cached Credentials 2 (DCC2), MS Cache 2 |
```

```
 15300 | DPAPI masterkey file v1                     |
 15900 | DPAPI masterkey file v2                     |
 12800 | MS-AzureSync PBKDF2-HMAC-SHA256             |
  1500 | descrypt, DES (Unix), Traditional DES       |
 12400 | BSDi Crypt, Extended DES                    |
   500 | md5crypt, MD5 (Unix), Cisco-IOS $1$ (MD5)   |
  3200 | bcrypt $2*$, Blowfish (Unix)                |
  7400 | sha256crypt $5$, SHA256 (Unix)              |
  1800 | sha512crypt $6$, SHA512 (Unix)              |
   122 | macOS v10.4, MacOS v10.5, MacOS v10.6       |
  1722 | macOS v10.7                                 |
  7100 | macOS v10.8+ (PBKDF2-SHA512)                |
  6300 | AIX {smd5}                                  |
  6700 | AIX {ssha1}                                 |
  6400 | AIX {ssha256}                               |
  6500 | AIX {ssha512}                               |
  2400 | Cisco-PIX MD5                               |
  2410 | Cisco-ASA MD5                               |
   500 | Cisco-IOS $1$ (MD5)                         |
  5700 | Cisco-IOS type 4 (SHA256)                   |
  9200 | Cisco-IOS $8$ (PBKDF2-SHA256)               |
  9300 | Cisco-IOS $9$ (scrypt)                      |
    22 | Juniper NetScreen/SSG (ScreenOS)            |
   501 | Juniper IVE                                 |
 15100 | Juniper/NetBSD sha1crypt                    |
  7000 | FortiGate (FortiOS)                         |
  5800 | Samsung Android Password/PIN                |
 13800 | Windows Phone 8+ PIN/password               |
  8100 | Citrix NetScaler                            |
  8500 | RACF                                        |
  7200 | GRUB 2                                      |
  9900 | Radmin2                                     |
   125 | ArubaOS                                     |
  7700 | SAP CODVN B (BCODE)                         |
  7800 | SAP CODVN F/G (PASSCODE)                    |
 10300 | SAP CODVN H (PWDSALTEDHASH) iSSHA-1         |
  8600 | Lotus Notes/Domino 5                        |
  8700 | Lotus Notes/Domino 6                        |
  9100 | Lotus Notes/Domino 8                        |
   133 | PeopleSoft                                  |
 13500 | PeopleSoft PS_TOKEN                         |
 11600 | 7-Zip                                       |
 12500 | RAR3-hp                                     |
 13000 | RAR5                                        |
 13200 | AxCrypt                                     |
 13300 | AxCrypt in-memory SHA1                      |
 13600 | WinZip                                      |
 14700 | iTunes backup < 10.0                        |
 14800 | iTunes backup >= 10.0                       |
  62XY | TrueCrypt                                   |
    X  | 1 = PBKDF2-HMAC-RIPEMD160                   |
```

```
   X  | 2 = PBKDF2-HMAC-SHA512                       |
   X  | 3 = PBKDF2-HMAC-Whirlpool                    |
   X  | 4 = PBKDF2-HMAC-RIPEMD160 + boot-mode        |
    Y | 1 = XTS 512 bit pure AES                     |
    Y | 1 = XTS 512 bit pure Serpent                 |
    Y | 1 = XTS 512 bit pure Twofish                 |
    Y | 2 = XTS 1024 bit pure AES                    |
    Y | 2 = XTS 1024 bit pure Serpent                |
    Y | 2 = XTS 1024 bit pure Twofish                |
    Y | 2 = XTS 1024 bit cascaded AES-Twofish        |
    Y | 2 = XTS 1024 bit cascaded Serpent-AES        |
    Y | 2 = XTS 1024 bit cascaded Twofish-Serpent    |
    Y | 3 = XTS 1536 bit all                         |
 8800 | Android FDE <= 4.3                           |
12900 | Android FDE (Samsung DEK)                    |
12200 | eCryptfs                                     |
137XY | VeraCrypt                                    |
   X  | 1 = PBKDF2-HMAC-RIPEMD160                    |
   X  | 2 = PBKDF2-HMAC-SHA512                       |
   X  | 3 = PBKDF2-HMAC-Whirlpool                    |
   X  | 4 = PBKDF2-HMAC-RIPEMD160 + boot-mode        |
   X  | 5 = PBKDF2-HMAC-SHA256                       |
   X  | 6 = PBKDF2-HMAC-SHA256 + boot-mode           |
    Y | 1 = XTS 512 bit pure AES                     |
    Y | 1 = XTS 512 bit pure Serpent                 |
    Y | 1 = XTS 512 bit pure Twofish                 |
    Y | 2 = XTS 1024 bit pure AES                    |
    Y | 2 = XTS 1024 bit pure Serpent                |
    Y | 2 = XTS 1024 bit pure Twofish                |
    Y | 2 = XTS 1024 bit cascaded AES-Twofish        |
    Y | 2 = XTS 1024 bit cascaded Serpent-AES        |
    Y | 2 = XTS 1024 bit cascaded Twofish-Serpent    |
    Y | 3 = XTS 1536 bit all                         |
14600 | LUKS                                         |
 9700 | MS Office <= 2003 $0/$1, MD5 + RC4           |
 9710 | MS Office <= 2003 $0/$1, MD5 + RC4, collider #1 |
 9720 | MS Office <= 2003 $0/$1, MD5 + RC4, collider #2 |
 9800 | MS Office <= 2003 $3/$4, SHA1 + RC4          |
 9810 | MS Office <= 2003 $3, SHA1 + RC4, collider #1 |
 9820 | MS Office <= 2003 $3, SHA1 + RC4, collider #2 |
 9400 | MS Office 2007                               |
 9500 | MS Office 2010                               |
 9600 | MS Office 2013                               |
10400 | PDF 1.1 - 1.3 (Acrobat 2 - 4)                |
10410 | PDF 1.1 - 1.3 (Acrobat 2 - 4), collider #1   |
10420 | PDF 1.1 - 1.3 (Acrobat 2 - 4), collider #2   |
10500 | PDF 1.4 - 1.6 (Acrobat 5 - 8)                |
10600 | PDF 1.7 Level 3 (Acrobat 9)                  |
10700 | PDF 1.7 Level 8 (Acrobat 10 - 11)            |
16200 | Apple Secure Notes                           |
 9000 | Password Safe v2                             |
```

```
  5200 | Password Safe v3                               |
  6800 | LastPass + LastPass sniffed                    |
  6600 | 1Password, agilekeychain                       |
  8200 | 1Password, cloudkeychain                       |
 11300 | Bitcoin/Litecoin wallet.dat                    |
 12700 | Blockchain, My Wallet                          |
 15200 | Blockchain, My Wallet, V2                      |
 13400 | KeePass 1 (AES/Twofish) and KeePass 2 (AES)    |
 15500 | JKS Java Key Store Private Keys (SHA1)         |
 15600 | Ethereum Wallet, PBKDF2-HMAC-SHA256            |
 15700 | Ethereum Wallet, SCRYPT                        |
 16300 | Ethereum Pre-Sale Wallet, PBKDF2-HMAC-SHA256   |
 99999 | Plaintext                                      |
```

Listing B.1: Tabelle aller durch hashcat unterstützten Hashverfahren.
(Quelle: Das Programm hashcat)

B.2 Funktionen für regelbasierte Angriffe

Die in Abbildung B.1 dargestellte Tabelle, listet alle Funktionen auf, die zu hundert Prozent kompatibel sind mit „John the Ripper“ und „PasswordsPro“.

Name	Function	Description	Example Rule	Input Word	Output Word	Note
Nothing	:	do nothing	:	p@ss-W0rd	p@ssW0rd	
Lowercase	l	Lowercase all letters	l	p@ss-W0rd	p@ssw0rd	
Uppercase	u	Uppercase all letters	u	p@ss-W0rd	P@SSW0RD	
Capitalize	c	Capitalize the first letter and lower the rest	c	p@ss-W0rd	P@ssw0rd	
Invert Capitalize	C	Lowercase first found character, uppercase the rest	C	p@ss-W0rd	p@SSW0RD	
Toggle Case	t	Toggle the case of all characters in word.	t	p@ss-W0rd	P@SSw0RD	
Toggle @	TN	Toggle the case of characters at position N	T3	p@ss-W0rd	p@sSW0rd	*
Reverse	r	Reverse the entire word	r	p@ss-W0rd	dr0Wss@p	
Duplicate	d	Duplicate entire word	d	p@ss-W0rd	p@ssW0rdp@ssW0rd	
Duplicate N	pN	Append duplicated word N times	p2	p@ss-W0rd	p@ssW0rdp@ssW0rdp@ssW0rd	
Reflect	f	Duplicate word reversed	f	p@ss-W0rd	p@ssW0rddr0Wss@p	
Rotate Left	{	Rotates the word left.	{	p@ss-W0rd	@ssW0rdp	
Rotate Right	}	Rotates the word right	}	p@ss-W0rd	dp@ssW0r	
Append Character	$X	Append character X to end	$1	p@ss-W0rd	p@ssW0rd1	
Prepend Character	^X	Prepend character X to front	^1	p@ss-W0rd	1p@ssW0rd	
Truncate left	[	Deletes first character	[	p@ss-W0rd	@ssW0rd	
Trucate right	]	Deletes last character	]	p@ss-W0rd	p@assW0r	
Delete @ N	DN	Deletes character at position N	D3	p@ss-W0rd	p@sW0rd	*
Extract range	xNM	Extracts M characters, starting at position N	x04	p@ss-W0rd	p@ss	* #
Omit range	ONM	Deletes M characters, starting at position N	O12	p@ss-W0rd	psW0rd	*
Insert @ N	iNX	Inserts character X at position N	i4!	p@ss-W0rd	p@ss!W0rd	*
Overwrite @ N	oNX	Overwrites character at position N with X	o3$	p@ss-W0rd	p@s$W0rd	*
Truncate @ N	'N	Truncate word at position N	'6	p@ss-W0rd	p@ssW0	*
Replace	sXY	Replace all instances of X with Y	ss$	p@ss-W0rd	p@$$W0rd	
Purge	@X	Purge all instances of X	@s	p@ss-W0rd	p@W0rd	+
Duplicate first N	zN	Duplicates first character N times	z2	p@ss-W0rd	ppp@ssW0rd	
Duplicate last N	ZN	Duplicates last character N times	Z2	p@ss-W0rd	p@ssW0rddd	
Duplicate all	q	Duplicate every character	q	p@ss-W0rd	pp@@ssssWW00rrdd	
Extract memory	XNMI	Insert substring of length M starting from position N of word saved to memory at position I	lMX428	p@ss-W0rd	p@ssw0rdw0	+
Append memory	4	Append the word saved to memory to current word	uMl4	p@ss-W0rd	p@ssw0rdP@SSW0RD	+
Prepend memory	6	Prepend the word saved to memory to current word	rMr6	p@ss-W0rd	dr0Wss@pp@ssW0rd	+
Memorize	M	Memorize current word	lMuX084	p@ss-W0rd	P@SSp@ssw0rdW0RD	+

Abbildung B.1: Rule-Based-Attack Tabelle mit Funktionen
(Quelle: https://hashcat.net/wiki/doku.php?id=rule_based_attack)

Abbildung B.2 zeigt eine Tabelle, die Funktionen zum Abweisen von Passwörtern beschreibt. Diese Funktionen funktionieren nur für die Parameter „-j“ und „-k“ des Hybridangriffs, nicht aber als normale Regeln innerhalb einer Regeldatei.

Name	Function	Description	Example Rule	Note
Reject less	<N	Reject plains if their length is greater than N	<G	*
Reject greater	>N	Reject plains if their length is less or equal to N	>8	*
Reject equal	_N	Reject plains of length not equal to N	_7	*
Reject contain	!X	Reject plains which contain char X	!z	
Reject not contain	/X	Reject plains which do not contain char X	/e	
Reject equal first	(X	Reject plains which do not start with X	(h	
Reject equal last	)X	Reject plains which do not end with X	)t	
Reject equal at	=NX	Reject plains which do not have char X at position N	=1a	*
Reject contains	%NX	Reject plains which contain char X less than N times	%2a	*
Reject contains	Q	Reject plains where the memory saved matches current word	rMrQ	e.g. for palindrome

Abbildung B.2: Rule-Based-Attack Tabelle zum Abweisen von Passwörtern
(Quelle: https://hashcat.net/wiki/doku.php?id=rule_based_attack)

Alle Funktionen der Tabelle in Abbildung B.3, zeigen ausschließlich Funktionen, die nur in `hashcat` verfügbar sind.

Name	Function	Description	Example Rule	Input Word	Output Word	Note
Swap front	k	Swaps first two characters	k	p@ssW0rd	@pssW0rd	
Swap back	K	Swaps last two characters	K	p@ssW0rd	p@ssW0dr	
Swap @ N	*NM	Swaps character at position N with character at position M	*34	p@ssW0rd	p@sWs0rd	*
Bitwise shift left	LN	Bitwise shift left character @ N	L2	p@ssW0rd	p@æsW0rd	*
Bitwise shift right	RN	Bitwise shift right character @ N	R2	p@ssW0rd	p@9sW0rd	*
Ascii increment	+N	Increment character @ N by 1 ascii value	+2	p@ssW0rd	p@tsW0rd	*
Ascii decrement	-N	Decrement character @ N by 1 ascii value	-1	p@ssW0rd	p?ssW0rd	*
Replace N + 1	.N	Replaces character @ N with value at @ N plus 1	.1	p@ssW0rd	psssW0rd	*
Replace N - 1	,N	Replaces character @ N with value at @ N minus 1	,1	p@ssW0rd	ppssW0rd	*
Duplicate block front	yN	Duplicates first N characters	y2	p@ssW0rd	p@p@ss-W0rd	*
Duplicate block back	YN	Duplicates last N characters	Y2	p@ssW0rd	p@ssW0r-drd	*
Title	E	Lower case the whole line, then upper case the first letter and every letter after a space	E	p@ssW0rd w0rld	P@ssw0rd W0rld	+
Title w/separator	eX	Lower case the whole line, then upper case the first letter and every letter after a custom separator character	e-	p@ssW0rd-w0rld	P@ssw0rd-W0rld	+

Abbildung B.3: Rule-Based-Attack Tabelle mit Funktionen die nur mit `hashcat` kompatibel sind
(Quelle: https://hashcat.net/wiki/doku.php?id=rule_based_attack)

Anhang C

HashcatXT Programmcode

Die nachfolgenden Code-Listings zeigen den Programmcode der einzelnen Module des Programms `hashcatXT`.

C.1 Modul hashcatXT.py

Listing C.1: Modul: hashcatXT.py

```
1   #!/usr/bin/python
2 """hashcatXT.
3
4 Usage:
5   hashcatXT.py a1 -m <hash-type> [-c <custom-charset>] --mk <mask> --hf
        <hash-path> -d <dictionary-path> -r <rule-path>
6   hashcatXT.py a2 -m <hash-type> --hf <hash-path> --df <folder-path>
7   hashcatXT.py -h | --help
8   hashcatXT.py --version
9
10 Options:
11   --help                 | Show this screen.              |
12   --version              | Show version.                  |
13   -m <hash-type>         | Hash-type, see references below | example: -m
        1700 (see table below)
14   -c <custom-charset>    | User-defined charset ?1        | example: -c -
        or -c @$&-
15   --mk <mask>            | Mask for a mask attack.        | example:
        ?d?l?u?d?1 (see table below)
16   --hf <hash-path>       | Path to the hash file          |
        /path/to/hashfile
17   -d <dictionary-path>   | Path to one dictionary         |
        /path/to/dictionary
18   --df <folder-path>     | path to folder of dictionaries |
        /path/to/folder/of/dictionaries
19   -r <rule-path>         | Path to rule file              |
        /path/to/rulefile
20
21
22 - [ Attack Modes ] -
```

```
23
24    # | Mode
25   ===+======
26    a1 | Tripple Attack
27    a2 | Multi Combination Attack
28
29  - [ Hash modes ] -
30
31        # | Name                                                | Category
32    ======+====================================================+============================
33      900 | MD4                                                | Raw Hash
34        0 | MD5                                                | Raw Hash
35     5100 | Half MD5                                           | Raw Hash
36      100 | SHA1                                               | Raw Hash
37     1300 | SHA-224                                            | Raw Hash
38     1400 | SHA-256                                            | Raw Hash
39    10800 | SHA-384                                            | Raw Hash
40     1700 | SHA-512                                            | Raw Hash
41     5000 | SHA-3 (Keccak)                                     | Raw Hash
42      600 | BLAKE2b-512                                        | Raw Hash
43    10100 | SipHash                                            | Raw Hash
44     6000 | RIPEMD-160                                         | Raw Hash
45     6100 | Whirlpool                                          | Raw Hash
46     6900 | GOST R 34.11-94                                    | Raw Hash
47    11700 | GOST R 34.11-2012 (Streebog) 256-bit               | Raw Hash
48    11800 | GOST R 34.11-2012 (Streebog) 512-bit               | Raw Hash
49       10 | md5($pass.$salt)                                   | Raw Hash, Salted
          and/or Iterated
50       20 | md5($salt.$pass)                                   | Raw Hash, Salted
          and/or Iterated
51       30 | md5(utf16le($pass).$salt)                          | Raw Hash, Salted
          and/or Iterated
52       40 | md5($salt.utf16le($pass))                          | Raw Hash, Salted
          and/or Iterated
53     3800 | md5($salt.$pass.$salt)                             | Raw Hash, Salted
        and/or Iterated
54     3710 | md5($salt.md5($pass))                              | Raw Hash, Salted
        and/or Iterated
55     4010 | md5($salt.md5($salt.$pass))                        | Raw Hash, Salted
        and/or Iterated
56     4110 | md5($salt.md5($pass.$salt))                        | Raw Hash, Salted
        and/or Iterated
57     2600 | md5(md5($pass))                                    | Raw Hash, Salted
        and/or Iterated
58     3910 | md5(md5($pass).md5($salt))                         | Raw Hash, Salted
        and/or Iterated
59     4300 | md5(strtoupper(md5($pass)))                        | Raw Hash, Salted
        and/or Iterated
60     4400 | md5(sha1($pass))                                   | Raw Hash, Salted
        and/or Iterated
```

```
61     110 | sha1($pass.$salt)                                | Raw Hash, Salted
           and/or Iterated
62     120 | sha1($salt.$pass)                                | Raw Hash, Salted
           and/or Iterated
63     130 | sha1(utf16le($pass).$salt)                       | Raw Hash, Salted
           and/or Iterated
64     140 | sha1($salt.utf16le($pass))                       | Raw Hash, Salted
           and/or Iterated
65    4500 | sha1(sha1($pass))                                | Raw Hash, Salted
           and/or Iterated
66    4520 | sha1($salt.sha1($pass))                          | Raw Hash, Salted
           and/or Iterated
67    4700 | sha1(md5($pass))                                 | Raw Hash, Salted
           and/or Iterated
68    4900 | sha1($salt.$pass.$salt)                          | Raw Hash, Salted
           and/or Iterated
69   14400 | sha1(CX)                                         | Raw Hash, Salted
           and/or Iterated
70    1410 | sha256($pass.$salt)                              | Raw Hash, Salted
           and/or Iterated
71    1420 | sha256($salt.$pass)                              | Raw Hash, Salted
           and/or Iterated
72    1430 | sha256(utf16le($pass).$salt)                     | Raw Hash, Salted
           and/or Iterated
73    1440 | sha256($salt.utf16le($pass))                     | Raw Hash, Salted
           and/or Iterated
74    1710 | sha512($pass.$salt)                              | Raw Hash, Salted
           and/or Iterated
75    1720 | sha512($salt.$pass)                              | Raw Hash, Salted
           and/or Iterated
76    1730 | sha512(utf16le($pass).$salt)                     | Raw Hash, Salted
           and/or Iterated
77    1740 | sha512($salt.utf16le($pass))                     | Raw Hash, Salted
           and/or Iterated
78      50 | HMAC-MD5 (key = $pass)                           | Raw Hash,
            Authenticated
79      60 | HMAC-MD5 (key = $salt)                           | Raw Hash,
            Authenticated
80     150 | HMAC-SHA1 (key = $pass)                          | Raw Hash,
           Authenticated
81     160 | HMAC-SHA1 (key = $salt)                          | Raw Hash,
           Authenticated
82    1450 | HMAC-SHA256 (key = $pass)                        | Raw Hash,
           Authenticated
83    1460 | HMAC-SHA256 (key = $salt)                        | Raw Hash,
           Authenticated
84    1750 | HMAC-SHA512 (key = $pass)                        | Raw Hash,
           Authenticated
85    1760 | HMAC-SHA512 (key = $salt)                        | Raw Hash,
           Authenticated
```

```
86   14000 | DES (PT = $salt, key = $pass)                | Raw Cipher,
        Known-Plaintext attack
87   14100 | 3DES (PT = $salt, key = $pass)               | Raw Cipher,
        Known-Plaintext attack
88   14900 | Skip32 (PT = $salt, key = $pass)             | Raw Cipher,
        Known-Plaintext attack
89   15400 | ChaCha20                                     | Raw Cipher,
        Known-Plaintext attack
90     400 | phpass                                       | Generic KDF
91    8900 | scrypt                                       | Generic KDF
92   11900 | PBKDF2-HMAC-MD5                              | Generic KDF
93   12000 | PBKDF2-HMAC-SHA1                             | Generic KDF
94   10900 | PBKDF2-HMAC-SHA256                           | Generic KDF
95   12100 | PBKDF2-HMAC-SHA512                           | Generic KDF
96      23 | Skype                                        | Network Protocols
97    2500 | WPA/WPA2                                     | Network Protocols
98    2501 | WPA/WPA2 PMK                                 | Network Protocols
99    4800 | iSCSI CHAP authentication, MD5(CHAP)         | Network Protocols
100   5300 | IKE-PSK MD5                                  | Network Protocols
101   5400 | IKE-PSK SHA1                                 | Network Protocols
102   5500 | NetNTLMv1                                    | Network Protocols
103   5500 | NetNTLMv1+ESS                                | Network Protocols
104   5600 | NetNTLMv2                                    | Network Protocols
105   7300 | IPMI2 RAKP HMAC-SHA1                         | Network Protocols
106   7500 | Kerberos 5 AS-REQ Pre-Auth etype 23          | Network Protocols
107   8300 | DNSSEC (NSEC3)                               | Network Protocols
108  10200 | CRAM-MD5                                     | Network Protocols
109  11100 | PostgreSQL CRAM (MD5)                        | Network Protocols
110  11200 | MySQL CRAM (SHA1)                            | Network Protocols
111  11400 | SIP digest authentication (MD5)              | Network Protocols
112  13100 | Kerberos 5 TGS-REP etype 23                  | Network Protocols
113  16100 | TACACS+                                      | Network Protocols
114  16500 | JWT (JSON Web Token)                         | Network Protocols
115    121 | SMF (Simple Machines Forum) > v1.1           | Forums, CMS,
          E-Commerce, Frameworks
116    400 | phpBB3 (MD5)                                 | Forums, CMS,
          E-Commerce, Frameworks
117   2611 | vBulletin < v3.8.5                           | Forums, CMS,
          E-Commerce, Frameworks
118   2711 | vBulletin >= v3.8.5                          | Forums, CMS,
          E-Commerce, Frameworks
119   2811 | MyBB 1.2+                                    | Forums, CMS,
          E-Commerce, Frameworks
120   2811 | IPB2+ (Invision Power Board)                 | Forums, CMS,
          E-Commerce, Frameworks
121   8400 | WBB3 (Woltlab Burning Board)                 | Forums, CMS,
          E-Commerce, Frameworks
122     11 | Joomla < 2.5.18                              | Forums, CMS,
          E-Commerce, Frameworks
123    400 | Joomla >= 2.5.18 (MD5)                       | Forums, CMS,
          E-Commerce, Frameworks
```

```
124     400 | WordPress (MD5)                                  | Forums, CMS,
            E-Commerce, Frameworks
125    2612 | PHPS                                             | Forums, CMS,
            E-Commerce, Frameworks
126    7900 | Drupal7                                          | Forums, CMS,
            E-Commerce, Frameworks
127      21 | osCommerce                                       | Forums, CMS,
            E-Commerce, Frameworks
128      21 | xt:Commerce                                      | Forums, CMS,
            E-Commerce, Frameworks
129   11000 | PrestaShop                                       | Forums, CMS,
            E-Commerce, Frameworks
130     124 | Django (SHA-1)                                   | Forums, CMS,
            E-Commerce, Frameworks
131   10000 | Django (PBKDF2-SHA256)                           | Forums, CMS,
            E-Commerce, Frameworks
132   16000 | Tripcode                                         | Forums, CMS,
            E-Commerce, Frameworks
133    3711 | MediaWiki B type                                 | Forums, CMS,
            E-Commerce, Frameworks
134   13900 | OpenCart                                         | Forums, CMS,
            E-Commerce, Frameworks
135    4521 | Redmine                                          | Forums, CMS,
            E-Commerce, Frameworks
136    4522 | PunBB                                            | Forums, CMS,
            E-Commerce, Frameworks
137   12001 | Atlassian (PBKDF2-HMAC-SHA1)                     | Forums, CMS,
            E-Commerce, Frameworks
138      12 | PostgreSQL                                       | Database Server
139     131 | MSSQL (2000)                                     | Database Server
140     132 | MSSQL (2005)                                     | Database Server
141    1731 | MSSQL (2012, 2014)                               | Database Server
142     200 | MySQL323                                         | Database Server
143     300 | MySQL4.1/MySQL5                                  | Database Server
144    3100 | Oracle H: Type (Oracle 7+)                       | Database Server
145     112 | Oracle S: Type (Oracle 11+)                      | Database Server
146   12300 | Oracle T: Type (Oracle 12+)                      | Database Server
147    8000 | Sybase ASE                                       | Database Server
148     141 | Episerver 6.x < .NET 4                           | HTTP, SMTP, LDAP
            Server
149    1441 | Episerver 6.x >= .NET 4                          | HTTP, SMTP, LDAP
            Server
150    1600 | Apache $apr1$ MD5, md5apr1, MD5 (APR)            | HTTP, SMTP, LDAP
            Server
151   12600 | ColdFusion 10+                                   | HTTP, SMTP, LDAP
            Server
152    1421 | hMailServer                                      | HTTP, SMTP, LDAP
            Server
153     101 | nsldap, SHA-1(Base64), Netscape LDAP SHA         | HTTP, SMTP, LDAP
            Server
```

```
154     111 | nsldaps, SSHA-1(Base64), Netscape LDAP SSHA | HTTP, SMTP, LDAP
          Server
155    1411 | SSHA-256(Base64), LDAP {SSHA256}           | HTTP, SMTP, LDAP
         Server
156    1711 | SSHA-512(Base64), LDAP {SSHA512}           | HTTP, SMTP, LDAP
         Server
157   16400 | CRAM-MD5 Dovecot                           | HTTP, SMTP, LDAP
        Server
158   15000 | FileZilla Server >= 0.9.55                 | FTP Server
159   11500 | CRC32                                      | Checksums
160    3000 | LM                                         | Operating Systems
161    1000 | NTLM                                       | Operating Systems
162    1100 | Domain Cached Credentials (DCC), MS Cache  | Operating Systems
163    2100 | Domain Cached Credentials 2 (DCC2), MS Cache 2 | Operating
         Systems
164   15300 | DPAPI masterkey file v1                    | Operating Systems
165   15900 | DPAPI masterkey file v2                    | Operating Systems
166   12800 | MS-AzureSync PBKDF2-HMAC-SHA256            | Operating Systems
167    1500 | descrypt, DES (Unix), Traditional DES      | Operating Systems
168   12400 | BSDi Crypt, Extended DES                   | Operating Systems
169     500 | md5crypt, MD5 (Unix), Cisco-IOS $1$ (MD5)  | Operating Systems
170    3200 | bcrypt $2*$, Blowfish (Unix)               | Operating Systems
171    7400 | sha256crypt $5$, SHA256 (Unix)             | Operating Systems
172    1800 | sha512crypt $6$, SHA512 (Unix)             | Operating Systems
173     122 | macOS v10.4, MacOS v10.5, MacOS v10.6      | Operating Systems
174    1722 | macOS v10.7                                | Operating Systems
175    7100 | macOS v10.8+ (PBKDF2-SHA512)               | Operating Systems
176    6300 | AIX {smd5}                                 | Operating Systems
177    6700 | AIX {ssha1}                                | Operating Systems
178    6400 | AIX {ssha256}                              | Operating Systems
179    6500 | AIX {ssha512}                              | Operating Systems
180    2400 | Cisco-PIX MD5                              | Operating Systems
181    2410 | Cisco-ASA MD5                              | Operating Systems
182     500 | Cisco-IOS $1$ (MD5)                        | Operating Systems
183    5700 | Cisco-IOS type 4 (SHA256)                  | Operating Systems
184    9200 | Cisco-IOS $8$ (PBKDF2-SHA256)              | Operating Systems
185    9300 | Cisco-IOS $9$ (scrypt)                     | Operating Systems
186      22 | Juniper NetScreen/SSG (ScreenOS)           | Operating Systems
187     501 | Juniper IVE                                | Operating Systems
188   15100 | Juniper/NetBSD sha1crypt                   | Operating Systems
189    7000 | FortiGate (FortiOS)                        | Operating Systems
190    5800 | Samsung Android Password/PIN               | Operating Systems
191   13800 | Windows Phone 8+ PIN/password              | Operating Systems
192    8100 | Citrix NetScaler                           | Operating Systems
193    8500 | RACF                                       | Operating Systems
194    7200 | GRUB 2                                     | Operating Systems
195    9900 | Radmin2                                    | Operating Systems
196     125 | ArubaOS                                    | Operating Systems
197    7700 | SAP CODVN B (BCODE)                        | Enterprise
         Application Software (EAS)
```

```
198    7800 | SAP CODVN F/G (PASSCODE)                    | Enterprise
       Application Software (EAS)
199   10300 | SAP CODVN H (PWDSALTEDHASH) iSSHA-1         | Enterprise
       Application Software (EAS)
200    8600 | Lotus Notes/Domino 5                        | Enterprise
       Application Software (EAS)
201    8700 | Lotus Notes/Domino 6                        | Enterprise
       Application Software (EAS)
202    9100 | Lotus Notes/Domino 8                        | Enterprise
       Application Software (EAS)
203     133 | PeopleSoft                                  | Enterprise
        Application Software (EAS)
204   13500 | PeopleSoft PS_TOKEN                         | Enterprise
       Application Software (EAS)
205   11600 | 7-Zip                                       | Archives
206   12500 | RAR3-hp                                     | Archives
207   13000 | RAR5                                        | Archives
208   13200 | AxCrypt                                     | Archives
209   13300 | AxCrypt in-memory SHA1                      | Archives
210   13600 | WinZip                                      | Archives
211   14700 | iTunes backup < 10.0                        | Backup
212   14800 | iTunes backup >= 10.0                       | Backup
213    62XY | TrueCrypt                                   | Full-Disk
       Encryption (FDE)
214      X  | 1 = PBKDF2-HMAC-RIPEMD160                   | Full-Disk
          Encryption (FDE)
215      X  | 2 = PBKDF2-HMAC-SHA512                      | Full-Disk
          Encryption (FDE)
216      X  | 3 = PBKDF2-HMAC-Whirlpool                   | Full-Disk
          Encryption (FDE)
217      X  | 4 = PBKDF2-HMAC-RIPEMD160 + boot-mode       | Full-Disk
          Encryption (FDE)
218       Y | 1 = XTS 512 bit pure AES                    | Full-Disk
            Encryption (FDE)
219       Y | 1 = XTS 512 bit pure Serpent                | Full-Disk
            Encryption (FDE)
220       Y | 1 = XTS 512 bit pure Twofish                | Full-Disk
            Encryption (FDE)
221       Y | 2 = XTS 1024 bit pure AES                   | Full-Disk
            Encryption (FDE)
222       Y | 2 = XTS 1024 bit pure Serpent               | Full-Disk
            Encryption (FDE)
223       Y | 2 = XTS 1024 bit pure Twofish               | Full-Disk
            Encryption (FDE)
224       Y | 2 = XTS 1024 bit cascaded AES-Twofish       | Full-Disk
            Encryption (FDE)
225       Y | 2 = XTS 1024 bit cascaded Serpent-AES       | Full-Disk
            Encryption (FDE)
226       Y | 2 = XTS 1024 bit cascaded Twofish-Serpent   | Full-Disk
            Encryption (FDE)
```

```
227      Y | 3 = XTS 1536 bit all                          | Full-Disk
         Encryption (FDE)
228    8800 | Android FDE <= 4.3                          | Full-Disk
         Encryption (FDE)
229   12900 | Android FDE (Samsung DEK)                   | Full-Disk
         Encryption (FDE)
230   12200 | eCryptfs                                    | Full-Disk
         Encryption (FDE)
231   137XY | VeraCrypt                                   | Full-Disk
         Encryption (FDE)
232      X  | 1 = PBKDF2-HMAC-RIPEMD160                   | Full-Disk
         Encryption (FDE)
233      X  | 2 = PBKDF2-HMAC-SHA512                      | Full-Disk
         Encryption (FDE)
234      X  | 3 = PBKDF2-HMAC-Whirlpool                   | Full-Disk
         Encryption (FDE)
235      X  | 4 = PBKDF2-HMAC-RIPEMD160 + boot-mode       | Full-Disk
         Encryption (FDE)
236      X  | 5 = PBKDF2-HMAC-SHA256                      | Full-Disk
         Encryption (FDE)
237      X  | 6 = PBKDF2-HMAC-SHA256 + boot-mode          | Full-Disk
         Encryption (FDE)
238      Y | 1 = XTS 512 bit pure AES                     | Full-Disk
         Encryption (FDE)
239      Y | 1 = XTS 512 bit pure Serpent                | Full-Disk
         Encryption (FDE)
240      Y | 1 = XTS 512 bit pure Twofish                | Full-Disk
         Encryption (FDE)
241      Y | 2 = XTS 1024 bit pure AES                   | Full-Disk
         Encryption (FDE)
242      Y | 2 = XTS 1024 bit pure Serpent               | Full-Disk
         Encryption (FDE)
243      Y | 2 = XTS 1024 bit pure Twofish               | Full-Disk
         Encryption (FDE)
244      Y | 2 = XTS 1024 bit cascaded AES-Twofish       | Full-Disk
         Encryption (FDE)
245      Y | 2 = XTS 1024 bit cascaded Serpent-AES       | Full-Disk
         Encryption (FDE)
246      Y | 2 = XTS 1024 bit cascaded Twofish-Serpent   | Full-Disk
         Encryption (FDE)
247      Y | 3 = XTS 1536 bit all                         | Full-Disk
         Encryption (FDE)
248   14600 | LUKS                                        | Full-Disk
         Encryption (FDE)
249    9700 | MS Office <= 2003 $0/$1, MD5 + RC4          | Documents
250    9710 | MS Office <= 2003 $0/$1, MD5 + RC4, collider #1 | Documents
251    9720 | MS Office <= 2003 $0/$1, MD5 + RC4, collider #2 | Documents
252    9800 | MS Office <= 2003 $3/$4, SHA1 + RC4         | Documents
253    9810 | MS Office <= 2003 $3, SHA1 + RC4, collider #1 | Documents
254    9820 | MS Office <= 2003 $3, SHA1 + RC4, collider #2 | Documents
255    9400 | MS Office 2007                              | Documents
```

```
256    9500 | MS Office 2010                                | Documents
257    9600 | MS Office 2013                                | Documents
258   10400 | PDF 1.1 - 1.3 (Acrobat 2 - 4)                 | Documents
259   10410 | PDF 1.1 - 1.3 (Acrobat 2 - 4), collider #1    | Documents
260   10420 | PDF 1.1 - 1.3 (Acrobat 2 - 4), collider #2    | Documents
261   10500 | PDF 1.4 - 1.6 (Acrobat 5 - 8)                 | Documents
262   10600 | PDF 1.7 Level 3 (Acrobat 9)                   | Documents
263   10700 | PDF 1.7 Level 8 (Acrobat 10 - 11)             | Documents
264   16200 | Apple Secure Notes                            | Documents
265    9000 | Password Safe v2                              | Password Managers
266    5200 | Password Safe v3                              | Password Managers
267    6800 | LastPass + LastPass sniffed                   | Password Managers
268    6600 | 1Password, agilekeychain                      | Password Managers
269    8200 | 1Password, cloudkeychain                      | Password Managers
270   11300 | Bitcoin/Litecoin wallet.dat                   | Password Managers
271   12700 | Blockchain, My Wallet                         | Password Managers
272   15200 | Blockchain, My Wallet, V2                     | Password Managers
273   13400 | KeePass 1 (AES/Twofish) and KeePass 2 (AES) | Password Managers
274   15500 | JKS Java Key Store Private Keys (SHA1)        | Password Managers
275   15600 | Ethereum Wallet, PBKDF2-HMAC-SHA256           | Password Managers
276   15700 | Ethereum Wallet, SCRYPT                       | Password Managers
277   16300 | Ethereum Pre-Sale Wallet, PBKDF2-HMAC-SHA256 | Password Managers
278   99999 | Plaintext                                     | Plaintext
279
280
281 - [ Built-in Charsets ] -
282
283   ? | Charset
284  ===+==========
285   l | abcdefghijklmnopqrstuvwxyz
286   u | ABCDEFGHIJKLMNOPQRSTUVWXYZ
287   d | 0123456789
288   h | 0123456789abcdef
289   H | 0123456789ABCDEF
290   s |  !"#$%&'()*+,-./:;<=>?@[\]^_`{|}~
291   a | ?l?u?d?s
292   b | 0x00 - 0xff
293
294 - [ Custom Charsets ] -
295
296   ? | Charset
297  ===+==========
298   1 | User defined
299
300 """
301
302 import sys
303 import os
304 sys.path.insert(0, os.getcwd()+'/library')
305 sys.path.insert(0, os.getcwd()+'/module')
306 from library.docopt import docopt
```

```
307 from module import tripple_attack, multi_combinator
308
309
310 def main():
311     args = docopt(__doc__, version='hashcatXT 1.0')
312     if (args['a1'] == True):
313         tripple_attack.tripple_attack(args['-d'], args['--mk'], args['-c'],
                args['-m'], args['--hf'], args['-r'])
314     if (args['a2'] == True):
315         multi_combinator.multi_combinator(args['-m'], args['--hf'],
                args['--df'])
316     # if ((args['a1'] is False) and (args['a2'] is False)):
317     #     print('\nWRONG ARGUMENT, PLEASE USE "hashcatXT.py --help" FOR
            MORE INFORMATION')
318
319
320 if __name__ == '__main__':
321     main()
```

C.2 Modul tripple_attack.py

Listing C.2: Programmcode des Moduls tripple_attack.py

```
1  #!/usr/bin/python
2
3  from subprocess import *
4  import threading
5
6
7  def stdout_printer(p):
8      for line in p.stdout:
9          print(line.rstrip().decode('utf-8'))
10
11 def tripple_attack(dictionary, mask, charset, hash_mode, hash_file,
       rule_file):
12     if charset is None:
13         hybrid_attack = Popen(['hashcat', '-a', '6', dictionary, mask,
               '-i', '--stdout'], stdout=PIPE)
14     else:
15         hybrid_attack = Popen(['hashcat', '-a', '6', dictionary, '-1',
               charset, mask, '-i', '--stdout'], stdout=PIPE)
16     rule_attack = Popen(['hashcat', '-a', '0', '-m', hash_mode, hash_file,
           '-r', rule_file, '-O', '--potfile-disable'],
17                         stdin=hybrid_attack.stdout, stdout=PIPE)
18     # t = threading.Thread(target=stdout_printer, args=(hybrid_attack,))
19     # t.start()
20     p = threading.Thread(target=stdout_printer, args=(rule_attack,))
21     p.start()
22
```

```
23      # rule_attack.stdin.close()
24      # t.join()
25      p.join()
26      # output = rule_attack.communicate()[0]
27      # print(output)
```

C.3 Modul multi_combinator.py

Listing C.3: Programmcode des Moduls multi_combinator.py

```
1     #!/usr/bin/python
2
3   import os
4   from subprocess import *
5   import itertools
6   import threading
7
8
9   def stdout_printer(p):
10      for line in p.stdout:
11          print(line.rstrip())
12
13
14  def multi_combinator(hash_mode, hash_file, directory):
15      erg = Popen(['hashcat', '-a', '0', '-m', hash_mode, hash_file, '-O',
            '--potfile-disable'],
16                  stdin=PIPE,
17                  stdout=PIPE,
18                  stderr=STDOUT,
19                  universal_newlines=True)
20      t = threading.Thread(target=stdout_printer, args=(erg,))
21      t.start()
22      file = []
23      with os.scandir(directory) as listOfEntries:
24          for entry in listOfEntries:
25              if entry.is_file() and entry.name is not ".DS_Store":
26                  file.append(open(directory + entry.name).readlines())
27          file = list(itertools.permutations(file))
28          for b in range(0, len(file)):
29              for i in itertools.product(*file[b]):
30                  test = '\n'.join(i).replace("\n", "")
31                  # print(test + "\n")
32                  erg.stdin.write((test + "\n"))
33                  erg.stdin.flush()
34      erg.stdin.close()
35      t.join()
```

Literatur

Bonneau, J. (2012). *Guessing human-chosen secrets* (Techn. Ber. Nr. UCAM-CL-TR-819). University of Cambridge. (Siehe S. 13).

Carus, M. (2008). *Ethical hacking: Strategien für Ihre Sicherheit*. Entwickler.Press. (Siehe S. 20, 21).

Dürmuth, M. & Kranz, T. (2014). On Password Guessing with GPUs and FPGAs. In *Proceedings of the International Conference on Passwords (PASSWORDS)* (Bd. 9393, S. 19–38). Lecture Notes in Computer Science. Springer. (Siehe S. 18, 19, 24).

Eckert, C. (2008). *IT-Sicherheit: Konzepte – Verfahren – Protokolle* (5. Auflage). Oldenbourg Wissensch.Vlg. (Siehe S. 12, 24).

Florencio, D. & Herley, C. (2006). *A Large Scale Study of Web Password Habits* (Techn. Ber. Nr. MSR-TR-2006-166). Microsoft Research. Zugriff unter https://www.microsoft.com/en-us/research/publication/a-large-scale-study-of-web-password-habits/. (Siehe S. 38)

Fox, D. (2008). Entropie. *Datenschutz und Datensicherheit - DuD*, *32*(8), 543–543. doi:10.1007/s11623-008-0128-2. (Siehe S. 11)

Goll, J. & Hommel, D. (2015). *Mit Scrum zum gewünschten System*. Springer Vieweg. (Siehe S. 63).

Hatzivasilis, G., Papaefstathiou, I. & Manifavas, C. (2015). Password Hashing Competition - Survey and Benchmark. Cryptology ePrint Archive, Report 2015/265. Zugriff unter https://eprint.iacr.org/2015/265. (Siehe S. 17)

ISO/IEC 27000. (2018). ISO/IEC 27000: Information technology – Security techniques – Information security management systems – Overview and vocabulary. ISO/IEC, Genf, Schweiz. (Siehe S. 1).

Jaeger, D., Pelchen, C., Graupner, H., Cheng, F. & Meinel, C. (2016). Analysis of Publicly Leaked Credentials and the Long Story of Password (Re-)use. In *Proceedings of the International Conference on Passwords (PASSWORDS)*. Zugriff unter http://mykayem.org/pdfs/Jaeger2016.pdf. (Siehe S. 13)

Karpfinger, C. & Kiechle, H. (2009, 27. Oktober). *Kryptologie*. Vieweg+Teubner Verlag. (Siehe S. 1).

Kasabov, A. & van Kerkwijk, J. (2011). *Distributed GPU Password Cracking*. Universiteit can Amsterdam. Zugriff unter http://ext.delaat.net/rp/2010-2011/p11/report.pdf. (Siehe S. 23)

LeBlanc, J. & Messerschmidt, T. (2016). *Identity and Data Security for Web Development: Best Practices* (1st). O'Reilly Media, Inc. (Siehe S. 20, 22).

Manaras, P., Hertlein, M. & Pohlmann, N. (2016). Die Zeit nach dem Passwort. *Datenschutz und Datensicherheit - DuD*, *40*(4), 206–211. (Siehe S. 1).

Mitnick, S. (2002, 27. September). *Art of Deception*. John Wiley & Sons. (Siehe S. 3).

Moschgath, M.-L., Roedig, U. & Schumacher, M. (2012, 29. Oktober). *Hacker Contest: Sicherheitsprobleme, Lösungen, Beispiele*. Springer. (Siehe S. 6, 7).

Nielsen, G., Vedel, M. & Jensen, C. D. (2014). Improving usability of passphrase authentication. In *Proceedings of the Twelfth Annual International Conference on Privacy, Security and Trust* (S. 189–198). doi:10.1109/PST.2014.6890939. (Siehe S. 11, 57)

Oechslin, P. (2003). Making a Faster Cryptanalytic Time-Memory Trade-Off. In D. Boneh (Hrsg.), *Proceedings of the 23rd Annual Cryptology Conference – Advances in Cryptology (CRYPTO 2003)* (Bd. 2729, S. 617–630). Lecture Notes in Computer Science (LNCS). Berlin, Heidelberg: Springer Berlin Heidelberg. (Siehe S. 23).

Paar, C. & Pelzl, J. (2016, 24. September). *Kryptografie verständlich*. Springer-Verlag GmbH. (Siehe S. 3, 19).

Provos, N. & Mazières, D. (1999). A Future-adaptive Password Scheme. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference* (S. 32–32). ATEC '99. Monterey, California: USENIX Association. Zugriff unter http://dl.acm.org/citation.cfm?id=1268708.1268740. (Siehe S. 19)

Raza, M., Iqbal, M., Sharif, M. & Haider, W. (2012). A Survey of Password Attacks and Comparative Analysis on Methods for Secure Authentication. *World Applied Sciences Journal*, *19*(4), 439–444. (Siehe S. 7, 21).

Rohr, M. (2015, 9. März). *Sicherheit von Webanwendungen in der Praxis*. Vieweg+Teubner Verlag. (Siehe S. 9, 12).

Schneier, B. (2000). *Secrets & Lies: Digital Security in a Networked World* (1st). New York, NY, USA: John Wiley & Sons, Inc. (Siehe S. 20).

Schwenk, J. (2014, 27. August). *Sicherheit und Kryptographie im Internet*. Vieweg+Teubner Verlag. (Siehe S. 14).

Shay, R., Kelley, P. G., Komanduri, S., Mazurek, M. L., Ur, B., Vidas, T., ... Cranor, L. F. (2012). Correct Horse Battery Staple: Exploring the Usability of System-assigned Passphrases. In *Proceedings of the Eighth Symposium on Usable Privacy and Security (SOUPS 2012)* (7:1–7:20). doi:10.1145/2335356.2335366. (Siehe S. 10)

Spitz, S., Pramateftakis, M. & Swoboda, J. (2011). *Kryptographie und IT-Sicherheit* (2. Aufl.). Vieweg+Teubner Verlag. (Siehe S. 5, 14, 15).

Ur, B., Noma, F., Bees, J., Segreti, S. M., Shay, R., Bauer, L., ... Cranor, L. F. (2015). „I Added '!' at the End to Make It Secure": Observing Password Creation in the Lab. In *Proceedings of the Eleventh Symposium On Usable Privacy and Security (SOUPS 2015)* (S. 123–140). Ottawa: USENIX Association. Zugriff unter https://www.usenix.org/conference/soups2015/proceedings/presentation/ur. (Siehe S. 21, 38, 44)

Weir, M., Aggarwal, S., Collins, M. & Stern, H. (2010). Testing Metrics for Password Creation Policies by Attacking Large Sets of Revealed Passwords. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS)* (S. 162–175). CCS '10. doi:10.1145/1866307.1866327. (Siehe S. 22)

Wheeler, D. L. (2016). zxcvbn: Low-Budget Password Strength Estimation. In *Proceedings of the 25th USENIX Security Symposium (USENIX Security)* (S. 157–173). Austin, TX: USENIX Association. Zugriff unter https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/wheeler. (Siehe S. 13)

Wiemer, F. & Zimmermann, R. (2014). High-speed implementation of bcrypt password search using special-purpose hardware. In *Proceedings of the International Conference on ReConFigurable Computing and [FPGAs (ReConFig14)* (S. 1–6). doi:10.1109/ReConFig.2014.7032529. (Siehe S. 18)

Woyand, H.-B. (2017). *Python für Ingenieure und Naturwissenschaftler*. Carl Hanser Verlag GmbH & Co. KG. (Siehe S. 67).