Hochschule Esslingen
University of Applied Sciences

Bachelorarbeit

Entwurf und Realisierung eines Fuzzing Tools für kryptographische Protokolle

im Studiengang Kommunikationstechnik
der Fakultät Informationstechnik
7. Semester
an der Hochschule Esslingen

Stefan Ringhoffer

Zeitraum: 1.3.2015 - 30.6.2015
Prüfer: Professor Dr. Dominik Schoop
Zweitprüfer: Professor Dr. Manfred Dausmann

Firma: secuvera GmbH
Betreuer: Dipl.-Inform. Sebastian Fritsch

Vorwort

Zunächst möchte ich mich an dieser Stelle bei all denjenigen bedanken, die mich während der Anfertigung dieser Bachelorarbeit unterstützt und motiviert haben.

Ganz besonders gilt dieses Dank Herrn Prof. Dr. Dominik Schoop, der meine Arbeit und somit auch mich betreut hat. Sie gaben immer wieder durch kritisches Hinterfragen wertvolle Hinweise. Vielen Dank für die Geduld und Mühen.

Auch meine Vorgesetzten und Kollegen haben maßgeblich daran mitgewirkt, dass diese Bachelorarbeit nun in dieser Form vorliegt. Vielen Dank, dass Sie mir die Möglichkeit gegeben haben, bei Ihnen zu forschen und zu arbeiten.

Daneben gilt mein Dank allen, welche in zahlreichen Stunden Korrektur gelesen haben. Alle wiesen auf Schwächen hin und konnten als Fachfremde immer wieder zeigen, wo noch Erklärungsbedarf bestand. Insbesondere ist ein Dank an Herrn Fritsch auszusprechen, der über die sechs Monate mein Betreuer innerhalb der Firma war.

Zusammenfassung

Im Rahmen dieser Arbeit werden zuerst die möglichen Ansätze für das Fuzzing von kryptographischen Protokollen erarbeitet. Dabei gibt es zwei verschiedene Ansätze ein kryptographisches Protokoll zu fuzzen. Der erste Ansatz beinhaltet die Nachrichten, die unverschlüsselt über das Netzwerk gesendet werden. Hierzu zählen Nachrichten, die den Austausch der Algorithmen sowie den Schlüsselaustausch ermöglichen. Der zweite Ansatz des Fuzzings umfasst die verschlüsselten Nachrichten, in denen Passwörter, Public-Keys und Längenfelder mit mutierten Werten gefuzzt werden. Die Anforderung an diesen Ansatz ist, dass die verschlüsselten Nachrichten zuerst entschlüsselt werden müssen. Nach der Mutation der zu fuzzenden Nachricht müssen diese mit einem neuen Nachrichtenauthentifizierungscode wieder verschlüsselt werden.

Auf Basis der erstellten Ansätze und der erstellten Anforderung wird ein Entwurf eines SSH-Fuzzers erstellt. Der SSH-Fuzzer ist nach dem Prinzip eines Interceptors aufgebaut: Die empfangenen Nachrichten des Clients werden abgefangen und durch den Fuzzer an der bestimmten Position eingebaut. Die Message-ID und die Position, die gefuzzt werden, sind dafür in einer Testfalldatenbank hinterlegt. Nach der Veränderung wird die Nachricht an die zu testende OpenSSH-Implementierung gesendet. Die Reaktion des SSH-Servers wird mit Funktionsblöcken zur Verbindungsüberprüfung, zur Ressourcenüberwachung und zur Auswertung des Loggingfiles überwacht. Eine weitere Funktion des SSH-Fuzzers ist, die gesammelten Ergebnisse des ausgeführten Testfalles in einer Text- und CSV-Datei festzuhalten.

Der implementierte SSH-Fuzzer, der in Perl programmiert ist, kann die SSH-Nachrichten Key Exchange Init, ECDH Key Exchange Init, Service Request und Userauth Request verändern. Das korrekte Fuzzing der aufgezählten Nachrichten wird durch genaue Tests verifiziert. Die Ergebnisse der durchgeführten Testfälle werden daraufhin analysiert und bewertet. Durch die Analyse des Ergebnisses aus Testfall 4 wurde ein Fehler in der Implementierung gefunden, da keine SSH-Disconnect Nachricht versendet wurde. Die Analyse und Bewertung der Testfälle zeigt, dass die Durchführung des kryptographischen Fuzzings seinen Sinn und Zweck erfüllt. Auf der Basis der Ergebnisse werden Weiterentwicklungsmöglichkeiten des Fuzzers erörtert. Hierbei sind die Aspekte der Quantisierung von Testfällen, die Effizienz des Fuzzers und der Mitschnitt von Paketen zu nennen.

Abstract

The aim of this thesis is to investigate the possibilities of cryptographic fuzzing and to give a proof of concept of the implemented SSH fuzzer. There are two different phases of cryptographic fuzzing. The first phase contains the following unencrypted messages: the exchange of algorithms and the key exchange of the cryptographic protocol. The second part includes the encrypted messages of a cryptographic protocol. In this part, it is possible to fuzz public keys, password strings and length values of the messages. Furthermore, in this part, the fuzzed data must first be decrypted. The modified data must be signed with a new MAC and must be encrypted.

Based on the approach of cryptographic fuzzing a draft of an SSH fuzzer is created. The SSH fuzzer is based on an interceptor, which receives messages from a SSH client and manipulates the message at a certain position. The fuzzing position and the message identifier are stored in a test case database. After the mutation of a certain SSH message has taken place the changed data is sent to the OpenSSH server. The reaction of the SSH server is monitored by the following functions: connection check to the SSH server, evaluation of the exhausted resources and the evaluation of the OpenSSH logging file. The purpose of the text and CSV files are to store all the collected information of each test case.

The implemented SSH fuzzer, which is programmed in Perl, can manipulate the following SSH messages: Key Exchange Init, ECDH Key Exchange Init, Service Request and Userauth Request. The proper manipulation of the given SSH messages is verified by precise tests. Afterwards the results of the fuzzing are being analysed and evaluated. A malfunction is found in test case 4. In this test case no SSH disconnect message is sent by the SSH server after the fuzzed message is sent to the server. The analysis of the test cases and the evaluation prove that it is rational to use fuzzing as an alternative method to audit an implementation.

The test setup of the SSH fuzzer could be refined by implementing more test cases, increasing the efficiency and recording the transferred packets.

Ehrenwörtliche Erklärung

Hiermit versichere ich, die vorliegende Arbeit selbstständig und unter ausschließlicher Verwendung der angegebenen Literatur und Hilfsmittel erstellt zu haben.

Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Esslingen, den 30. Juni 2015 _____

Unterschrift

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

1 Einleitung

In der heutigen Zeit spielt das Internet im alltäglichen Leben eine immer bedeutendere Rolle, so zum Beispiel bei der Verwendung von Social Media Plattformen oder der Verwaltung des eigenen Bankkontos per Onlinebanking. Die Software, welche solche Dienste bereitstellt, tauscht über das Internet sensible Daten aus. Damit solche sensiblen Daten nicht von unberechtigten Personen gelesen oder verändert werden können, haben sich kryptographische Protokolle zur sicheren Kommunikation etabliert. Leider kann auch Software, welche die sichere Kommunikation gewährleistet, aufgrund ihrer hohen Komplexität eklatante Schwachstellen aufweisen. Dies zeigen, die in jüngster Vergangenheit aufgetretenen Schwachstellen wie zum Beispiel Heartbleed oder Poodle. Das Fuzzing ist eine mögliche Methode, Schwachstellen in einer Implementierung aufzuspüren.

Die Ziele dieser Arbeit beruhen auf zwei wichtigen Punkten. Der erste Punkt beinhaltet, dass die Möglichkeiten und damit auch die Grenzen des kryptographischen Fuzzings bestimmt werden sollen. Der zweite Punkt ist zu zeigen, dass das Fuzzing eines kryptographischen Protokolls einen Mehrwert für die Untersuchung der Implentierung auf Schwachstellen bewirkt. Dies soll durch die Implementierung eines SSH-Fuzzers geschehen. Dieser SSH-Fuzzer testet die OpenSSH Implementierung über die Netzwerkschnittstelle auf seine kryptographischen Eigenschaften.

Für eine strukturierte Herangehensweise an das Thema des Fuzzings wird zuerst das Fuzzing und das SSH-Protokoll (Version 2) im Kapitel 2 “Grundlagen” erläutert. Desweiteren werden aus den Grundlagen des Fuzzings und des SSH-Protokolls die Anforderungen und Möglichkeiten des kryptographischen Fuzzings erstellt.

Aus den formulierten Anforderungen und Möglichkeiten wird ein Entwurf für einen SSH-Fuzzer erstellt. Der implementierte SSH-Fuzzer auf Basis des Entwurfes wird in Kapitel 4 “Implementierung” näher beschrieben. Danach gilt es den SSH-Fuzzer im Kapitel 5 “Test des Fuzzers” auf seine Funktionalität zu überprüfen.

Im Kapitel 6 “Ergebnisse” werden die ausgeführten Tests des SSH-Fuzzers jeweils einzeln analysiert und bewertet. Die Analyse der Testfälle erfolgt mit den gesammelten Informationen der Ressourcenauswertung, der Loggingauswertung und der Verbindungsüberprüfung. Auf Basis der Analyse wird bewertet, ob die Testfälle ein Fehlverhalten ausgeöst haben oder nicht.

Zum Schluss wird das Ergebnis der Arbeit bewertet und in einem Ausblick werden Verbesserungsvorschläge zum SSH-Fuzzer in den Punkten Leistungsverbesserung, Quantisierung der Testfälle und Mitschnitt der Pakete gegeben.

2 Grundlagen

Das nötige Verständnis für das Entwerfen und Implementieren eines Fuzzing Tools für kryptographische Protokolle soll in diesem Kapitel gelegt werden. Das Verständnis basiert auf den allgemeinen Grundlagen des Fuzzings und auf den daraus abgeleiteteten Möglichkeiten und Anforderungen für das Fuzzing eines kryptographischen Protokolls.

2.1 Grundlagen des Fuzzings

Das White- oder Blackboxtesting von Software umfasst den Begriff des Fuzzing. Hierfür werden mit einem Programm zufällige Daten erzeugt, die über Eingabeschnittstellen eines Programms verarbeitet werden. Das Ziel des Fuzzing ist es, durch die zufällig generierten Daten einen Fehler im Ablauf eines Programms zu verursachen. Doch bevor auf die Funktionsweise des Fuzzings eingegangen werden soll, erfolgt im nächsten Kapitel eine kleine historische Einführung zum Thema Fuzzing.

2.1.1 Historie des Fuzzings

Die ersten Erfolge im Bereich des Fuzzings sind im Jahr 1989 an der Universität von Wisconsin Madison erzielt worden. An dieser Universität gab es ein Projekt, das von Professor Barton Miller geleitet wurde. Dieses Projekt wurde einem zweiköpfigen Studententeam zugewiesen. Dieses Studententeam bestand aus den Studenten Lars Fredriksen und Bryan So [2] . Sie untersuchten die Fragestellung, ob durch zufällig generierte Eingabedaten Abstürze eines zu testenden Programms erzeugt werden können. Als ihre Idee funktionierte, begannen auch andere Personen sich mit diesem Thema auseinander zu setzen. So wurde im Jahr 2002 das Fuzzing Tool PROTOS [3], das von der Oulu Universität entwickelt wurde, vorgestellt. Desweiteren wurde in der selben Zeit das Fuzzing Framework SPIKE [4] von Dave Aitel auf der Blackhat Konferenz in den USA vorgestellt. Es wurden weitere Fuzzing Tools und Frameworks entwickelt, die sich in Abhängigkeit ihres Zieles in zwei Kategorien unterteilen lassen. Die erste Kategorie nennt sich Local Fuzzers. In diese Kategorie fallen Fuzzer, die lokal auf einem System versuchen Sicherheitslücken von Programmen zu entdecken. Dafür wird das Kommandozeilenfuzzing oder das Umgebungsvariablenfuzzing genutzt. Die zweite Art von Fuzzer sind die so genannten Remote Fuzzer. Die Remote Fuzzer versuchen über eine Netzwerkschnittstelle auf einem fremden System

Sichheitslücken durch Fuzzing zu entdecken. Die Unterscheidung zwischen den zwei Fuzzertypen ist notwendig, damit ein möglicher Anwender schneller den gewünschten Fuzzer, der auf seine zu testende Implementierung passt, findet.

2.1.2 Funktionsprinzip des Fuzzings

Damit durch einen Fuzzer, egal von welchem Typ der Fuzzer ist, eine lückenlose Prüfung durchgeführt wird, müssen die im Folgenden aus dem Buch "Fuzzing: Brute Force Vulnerability Discovery" [16] entnommenen Phasen durchlaufen werden. Diese sechs zu durchlaufenden Phasen sind in Abbildung 2.1 zu sehen.

Abb. 2.1: Entwurfsphasen des Fuzzing

2.1.2.1 Festlegen des Zieles

Bevor ein bestimmtes Fuzzing Tool ausgewählt wird, muss festgelegt werden, welches Ziel gefuzzed werden soll. Im Falle eines Remote Fuzzers muss man sich auf ein Netzwerkprotokoll festlegen, mit dem die zu testende Schnittstelle der Implementierung kommuniziert.

2.1.2.2 Ermitteln der jeweiligen Eingabeparameter

Nachdem das Ziel festgelegt wurde, muss das Ziel auf mögliche Eingabeparameter untersucht werden. Diese ermittelten Eingabeparameter sollten eine hohe Möglichkeit bieten,

Schwachstellen des Progammes ausfindig zu machen. Nahezu alle verwertbaren Schwachstellen werden von Programmen verursacht, die Benutzereingaben akzeptieren und diese Daten, ohne sie zuerst zu bereinigen oder zu überprüfen, verarbeiten. Für den Erfolg des Fuzzings ist es von Bedeutung, eine feste Anzahl von Eingabeparametern zu bestimmen. Die Eingabeparameter bestehen aus Headern, Dateinamen, Umgebungsvariablen, Registrierungsschlüssel und so weiter. Alle berücksichtigten Eingabeparameter sollten als potenzielle Fuzzparameter in Betracht gezogen werden.

2.1.2.3 Erzeugen von gefuzzten Daten

Sobald die Eingabeparameter festgelegt sind, gilt es passende Fuzzdaten zu generieren. Bei diesem Schritt sollte darauf geachtet werden, dass die Fuzzdaten so generiert werden, dass sie auch vom zu testenden Programm verarbeitet werden. Die Entscheidung, ob statisch oder dynamisch gefuzzte Daten benutzt werden, hängt vom jeweiligem Ziel ab. Im Beispiel eines Remote Fuzzers sollten die Fuzzdaten nach dem definierten Standard des Protokolls entworfen werden. Die Vorgehensweise dazu ist, dass man einzelne Testfälle manuell aus den Protokollspezifikationen erstellt. Mehr Testfälle können durch weitere Variation der Fuzzdaten erstellt werden. Diese Variation sollte aus Zeitgründen automatisiert erfolgen. Dies kann von einem Computer geschehen, da diese Aufgabe kein Wissen der zu fuzzenden Eingabeschnittstelle erfordert.

2.1.2.4 Injektion der gefuzzten Daten

Dieser Schritt hängt mit dem vorherigen Schritt eng zusammen. Die Ausführung kann das Senden von Datenpaketen an das zu fuzzene Ziel, das Öffnen einer Datei oder das Starten des Zielprozesses beinhalten. Auch hier ist die Automatisierung entscheidend. Denn ohne Automatisierung könnte kein effektives Fuzzing betrieben werden. Die manuelle Ausführung von Millionen an Testfällen ist im Verhältnis zu der automatischen Ausführung der Testfälle um ein vielfaches zeitaufwändiger.

2.1.2.5 Überwachung des Zieles

Ein unverzichtbarer, aber oft unterschätzter Schritt während des Fuzzings ist der Überwachungsprozess des zu testenden Programms auf mögliche Fehler. Die Übermittlung von 10.000 gefuzzten Paketen an einen Webserver, die letztendlich einen Absturz des Webservers verursachen, sind nicht verwertbar, sofern wir das Paket nicht identifizieren können, das für den Absturz verantwortlich ist. Daher ist ein Logging der Testfälle, die das Fuzzing Tool ausführt, unverzichtbar. Nur so ist gewährleistet, dass Testfälle zu einem beliebigen Zeitpunkt wiederholt werden können und dadurch der gleiche Fehler reproduziert werden kann.

2.1.2.6 Untersuchung der gefundenen Fehler

Wenn ein Fehler entdeckt wird, muss festgestellt werden, ob der entdeckte Fehler weiter ausgenutzt werden kann oder nicht. Dies ist normalerweise ein Vorgang, der spezielles Wissen erfordert. In diesem Schritt ist es nötig, einen weiteren IT-Sicherheitsexperten hinzuzuziehen. Dieser IT-Sicherheitsexperte sollte Kenntnisse über die Funktionsweise des Betriebsystems besitzen, damit versucht werden kann den entdeckten Fehler auszunutzen. Die Ausnutzung des Fehlers kann dazu führen, dass der IT-Sicherheitsexperte nicht erlaubten Code ausführen kann.

2.1.3 Theoretische Ansätze des Fuzzings

Die möglichen Ansätze, auf dem ein Fuzzingprogramm beruht, lassen sich auf zwei theoretische Möglichkeiten herunterbrechen.

2.1.3.1 Mutationsbasierter Ansatz

Der mutationsbasierte Ansatz basiert darauf, dass zum Beispiel über einen Sniffer gültige Testdaten, die auf das zu testende Protokoll passen, gesammelt werden. Die Fuzzeranwendung mutiert diese Daten auf die verschiedenste Weise und schickt diese an die Zielanwendung. Im Falle des Netzwerkfuzzings kommt ein Sniffer zum Einstatz, um gültigen Verkehr des Protokolls mitzuschneiden und zu analysieren.

Der mutationsbasierte Ansatz für einen Fuzzer kommt vor allem dann zum Einsatz, wenn von der zu testenden Implementierung kein Sourcecode oder ein entsprechender beschriebener Standard vorliegt. Somit kann keine Analyse der Implementerung statt finden. Der darauf basierende logische Aufbau der gefuzzten Daten kann für den späteren Fuzzer nicht erarbeitet werden.

Der Entwurf des mutationsbasierten Fuzzing findet seine Anwendung eher im Bereich des Filefuzzings. Der nächste beschriebene Ansatz eignet sich daher besser für das Fuzzing eines Netzwerkprotokolls.

2.1.3.2 Generationsbasierter Ansatz

Unter dem Begriff generationsbasierter Ansatz versteht man, dass das Fuzzingprogramm intelligente Testfälle basierend auf einem Netzwerkprotokoll oder eines bestimmtes Dateiformates erstellt. Für den Ansatz des generationsbasierten oder auch intelligentes Brute Force Fuzzing genannt, ist es wichtig, eine Analyse der zu fuzzenden Implementierung durchzuführen. Diese Analyse kann sich in den folgenden Punkten unterscheiden:

Wenn der definierte Standard oder der Sourcecode des zu testenden Programms dem Entwickler des Fuzzingtools vorliegt, spricht man von Whiteboxtesting. Im Falle des Whiteboxtesting liegt entweder der komplette Sourcecode vor oder die Implemementierung wurde exakt nach einem beschriebenen Standard programmiert.

Der Begriff des Greyboxtesting im Zusammenhang mit einem Fuzzingtool wird dann verwendet, wenn nur Teile des Sourcecodes vorliegen oder nur Teile der Implementierung nach dem vorliegenden Standard programmiert wurden. Zudem kann der Fall auftreten, dass während des Fuzzings eines Programms undokumentierte Schnittstellen auftauchen. Durch das Fehlen dieser Informationen erhöht der Fuzzer seine theoretische Codeabdeckung deutlich.

Je mehr Protokoll- oder Dateiwissen aus einem Standard oder dem Sourcecode in den Fuzzer eingearbeitet werden kann, desto mehr Codeabdeckung und Fahrtengang kann durch den Fuzzer erreicht werden. Unter dem Begriff der Codeabdeckung und Fahrtengang versteht man, wieviel Code durch das Testen des Programms durchlaufen wird. Beim Testen sollte man immer versuchen so viel wie möglich Codeabdeckung zu generieren, da man nur so sicher gehen kann, dass das entwickelte Programm auch sauber auf fehlerbehaftete Eingaben reagiert.

Im Falle des Whiteboxtesting sollte versucht werden eine größtmögliche Codeabdeckung zu erzielen, da der vorhandene Code oder der vorhandene Standard vollständig vorliegt und analysiert werden kann. Durch die genaue Analyse können interessante Codestellen der Implementierung für das Fuzzing besser gefunden werden. Die Analyse bietet auch die Gefahr, dass die Analyse der interessanten Codestellen nicht vollständig ist und somit der Fuzzer nicht effektiv arbeiten kann.

Ein Fuzzer, der speziell für eine Anwendung oder für einen beschriebenen Standard entwickelt wird, kann nicht mehr für andere Anwendungen oder Standards verwendet werden. Eine allgemeine Entwicklung eines Fuzzers nach diesem Prinzip ist somit nicht realisierbar.

Diese beiden Ansätze lassen sich sowohl auf das lokale Fuzzing als auch auf das remote Fuzzing anwenden. Die Eigenschaften der beiden Systeme und wo das jeweilige System zum Einsatz kommt, soll im Folgenden erläutert werden.

2.1.4 Lokaler Fuzzer

Das Fuzzing auf lokaler Ebene eines Computersystems fällt unter den Begriff des lokalen Fuzzers. Das Ziel des lokalen Fuzzings ist es über die lokale Eingabe von Fuzzingdaten Privilegien auf dem zu testenden System zu erlangen um somit schädlichen Code ausführen zu können. Zu den möglichen Zielen eines lokalen Fuzzers zählen zum Beispiel unter UNIX

Programme, die über “setuid” temporäre Administratorrechte bekommen. Dabei teilt sich der Oberbegriff “Lokaler Fuzzer” in die drei folgenden Unterbegriffe auf.

2.1.4.1 Kommandozeilenfuzzer

Der Kommandozeilenfuzzer versucht mutierte Eingabewerte über die Kommandozeile an Programme weiter zu geben. Für das Fuzzing bieten sich hier die Übergabeargumente der Kommandozeile an das zu testende Programm an.

Bereits entwickelte Kommandozeilenfuzzer sind clfuzz [5] und iFuzz [6].

2.1.4.2 Umgebungsvariablenfuzzer

Die zweite Unterkategorie ist der Umgebungsvariablenfuzzer. Dieser versucht über die erzeugten gefuzzten Daten, die sich in Umgebungsvariablen befinden, temporär administrative Rechte auf einem System zu bekommen.

Bereits entwickelte Umgebungsvariablenfuzzer sind Sharefuzz [7] und ebenfalls iFuzz [6].

2.1.4.3 Dateiformatfuzzer

Eine große Anzahl von Programmen müssen an irgendeinem Punkt des Programmablaufes mit dem Lesen von Dateien umgehen können. Der Dateiformat-Fuzzer zielt darauf ab, Fehler in einem Programm beim Parsen der einzulesenden Datei zu erzeugen. Ein Dateiformatfuzzer wird dynamisch verschiedene veränderte Dateien erzeugen, die anschließend vom zu testenden Programm geöffnet werden. Das Öffnen der Dateien muss dann vom jeweiligen zu testenden Programm korrekt ausgeführt werden. Falls dies nicht der Fall ist, hat man möglicherweise eine Schwachstelle des Programms entdeckt.

Bereits entwickelte Dateiformatfuzzer sind FileFuzz [8], notSPIKEfile [10] und SPIKEfile [9].

2.1.5 Remotefuzzer

Die zweite Gruppe von Fuzzern sind die Remotefuzzer. Generell lässt sich sagen, dass sobald ein Fuzzer mit seinem Ziel über einen Netzwerksocket kommuniziert, es sich um einen Remotefuzzer handelt. Auch hier gibt es die Möglichkeit Unterkategorien zu definieren.

2.1.5.1 Netzwerkfuzzing

Speziell für die spätere Anwendung des Fuzzings bei kryptographischen Protokollen spielt das Netzwerkfuzzing eine wichtige Rolle. Die bis dato existierenden Netzwerkprotokollfuzzer können in zwei Gruppen unterteilt werden. Einige von ihnen sind allgemeine Frameworks, die fähig sind verschiedene Protokolle zu fuzzen. Ein Fuzzingframework besteht aus zwei Teilen. Der erste Teil enthält eine Datenbank, die eine Vielzahl an Testdaten enthält. Der zweite Teil des Frameworks besteht aus Schnittstellen, die zur Realisierung eines Fuzzers benötigt werden. Die vorhandenen Schnittstellen können auf die Testfalldaten der Datenbank zugreifen, die Daten an gewünschter Stelle einbauen und die gefuzzten Daten an die zu testende Implementierung senden. Zu erwähnende Frameworks sind SPIKE [4] oder ProtoFuzz [11] . Zu der anderen Art von Netzwerkprotokollfuzzern zählen jene, die dafür entwickelt worden sind, ein spezielles Protokoll zu fuzzen. Beispiele hierfür sind dhcpfuzz [12] und Infigio FTPStressFuzzer [13] .

2.1.5.2 Fuzzer für Webapplikationen

Das Fuzzing auf Webapplikationen ist ein spezieller Fall des Netzwerkfuzzings. Dieser Fuzzer bezieht sich im speizellen auf das HTTP Protokoll, das von vielen Webapplikationen benutzt wird. Das Fuzzing von Webapplikationen wird deshalb extra beleuchtet, da diese Art von Anwendungen immer mehr an Bedeutung gewinnen. Das Fuzzing zielt hier vor allem darauf ab, Schwachstellen in Form von SQL Injections und Cross Site Scripting aufzudecken.

Bereits entwickelte Webapplikationsfuzzer sind: WebScarab [14] und Codenomicon HTTP Server Suite [15].

2.1.6 Generierung von Daten

Die erste Überlegung zu der Generierung von Daten beinhaltet, dass man die zu fuzzenden Eingabeparameter anhand von Dokumentationen genauer untersucht. Hierfür sind beim Netzwerkfuzzing eines bestimmtes Protokolls das RFC des jeweiligen Protokolls zu untersuchen. Bei der Untersuchung der Eingabeparameter sollen Parameter gefunden werden, die sich dazu eignen gefuzzt zu werden. Dabei handelt es sich zum Beispiel um Längenfelder, Trennzeichen oder Strings.

2.1.6.1 Erzeugung von Integerwerten

Die Integerwerte werden häuftig benutzt, um die Größe von Feldern anzugeben. Hierbei bietet es sich an, die jeweiligen Grenzen der zugelassenen Integerwerte in die Auswahl der Testfälle aufzunehmen, da dies die zu testenden Integerwerte deutlich eingrenzen würde. Wenn man als Beispiel von einer 32-bit Integerzahl ausgehen würde, dann wären die

Grenzwerte, die zu definieren wären, 0 und 0xFFFFFFFF. Unter der Berücksichtigung, dass Integer-Overflows (Additionen, die einen Umbruch bei der maximalen 32-bit Integerzahl verursachen) und Integer-Underflows (Subtraktionen, die einen Umbruch bei 0 erzeugen) in einem möglichen Sicherheitsproblem enden, wäre es klug, grenznahe Testfälle zu implementieren. Diese Testfälle wären zum Beispiel: 0xFFFFFFFF-1, 0xFFFFFFFF-2, 0xFFFFFFFF-3, ..., und 1, 2, 3, 4 und so weiter. Gleichzeitig kann ein Multiplikationsfaktor auf eine bestimmte Größe angewendet werden. Es muss der Fall in Betracht gezogen werden, dass die Fuzzingdaten in Unicode gewandelt werden. Dies würde bedeuten, dass die festgelegte Größe mit 2 multipliziert werden müsste. Darüber hinaus können zwei zusätzliche Bytes, die die NULL-Termination berücksichtigen, einbezogen werden. Daraus würden die folgenden Testfälle resultieren: 0xFFFFFFFF/2, 0xFFFFFFFF/2 -1, 0xFFFFFFFF/2 -2 und so weiter. Desweiteren können auch Grenzfälle für 16-bit (0xFFFF) große und 8-bit (0xFF) große Integerwerte berücksichtigt werden.

2.1.6.2 Erzeugung von Strings

Sobald in einem Feld, das Strings enthält, gefuzzt werden soll, bietet es sich an, eine große Anzahl an Stringwiederholungen zu generieren. Als Beispiel wäre eine lange Kette des ASCII-Zeichens 'A' zu nennen. Desweiteren sollte man in Betracht ziehen auch andere ASCII-Zeichenketten zu verwenden, da viele Programmierer in ihrem Code lange 'A' Stringketten abfangen.

2.1.6.3 Nicht alphanumerische Zeichen

Weiterhin bietet es sich an, nicht alphanumerische Zeichen, wie zum Beispiel Leerzeichen oder Tabstops zu berücksichtigen. Diese Zeichen werden oft als Trennzeichen und Schlusszeichen verwendet. Diese nicht alphanumerischen Zeichen zufällig und durchgehend in die generierten Fuzzstrings einzubauen, erhöht die Chance das Protokoll in Teilen nachzubilden. Zu den nicht alphanumerischen Zeichen gehören zum Beispiel: !@#%&*() - _ =+;:"',<.>/?

2.1.6.4 Format Strings

Die oft verwendeten Format Strings (meistens in C programmierte Programme) bieten bei Mutierung im Testfall eine große Wahrscheinlichkeit einen Fehler zu generieren. Die Daten, die von einem Fuzzer generiert werden, sollten diese Format Strings beinhalten. Fehleranfälligkeiten, die in Zusammenhang mit dem Formatstring entstehen, können theoretisch mit jedem Formatstringzeichen hervorgerufen werden. Eine bessere Auswahl von Formatstringzeichen für das Fuzzing sind entweder das %s oder das %n Zeichen. Diese Zeichen bieten eine bessere Wahrscheinlichkeit einen nachweisbaren Fehler, wie zum Beispiel eine Speicherzugriffsverletzung, zu verursachen. Durch das %s Zeichen entsteht eine

höhere Menge an Speicherlesezugriffen, da der Stack nach der Suche des NULL Bytes, das ein Stringende anzeigt, dereferenziert wird. In den meisten Fällen bietet das %n Zeichen die beste Möglichkeit einen Fehler auszulösen.

2.1.7 Fehlererkennung beim Fuzzing

Der nächste Schritt im Aufbau eines effektiven Fuzzers ist, dass eine Entscheidung getroffen werden muss, wann das Fuzzing im zu fuzzenden Programm für Probleme gesorgt hat. Die drei verschiedenen Fehlererkennungsarten, die im Folgenden beschrieben werden sollen, stellen eine Möglichkeit dar, die Fehlererkennung im Fuzzer zu realisieren.

2.1.7.1 Verbindungsüberprüfung

Der einfachste Weg zur Überprüfung der Verfügbarkeit der zu fuzzenden Resource, ist eine Verbindungsüberprüfung durchzuführen. Diese Überprüfung muss zwischen jedem Testfall durchgeführt werden. Im Beispeil eines Netzwerkprotokollfuzzers kann versucht werden eine TCP-Verbindung neu herzustellen, um so zu überprüfen, ob der Prozess auf dem Serversystem noch korrekt ausgeführt wird. Als weitere Überlegung zu der reinen Verbindungsüberprüfung, kann versucht werden, einen vorher definierten gültigen Testfall nach jedem Testfall des Fuzzers zu versenden. Somit kann genauer beobachtet werden, ob der Prozess die Anfrage korrekt verarbeitet. Dies könnte zum Beispiel ein gültiger Login, im Falle von SSH, auf dem jeweiligen Server sein.

2.1.7.2 Debugging

Die zweite Möglichkeit, die sich dem Entwickler bietet, ist, einen Debugger den Prozess überwachen zu lassen. Das Debugging verfolgt dabei den folgenden Gedanken: Auf der untersten Ebene informiert die CPU das Betriebssystem, wenn diverse Fehler, Ausnahmen oder Interrupts wie zum Beispiel Zugriffsverletzungen oder Nulldivision, auftreten.

Dabei muss der gewählte Debugger die folgenden Ausnahmen abfangen können:

- Unerlaubtes Springen der Ausführungsadresse

- Lesen von Daten, die nicht gelesen werden dürfen

- Schreiben von Daten, die nicht geschrieben werden dürfen

Damit ein Prozess durch einen Debugger auf die oben genannten Fehler überwacht wird, gibt es die folgenden Möglichkeiten: Das zu beobachtende Ziel kann entweder mit dem Debugger verbunden werden oder unter die Kontrolle des Debuggers gestellt werden. Falls der Debugger eine Exception abfängt, muss diese an den Fuzzer, der womöglich auf einem

anderen System läuft, gesendet werden. Dieser Kanal sollte einen Hin- und Rückkanal enthalten. Der Rückkanal zum Debugger wird benötigt, damit der Fuzzer signaliseren kann, wann der Debugger den zu überwachenden Prozess für den nächsten Testfall zu starten hat. Die beste Wahl dafür sind Sockets, da sie uns nicht nur erlauben unseren Fuzzer und Debugger in unterschiedlichen Programmiersprachen zu schreiben, sondern möglicherweise auch auf anderen Systemen und sogar auf verschiedenen Plattformen einsetzbar sind.

2.1.7.3 Logging

Die letzte Möglichkeit, die sich dem Entwickler bietet, sind die Logging-Dateien des jeweiligen Prozesses auszunutzen. So kann man diese auf dokumentierte Fehler während eines Testfalles untersuchen. Wenn ein Fehler auftritt muss auch hier eine Rückmeldung an den Fuzzer geschehen. Um diese Rückmeldung über die Loggingdatei des Prozesses an den Fuzzer zu realisieren, kann auch hier eine Verbindung über einen Socket realisiert werden.

2.1.7.4 Resourcenüberwachung

Auf dem zu testenden Zielsystem bietet sich weiterhin die Möglichkeit, die Auslastung des Systems zu überwachen. Die Überwachung bezieht sich im Speziellen auf das zu testende ausgeführte Programm. Hierbei ist es notwendig, einen normalen Auslastungswert des Prozesses zu kennen. Hierfür kann zum Beispiel über eine bestimme Zeit ein Mittelwert berechnet werden. Wenn nun ein fest definierter Grenzwert von dieser Normalauslastung überschritten wird, muss dies der Steuerlogik des Fuzzers gemeldet werden. Dieser kann dann eine Verbindung zwischen dem ausgeführten Testfall und der Überschreitung des Grenzwertes ziehen.

2.1.8 Logging der Testfälle

Der letzte Schritt der theoretischen Überlegung für den Aufbau eines Fuzzing-Tools ist die Dokumentation der Testfälle. Diese Dokumentation ist nötig, um zu einem späteren Zeitpunkt gewünschte Testfälle reproduzieren zu können. Die Loggingdateien der Testfälle sollten dafür eine durchlaufende, einmalig vergebene Nummer erhalten. Als Nächstes sollte die Loggingdatei enthalten, welche spezielle Nachricht eines bestimmten Protokolls oder welche Datei gefuzzed wurde. Der spezielle Nachrichtentyp des untersuchten Protokolls oder der jeweilige Dateityp bieten sich an, um eine Baumstruktur für das Logging einzubinden. Dazu soll jeder Nachrichtentyp der Übersicht wegen einen eigenen Ordner enthalten, der die jeweiligen Testfälle enthält. Der zweite Punkt, der wichtig für ein sinnvolles Logging ist, beinhaltet den Mitschnitt der verschickten und veränderten Pakete des Fuzzers, so dass nach einem durchgeführten Test genau nachvollzogen werden kann, was

zu einem Fehler geführt hat, damit dieser zu jedem Zeitpunkt wieder reproduziert werden kann. Der letzte Punkt des Loggings eines Fuzzingtestfalles enthält die Information, ob der gefuzzte Wert zu einem Abstürzen/Fehler des Programms geführt hat oder nicht. Hier kann am Beispiel eines Netzwerkfuzzers die gültige Verbindungsüberprüfung oder das Logging des zu testenden Prozesses mit eingepflegt werden.

2.2 Das SSH-Protokoll

Das SSH-Protokoll (SecureShell-Protokoll) soll in diesem Kapitel behandelt werden, da eine Implementierung gefuzzt werden soll, die zur Komunikation dieses Protokoll nutzt. Das SSH-Protokoll wurde 1995 von Tatu Ylönen entwickelt [1], um Dienste wie zum Beispiel telnet, das unverschlüsselt kommuniziert, abzulösen. Das SSH-Protokoll bietet die Möglichkeit, über einen unsicheren Kommunikationskanal eine sichere Verbindung zwischen zwei Kommunikationspartnern aufzubauen. Die erste Version des SSH-Protokolls nannte sich SSHv1. Darauf folgte 1996 das SSH-Protokoll in der Version 2. Das Verständnis des SSH-Protokoll, das in diesem Kapitel gelegt werden soll, dient dem Zweck, mögliche Testfälle für das Fuzzing Tool zu entwerfen. Dazu wird zuerst ein Einblick in die verwendeten Datentypen des SSH-Protokolls gegeben.

2.2.1 Datentypen des SSH-Protokolls

Die verschiedenen definierten und verwendeten Datentypen im SSH-Protokoll legen die Grundlage der Kommunikation. Die Datentypen werden dazu verwendet, um spätere Nachrichtentypen genau zu beschreiben. Im Folgenden sollen alle möglichen Datentypen des SSH-Protokolls anhand des RFC 4251 [18] aufgezeigt werden.

2.2.1.1 Der Datentyp *byte*

Der Datentyp *byte* speichert einen beliebigen 8bit Wert. Wenn dieser Datentyp in einem Array auftritt, kommt die folgende Schreibweise zum Einsatz: byte [n]. Dabei gibt n die Anzahl der Bytes an.

2.2.1.2 Der Datentyp *boolean*

Dieser Datentyp wird in einem einzelnen Byte gespeichert. Der Wert null repräsentiert FALSCH. Alle Werte, die nicht den Wert null haben, müssen als WAHR interpretiert werden.

2.2.1.3 Der Datentyp *unit32*

Der Daten typ *uint32* repräsentiert einen vorzeichenfreien 32-bit Integerwert. Dieser wird mit fallender Signifikanz in vier Byte abgespeichert (Netzwerkbytefolge). Dazu ein Beispiel: Der Wert 699921578 (0x69b7f4aa) wird folgendermaßen abgespeichert: 69 b7 f4 aa.

2.2.1.4 Der Datentyp *unit64*

Der Datentyp *uint64* repräsentiert einen vorzeichenfreien 64-bit Integerwert. Dieser wird mit fallender Signifikanz in acht Byte abgespeichert (Netzwerkbytefolge).

2.2.1.5 Der Datentyp *string*

Ein String kann beliebige Zeichen, die im 8-bit Format dargestellt werden, enthalten. Der String besteht zum einen aus einem unit32 Datentyp, der die Länge des Strings (Anzahl der Bytes, die nach der Länge folgen) enthält und zum anderen aus den Bytes, die die Werte des Strings enthalten. Das terminierende null-Zeichen wird nicht im String verwendet.

Strings werden dazu benutzt, um Textinformationen zu speichern. Diese werden mit dem Kodierverfahren ASCII kodiert. Zur Veranschaulichung dient das folgende Beispiel: Der ASCII String “testing” wird durch 00 00 00 07 (Längenangabe) testing (String) repräsentiert.

2.2.1.6 Der Datentyp *mpint*

Der Datentyp mpint repräsentiert einen Integerwert im 2er- Komplementformat, der als String gespeichert wird. Dabei gilt es zu beachten, dass die Most Signifikant Bitorder angewendet wird. Negative Zahlen haben den Wert eins an der Stelle des höchsten Bits des ersten Bytes. Falls das höchste Bit eine positive Zahl anzeigt, muss der Zahl ein null Byte vorausgestellt sein. Führende Bytes mit den Werten 0 oder 255 dürfen nicht beinhaltet sein. Der Wert null muss als String mit 0 Bytes an Daten abgebildet werden.

2.2.1.7 Der Datentyp *name-list*

Die name-list ist ein Datentyp, der eine Liste von Strings enthält, die durch ein Komma getrennt werden. Eine name-list wird von einem unit32 Wert, der die Länge angibt (Anzahl der Bytes, die folgen), und von einer durch Komma getrennten Liste von Nullen oder mehreren Namen (Strings), repräsentiert. Ein Element darf nicht von der Länge null sein und es darf keine Kommas enthalten. Da dies eine Liste an Namen ist, müssen alle enthaltenen Elemente im ASCII Format kodiert sein. Abschließende Nullzeichen dürfen nicht verwendet werden - weder für die einzelne Elemente noch für die gesamte Liste.

2.2.2 Das SSH-Transportprotokoll

Unter dem SSHv2-Protokoll verbergen sich drei verschiedene Protokolle. Diese heißen SSH-Transportprotokoll [19], SSH-Authentikationsprotokoll [20] und SSH-Verbindungsprotokoll [21]. Das SSH-Transportprotokoll baut auf dem TCP/IP Stack des OSI-Referenzmodells auf. Das Protokoll kann als eine Grundlage für verschiedene sichere Netzwerkdienste verwendet werden. Es bietet jeweils eine starke Verschlüsselung, Server Authentifikation und Datenintegrität. Im folgenden sollen jeweils die Grundlagen zu den drei SSH-Protokollen gelegt werden.

2.2.2.1 Aufbau SSHv2-Binärpakete

Die Binärpakete bilden die Grundlage einer jeden SSH-Transportprotokollnachricht. Jedes Paket hat den in Tabelle 2.1 beschriebenen Aufbau.

Tab. 2.1: Aufbau Binärpaket[19]

Datentyp	Name
uint32	packet_length
byte	padding_length
byte[n1]	$payload; n1 = packet\_length - padding\_length - 1$
byte[n2]	$randompadding; n2 = padding\_length$
byte[m]	$mac(MessageAuthenticationCode - MAC)$; $m = mac\_length$

packet_length

Gibt die Länge des Paketes in Byte an, berücksichtigt aber nicht das 'mac'- oder das 'packet_length'-Feld.

padding_length

Gibt die Länge des zufälligen Paddings in Byte an.

payload

Enthält den Nutzdaten des Paketes.

random_padding

Die zufällige Paddinglänge sollte so gewählt werden, dass die totale Länge bestehend aus (packet_length||padding_length||payload||random_padding) ein Vielfaches der Cipherblocklänge oder von acht ist - welches von beiden jeweils größer ist. Es müssen mindestens vier Bytes an Padding enthalten sein. Das Padding sollte aus zufälligen Bytes bestehen. Die maximale Anzahl des Paddings sind 255 Byte.

mac

Falls die Nachrichtenauthentifikation verwendet wird, enthält dieses Feld den Message Authentication Code. Bei Beginn der Verbindung muss der MAC Algorithmus auf 'none' gesetzt sein.

2.2.2.2 Maximale Paketlänge

Alle Implementationen müssen imstande sein, Pakete im nicht komprimierten Zustand mit der Payloadgröße von 32768 Bytes oder weniger und mit der totalen Paketlänge von 35000 Bytes oder weniger zu verarbeiten.

2.2.2.3 Die Verschlüsselung im Transportprotokoll

Die Verschlüsselung der unsicheren Leitung, über die das SSH-Protokoll kommuniziert, übernimmt das Transportprotokoll. Der Verschlüsselungsalgorithmus und der dazugehörige Schlüssel werden während des Schlüsselaustausches verhandelt. Falls die Verschlüsselung angewendet wird, müssen die Paketlänge, die Paddinglänge, die Payload- und die Paddingfelder mit dem angewendeten Algorithmus verschlüsselt sein.

Die verschlüsselten Daten sollten in allen Paketen, die in eine Richtung gesendet werden, als ein einziger Datenstream betrachtet werden. Zudem sollten alle Chiffren Schlüssel verwenden, die eine effektive Schlüsselänge von mindestens 128 Bit vorweisen. In der Praxis wird empfohlen, dass der gleiche Algorithmus für beide Richtungen verwendet wird.

Die in Tabelle 2.2 beschriebenen Chiffren sind zum einen ein Auszug aus dem RFCs 4253, 5647 und zum anderen aus der Implementierung OpenSSH Version 6.6.1.

2.2.2.4 Die Datenintegrität des Transportprotokolls

Die Integrität der Daten wird geschützt, indem in jedem verschlüsselten Paket ein MAC enthalten ist. Diese wird aus dem Schlüssel, der Paketnummer und dem Inhalt des Paketes berechnet. Beim Verbindungsaufbau wird noch keine MAC verwendet. Bei diesem muss die Länge der MAC null sein. Nach dem Schlüsselaustausch wird die MAC nach dem gewählten MAC-Algorithmus wie folgt berechnet:

$$mac = MAC(key, sequence\_number||unencrypted\_packet)$$

Das Ergebnis des MAC-Algorithmus muss unverschlüsselt als letzter Teil des Paketes übermittelt werden. Die Anzahl der MAC Bytes hängen vom gewählten Algorithmus ab. Die

Tab. 2.2: Verschlüsselungschiffren SSH [19]

Verschlüsselung	Beschreibung
3des-cbc	3DES im CBC-Modus
blowfish-cbc	Blowfish im CBC-Modus
aes256-ctr	AES im CTR-Modus mit einem 256 Bit Schlüssel
aes192-ctr	AES im CTR-Modus mit einem 192 Bit Schlüssel
aes128-gcm	AES im GCM-Modus mit einem 128 Bit Schlüssel
aes256-gcm	AES im GCM-Modus mit einem 256 Bit Schlüssel
cast128-cbc	CAST-128 im CBC Modus
none	keine Verschlüsselung

Algorithmen, die in Tabelle 2.3 beschrieben sind, sind zum einen ein Auszug aus dem RFC 4253 und zum anderen aus der Implementierung OpenSSH Version 6.6.1.

Tab. 2.3: Algorithmen Datenintegrität [19]

MAC-Algorythmus	Beschreibung
umac-128-etm@openssh.com	Universal Message Authentication Code mit einer Taglänge von 128
hmac-md5-etm@openssh.com	Hash Message Authentication Code auf Basis von MD5 Algorithmus
hmac-sha1-etm@openssh.com	Hash Message Authentication Code auf Basis des SHA1 Algorithmus
hmac-sha2-256-etm@openssh.com	Hash Message Authentication Code auf Basis des SHA2 Algorithmus
none	keine MAC

2.2.2.5 Die Methoden zum Schlüsselaustausch

Die Methoden zum Schlüsselaustausch beschreiben wie einmalige Sitzungschlüssel zur Verschlüsselung, Clientauthentifikation und zur Serverauthentifikation generiert werden.

Dazu wurden die in Tabelle 2.4 zur Verfügung stehenden Schlüsselaustauschverfahren, die zum einen ein Auszug aus dem RFC 4253 und zum anderen ein Ausschnitt aus der Implementierung von SSH in OpenSSH Version 6.6.1 sind, definiert:

Tab. 2.4: Schlüsselaustauschverfahren[19]

Algorithmus	Beschreibung
curve25519-sha256@libssh.org	empfohlen
diffie-hellmann-group14-sha1	erforderlich

Der Schlüsselaustausch (KEX) startet, sobald sich beide Seiten die unterstützten Algorithmen zugesandt haben. Jede Seite hat einen präferierten Algorithmus für jede Kategorie. Es wird angenommen, dass die meisten Implementationen zu jedem Zeitpunkt den gleichen bevorzugten Algorithmus verwenden. Jede Seite darf versuchen, den Algorithmus der jeweiligen anderen Seite zu erraten und darf eine Initial Key Exchange Nachricht, die auf den ausgewählten Algorithmen basiert, verschicken. Wenn die Vermutung jedoch falsch ist, muss die jeweils andere Seite das korrekte Initialpaket senden.

Das Paket, das den Schlüsselaustausch einleitet, hat den in Tabelle 2.5 defnierten Aufbau.

Nachdem die Nachrichten SSH_MSG_KEXINIT gesendet wurden, startet der SSH-Client den Schlüsselaustausch mit dem Versenden der Nachricht SSH_MSG_DIFFIE_HELLMAN_KEY_EXCHANGE_INIT. Der SSH-Server reagiert auf diese Nachricht mit der Nachricht SSH_MSG_DIFFIE_HELLMAN_KEY EXCHANGE_REPLY. Der Schlüsselaustausch gilt als erfolgreich, wenn beide Seiten die Nachricht SSH_MSG_NEWKEYS versenden. Diese Nachricht wird entweder verschlüsselt (aktive Verschlüsselung) oder im Klartext (erster Verbindungsaufbau) versendet. Alle Nachrichten, die danach versendet werden, müssen mit den neuen Schlüsseln gesichert werden. Falls etwas beim Schlüsselaustausch falsch abläuft, gibt es die Möglichkeit, dass die Verbindung mit der Nachricht SSH_MSG_DISCONNECT getrennt wird.

Tab. 2.5: Init Schlüsselaustausch [19]

Datentyp	Beschreibung
byte	*SSH_MSG_KEXINIT*
byte [16]	Cookie (zufälle Daten)
name-list	kex_algorithm
name-list	server_host _key_algorithm
name-list	encryption_algorithms_client_to_server
name-list	encryption_algorithms_server_to_client
name-list	mac_algorithms_client_to_server
name-list	mac_algorithms_server_to_client
name-list	compression_algorithms_client_to_server
name-list	compression_algorithms_server_to_client
name-list	languages_client_to_server
name-list	languages_server_to_client
boolean	first_kex_packet_follows
uint32	0 (reserviert für Erweiterungen)

Die Nachricht SSH_MSG_NEWKEYS hat den in Tabelle 2.6 beschriebenen Aufbau:

Tab. 2.6: Aufbau Schlüsselaustausch erfolgreich[19]

Datentyp	Beschreibung
byte	SSH_MSG_NEWKEYS

2.2.2.6 Die SSH-Serverauthentisierung

Bevor der Server und der Client den Schlüsselaustausch vollziehen, muss sich der Server mit einer Signatur gegenüber dem Client authentifizieren. Der Benutzer des Clients bekommt den Schlüssel bei erstmaliger Verbindung angezeigt. Er muss dann die Signatur des Servers prüfen und bei Richtigkeit bestätigen.

2.2.2.7 Der Schlüsselaustausch mit dem Diffie-Hellmann Verfahren

Der Diffie-Hellmann Algorithmus ist eine Variante im SSH-Protokoll die Schlüssel auszutauschen. Seine Sicherheit basiert auf der Problematik, dass es keine eindeutigen, sichere mathematische Verfahren gibt den diskreten Logarithmus zu berechnen, es aber umgekehrt einfach ist, eine Zahl zu potenzieren. Um das Verfahren anschaulich zu erläutern,

werden im Folgenden vereinfachte Zahlenbeispiele verwendet. In diesem Beispiel sollen die Kommunikationspartner Alice und Bob genannt werden.

1. Einer der beiden Kommunikationspartner legt eine möglichst große Primzahl P und eine Zahl z, die kleiner als P sein muss, fest. Dies geschieht im SSH Protokoll unter der Nachricht Diffie-Hellman Group Exchange Group unter dem "diffie-hellmann-group14-sha1" Algorithmus. Diese beiden Zahlen werden nun an den anderen Kommunikationspartner mit der Nachricht SSH_DIFFIE_HELLMAN_GROUP_EXCHANGE_GROUP gesendet. Wird als Algorithmus Elliptic Curve Diffie-Hellman verwendet, ist diese Primzahl per Definiton den beiden Kommunikationspartnern bekannt und muss nicht über das Netzwerk versandt werden.

2. Jeder der beiden Kommunikationspartner überlegt sich eine geheime Zahl. Alice wählt die geheime Zahl a und Bob die geheime Zahl b. Alice berechnet nun:

$$X = z^a mod P$$

und Bob berechnet:

$$Y = z^b mod P$$

Die errechneten Zahlen X und Y werden nun jeweils an den anderen Kommunikationspartner geschickt. Dies entspricht im SSH Protokoll den Nachrichten SSH_DIFFIE_HELLMAN_KEY_EXCHANGE_INIT und SSH_DIFFIE_HELLMANN_KEY_EXCHANGE_REPLY. Alice und Bob berechnen nun daraus den geheimen Schlüssel. Alice berechnet diesen wie folgt:

$$K_a = Y^a mod P$$

und Bob wie folgt:

$$K_b = X^b mod P$$

Damit erhalten beide den gleichen Wert K_a und K_b, denn es gilt:

$$(z^b mod P)^a mod P = (z^a mod P)^b mod P = z^{(ab)} mod P$$

Damit haben nun die Kommunikationspartner Alice und Bob einen geheimen Schlüssel, den außer ihnen keiner kennt. Auf diesem geheimen Schlüssel basierend werden die Schlüssel zur Verschlüsselung und zur Integrität der Daten berechnet.

Der heute zum großen Teil verwendete Elliptic Curve Schlüsselaustausch Algorithmus entspricht im Wesentlichen dem Ablauf des Diffie Hellman Algorithmus. Der Unterschied der beiden Algorithmen ist, dass ECDH das gleiche Maß an Sicherheit jedoch mit kürzeren Schlüsseln erreicht, da er auf elliptischen Kurven beruht.

Eine Man-in-the-Middle Attacke auf den Schlüsselaustausch ist generell möglich, sobald der Angreifer Datenpakete von Bob und Alice verändern kann. Somit könnte der Angreifer zweimal einen Schlüsselaustausch durchführen. Dabei denken Bob und Alice fälschlicherweise, dass sie einen einzigen Schlüsselaustausch miteinander durchgeführt haben. Die Absicherung gegen eine Man-in-the-Middle Attacke ist, dass die ausgetauschten Daten authentisiert sein müssen. Dies geschieht im Falle von SSH durch die SSH-Serverauthentisierung.

2.2.3 Das SSH-Authentikationsprotokoll

Das SSH-Authentikationsprotokoll wird dazu genutzt, damit sich ein Client gegenüber einem Server authentifizeren kann. Die im folgenden beschriebenen Informationen basieren auf den Informationen aus dem RFC 4252 [20]. Der Server startet die Authentifikation, indem er den Client mitteilt welche Methoden er verwenden kann. Dabei hat der Client die Freiheit, jede der vorgegebenen Methoden in seiner gewollten Reihenfolge aus zu testen. Die Authentifikationsmethoden werden hierbei durch ihre Namen identifiziert. Die Methode 'none' darf nicht als eine unterstützte Methode aufgeführt werden.

2.2.3.1 Die Authentifikationsanfrage

Alle Authentifikationsanfragen müssen das in Tabelle 2.7 beschriebene Format haben. Nur die ersten vier Felder sind definiert. Die übrigen Felder hängen von der jeweiligen gewählten Authentifikationsmethode ab.

Tab. 2.7: Authentifizierung Aufbau Request [20]

Datentyp	Beschreibung
byte	*SSH_MSG_USERAUTH_REQUEST*
string	user name in UTF-8 kodiert
string	service name
string	method name
...	methodenspezifische Felder

Die in Tabelle 2.8 dargestellten Methoden stehen nach RFC 4252 zur Authentifizierung zur Verfügung.

Tab. 2.8: Authentifizierung Methoden [20]

Methode	Implementierung
“publickey”	VERLANGT
“password”	OPTIONAL
“hostbased”	OPTIONAL
“none”	NICHT EMPFOHLEN

2.2.3.2 Die Authentifikationsantwort

Falls der Server die Authentifikationsanfrage ablehnt, muss er wie in Tabelle 2.9 antworten:

Tab. 2.9: Authentifizierung Aufbau Fehler [20]

Datentyp	Beschreibung
byte	SSH_MSG_USERAUTH_FAILURE
name-list	authentications mit denen weiter gemacht werden kann
boolean	partial success

Wenn der Server die Authentifikationsanfrage annimmt, muss er wie in Tabelle 2.10 beschrieben antworten:

Tab. 2.10: Authentifizierung Aufbau Erfolg [20]

Datentyp	Beschreibung
byte	SSH_MSG_USERAUTH_SUCCESS

Die darauf folgenden SSH_MSG_USERAUTH_REQUEST Nachrichten sollten vom Server ignoriert werden.

2.2.3.3 Die Authentifikationsmethode “Public Key”

Alle Implementationen müssen die Public Key Methode unterstützen. Bei dieser Methode dient der Besitz eines Private Keys als Authentifikation. Zur Authentifikation wird eine Signatur aus dem private Key des Clients erstellt und an den Server gesendet. Der Server muss prüfen, ob der Schlüssel eine gültige Echtheitsbestätigung des Benutzers ist und ob die Signatur gültig ist. Falls beide gültig sind, muss die Authentifikationsanfrage angenommen werden, falls nicht, muss sie abgelehnt werden. Private Keys werden oft in verschlüsselter Form auf dem Client gespeichert und der Benutzer muss meistens einmalig eine Passphrase angeben, bevor der Schlüssel verwendet werden kann.

Um zu prüfen, ob der Server die Public Key Methode unterstützt, kann die in Tabelle 2.11 gezeigte Nachricht vom Client gesendet werden:

Tab. 2.11: Public Key AuthentifizierungRequest[20]

Datentyp	Beschreibung
byte	SSH_MSG_USERAUTH_REQUEST
string	user name
string	service name
string	”publickey”
boolean	FALSE
string	public key algorithm name
string	public key blob

Wenn der Server den angefragten Algorithmus nicht unterstützt, muss er die Anfrage ablehnen. Der Server muss auf die Anfrage entweder mit SSH_MSG_USERAUTH_FAILURE oder mit der SSH_MSG_USERAUTH_PK_OK (Tabelle 2.12) Nachricht reagieren.

Tab. 2.12: Public Key Authentifizierung OK [20]

Datentyp	Beschreibung
byte	SSH_MSG_USERAUTH_PK_OK
string	public key algorithm name from the request
string	public key blob from the request

Die Clientauthentifikation wird durch das Versenden der Signatur an den Server durchgeführt. Diese Signatur wird aus dem Private Key generiert. Die Nachricht hat den in Tabelle 2.13 gezeigten Aufbau:

Tab. 2.13: Public Key Authentifizierung Aufbau [20]

Datentyp	Beschreibung
byte	SSH_MSG_USERAUTH_REQUEST
string	user name
string	service name
string	”publickey”
boolean	TRUE
string	public key algorithm name
string	signature

Wenn der verwendete Publickey Algorithmus in der Authentifikation SSH-RSA ist, dann hat der übertragene Schlüssel das in Abbilung 2.14 gezeigte Format:

Tab. 2.14: SSH-RSA Schlüsselformat [19]

Datentyp	Beschreibung
string	ssh-rsa
mpint	e (exponent)
mpint	n (modulus)

Wenn die Prüfung der Signatur erfolgreich war, muss der Server mit der Nachricht SSH_MSG_USERAUTH_SUCCESS reagieren. Falls die Prüfung der Signatur nicht erfolgreich war, mus der Server die Nachricht SSH_MSG_USERAUTH_FAILURE versenden.

2.2.3.4 Die Authentifikationsmethode “password”

Es besteht die Möglichkeit, sich im SSH-Protokoll gegenüber dem Server mit einem Benutzer und einem zugehörigen Passwort zu authentifizieren. Die Passwortauthentifikation benutzt das in Tabelle 2.15 beschriebene Paket. Alle Implementierungen sollten die Passwortauthentifizierung unterstützen.

Der Server antwortet auf die Anfrage des Clients bei Erfolg mit der Nachricht SSH_MSG_USERAUTH_SUCCESS oder bei Scheitern mit der Nachricht SSH_MSG_USERAUTH_FAILURE.

Tab. 2.15: Passwortauthentifizierung Aufbau [20]

Datentyp	Beschreibung
byte	SSH_MSG_USERAUTH_REQUEST
string	username
string	service_name
string	”password”
boolean	FALSE
string	plaintext password in UTF-8 kodiert

2.2.4 Das SSH-Verbindungsprotokoll

Der dritte Layer des SSH-Standards nennt sich SSH-Verbindungsprotokoll. Es wurde so entworfen, dass es über dem SSH-Tansport- und Authentikationprotokoll arbeitet. Es bietet interaktive Loginsitzungen, Kommandoausführung über Remotezugriff, Weiterleitung von TCP/IP Verbindungen, Weiterleitung von X11 Verbindungen. Auf das SSH-Verbindungsprotokoll soll nicht weiter eingegangen werden, da die Dienste, die das SSH-Verbindungsprotokoll bietet, keinerlei kryptographische Funktionen beinhalten. Somit hat das SSH-Verbindungsprotokoll keine weitere Relevanz für das spätere kryptographische Fuzzing.

2.2.5 Die Message IDs

Abschließend sollen alle relevanten Message IDs des SSH-Protokolls in Tabelle 2.16 zusammengefasst werden. Die Nachrichten stammen hierbei aus den Transport- und Authentikationschichten des SSH-Standards.

Tab. 2.16: Nachrichten-IDs SSH [22]

Message ID	Wert
SSH_MSG_DISCONNECT	1
SSH_MSG_IGNORE	2
SSH_MSG_UNIMPLEMENTED	3
SSH_MSG_DEBUG	4
SSH_MSG_SERVICE_REQUEST	5
SSH_MSG_SERVICE_ACCEPT	6
SSH_MSG_KEXINIT	20
SSH_MSG_NEWKEYS	21
SSH_MSG_USERAUTH_REQUEST	50
SSH_MSG_USERAUTH_FAILURE	51
SSH_MSG_USERAUTH_SUCCESS	52
SSH_MSG_USERAUTH_BANNER	53

2.3 Anforderungen und Möglichkeiten des kryptographischen Fuzzings

Aus den beiden Kapiteln des Fuzzings und des Aufbaus des SSH-Protokolls sollen nun allgemeine Möglichkeiten und Anforderungen für das kryptographische Fuzzing abgeleitet werden. Die Anforderungen und Möglichkeiten des kryptographischen Fuzzings sollen in Bezug auf den Austausch der Algorithmen, den Schlüsselaustausch, die Vertraulichkeit und Integrität der Daten und die Authentisierung des zu fuzzenden kryptographischen Protokolls erarbeitet werden.

Die Nachrichten des kryptographischen Netzwerkprotokolls, die zur schlussendlichen Anwendung der kryptographischen Funktionen führen, sind als Fuzzingwerte von Interesse. Dabei handelt es sich um Nachrichten, die die anwendbaren Algorithmen zwischen zwei Kommunikationspartnern aushandeln. Der interessante Teil des Fuzzings liegt in dem Abschnitt der jeweiligen Implementierung des verwendeten kryptographischen Protokolls, dass die Herstellung einer sicheren Verbindung gewährleistet. Zur Herstellung der Vertraulichkeit und Integrität der Daten wird auch der Schlüsselaustausch im Protokoll benötigt. Dieser Teil des Protokolls bietet sich ebenfalls an, gefuzzed zu werden. In diesen beiden Teilen der Implementierung sollte versucht werden, eine größtmögliche Codeabdeckung durch das Fuzzing zu erreichen.

Sobald ein kryptographisches Protokoll eine sichere Verbindung hergestellt hat, ist ein Punkt erreicht, ab dem das kryptographische Fuzzing abgeschlossen ist. Das Fuzzing von Nachrichten, die nur verschlüsselte und durche eine Prüfsumme gesicherte Daten enthält, würde zum sofortigen Abbruch der Verbindung führen und die gefuzzte Nachricht würde zu keiner weiteren Codeabdeckung der Implementierung führen. Die fuzzbaren Teile werden in Abbildung 2.2 nochmals dargestellt.

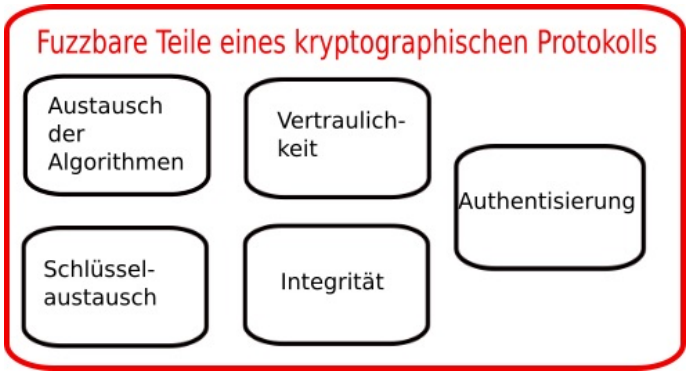

Abb. 2.2: Fuzzbare Teile eines kryptographischen Fuzzings

2.3.1 Austausch der Algorithmen

Der erste Schritt beim Aufbau einer sicheren Verbindung über ein kryptographisches Protokoll ist der Austausch der möglichen Algorithmen. Diese Algorithmen werden dazu genutzt, um den Schlüsselaustausch, die Verschlüsselung und Integrität der Daten und die Authentisierung umzusetzen. Die möglichen Algorithmen werden im Klartext in der jeweiligen verantwortlichen Nachricht des Protokolls übertragen. Ein Algorithmus selbst wird als String in einer Liste von Strings übertragen. Die erste Möglichkeit, die sich einem Fuzzer bietet, wäre das Verändern dieser Strings. Als Beispiel wird ein String mit dem Inhalt *curve-25519-sha256@libssh.org* übertragen. Ein Fuzzer könnte nun den String mit folgendem Inhalt bereitstellen: *curve–5519-sha256@libssh.org*. Dabei gilt es zu testen, ob die Implementierung des Protokolls diesen Fehler erkennt und diesen Algorithmus zur Schlüsselgenerierung nicht verwendet.

Die verschiedenen Algorithmen, die sich in den Listen befinden, werden häufig durch ein bestimmtes Zeichen voneinander getrennt. Hierbei bietet sich die Möglichkeit, dass der Fuzzer versucht, eine Manipulation dieser Trennzeichen vorzunehmen. Im SSH-Protokoll werden zum Beispiel für die Trennung der Algorithmen Kommas verwendet. Hier kann versucht werden, mehrere Kommas hintereinander einzufügen.

Vor jedem Feld für die verschiendenen Algorithmen ist eine Längenangabe der enthaltenen Zeichen. Diese Längenangabe kann durch den Fuzzer verändert werden. Dadurch wird geprüft, ob die zu testende Implementierung die Angabe der Länge auch mit der tatsächlichen Länge des Strings vergleicht. Der anzuwendende Datentyp für die Länge ist meist ein Integerwert. Dieser sollte in den Grenzbereichen der maximal und minimal definierten Größe liegen.

Sobald ein Nachrichtentyp, der im Ablauf der Verbindung nach dem Schlüssaustausch folgt, gefuzzten werden soll, muss die Nachricht, die den Schlüsselaustausch beinhaltet, unverändert weitergeleitet werden. Wenn dies nicht geschehen würde, hätte dies zur Folge, dass man in diesem Testfall gar nicht erst zu dem Punkt kommt an dem man die gefuzzten Daten einfügen will. Die zu testende Implementierung würde die Verbindung zum Fuzzer trennen, bevor die gefuzzten Daten an die Implementierung gesendet werden könnten.

2.3.2 Schlüsselaustausch

Nachdem die möglichen Algorithmen ausgetauscht sind, wird durch den gewählten Algorithmus der Schüsselaustausch vollzogen. Aus dem berechneten Schlüssel werden dann die Schlüssel zur Verschlüsselung der Daten und zur Integritätssicherung berechnet.

In den meisten Fällen wird zum Schlüsselaustausch der Elliptic Curve Diffie-Hellmann Algorithmus verwendet. Hierbei wird die Länge der öffentlichen Zahl und die öffentliche Zahl selbst in den Nachrichten zwischen Client und Server ausgetauscht. Das Längenfeld kann hierbei durch Veränderung der Zahl in den Grenzbereichen getestet werden. Die Zahl selbst ist ebenfalls ein Ziel des Fuzzers. Hierbei kann versucht werden Trennzeichen,

Steuerungszeichen und Integerwerte als Fuzzingdaten einzubauen. Zudem sollte versucht werden, möglichst viele Algorithmen des Schlüsselaustausches zu testen, da so die Codeabdeckung durch den Fuzzer weiter erhöht werden kann.

Die Anforderungen an den Fuzzer, der versucht im Ablauf nach dem Schlüsselaustausch Nachrichten zu fuzzen, darf die Nachrichten des Schlüsselaustausches nicht verändern. Wenn dies geschehen würde, wäre mit hoher Wahrscheinlichkeit ein Verbindungsabbruch die Folge.

2.3.3 Vertraulichkeit

Der nächste Schritt zum Aufbau einer sicheren Verbindung über ein kryptographisches Protokoll ist die Herstellung der Vertraulichkeit der Verbindung. Dies geschieht bei den meisten kryptographischen Protokollen über die Verschlüsselung der zu sendenden Daten. Die einzige Möglichkeit für den Fuzzer ist, das Längenfeld einer verschlüsselten Nachricht zu fuzzen. Hierdurch kann geprüft werden, ob der Verschlüsselungsalgorithmus die angegebene Länge der veschlüsselten Daten mit der Menge der verschlüsselten Daten prüft. Die verschlüsselten Daten selbst zu testen, ist für einen Fuzzer nicht möglich, da sonst die Verbindung sofort beendet werden würde.

Die Anforderung an den Fuzzer, der versucht im Ablauf nach Aktivierung der Verschlüsselung folgendene Nachrichten zu fuzzen, ist, dass eine Berechnung der jeweiligen Keys zur Verschlüsselung innerhalb diesem statt finden muss oder die benötigten Keys der Session dem Fuzzer vorliegen. Der Fuzzer muss erkennen wann ein Protokoll von dem unverschlüsselten Zustand in den verschlüsselten Zustand wechselt. So kann der Fuzzer dennoch im verschlüselten Zustand die zu fuzzende Nachricht erkennen, diese anschließend bearbeiten und wieder korrekt verschlüsseln.

2.3.4 Integrität

Die Integrität der Daten ist ein wichtiger Bestandteil von kryptographischen Protokollen. Daher muss auch auf diesen Teil in Bezug auf das Fuzzing Anforderungen und Möglichkeiten formuliert werden. Die Integritätsprüfsumme im Fuzzer zu verändern bietet sich hier nicht an, da bei Veränderung der MAC der Kommuniktationspartner die Integrität der Daten nicht mehr verifizieren kann und somit die Verbindung abbrechen würde.

Die Anforderung an den Fuzzer, der versucht Nachrichten zu fuzzen, die mit einem Schutz zur Integrität der Daten versehen sind, ist, dass in der Implementierung eine Berechnung des Schlüssels, der zur Integritätssicherung benötigt wird, stattfindet. Sobald ein Paket durch den Fuzzer manipuliert wird, muss durch eine Funktion die Integritätsprüfsumme neu berechnet werden und in das Paket eingebaut werden.

2.3.5 Authentisierung

Eine weitere Möglichkeit, ein kryptographisches Protokoll zu fuzzen ist, manipulierte Daten in Nachrichtentypen einzubauen, die sich mit der Authentifizeriung eines Clientsystems befassen. Zur Authentifizierung eines Clients gegenüber eines Servers werden bei kryptographischen Protokollen meistens Public-Key Verfahren oder Passwortverfahren verwendet. Hierbei ergibt sich die Möglichkeit für einen Fuzzer Felder bezüglich des Passwortes und des Public Keys zu manipulieren.

Die Anforderungen an den Fuzzer für im Ablauf folgende Nachrichtentypen erübrigt sich, da nach der Authentisierung keine weiteren zu fuzzenden Zustände auftreten. Denn sobald eine sichere Verbindung zwischen zwei Kommunikationspartnern hergestellt worden ist, kommen häufig Dienste zum Einsatz, die keinerlei kryptographische Eigenschaften beinhalten.

2.3.6 Fazit

Das Fuzzing eines kryptographischen Protokolls lässt sich in zwei verschiedene Teile trennen. Der erste Teil beinhaltet die Nachrichten, die unverschlüsselt über das Netzwerk geschickt werden. Hierzu zählen die Nachrichten die den Austausch der Algorithmen und den Schlüsselaustausch bewerkstelligen. In diesem Teil des Fuzzing bietet es sich an, Fuzzdaten aus den Bereichen der Integerwerte, Trennzeichen und Steuerungszeichen zu verwenden.

Der zweite Teil des Fuzzings eines kryptographischen Protokolls umfasst das Fuzzen der Authentifikationphase. Hier kann versucht werden, Passwörter, Public-Keys oder Zertifikate mit mutierten Werten zu verändern. Hierfür bieten sich vorallem wieder die Trennzeichen und Steuerungszeichen an. Dabei muss beachtet werden, dass die verschlüsselte Nachricht entschlüsselt und nach der Mutation der zu fuzzenden Nachricht mit einer neuen MAC verschlüsselt wird. Zusätzlich kann versucht werden, in verschlüsselten SSH- Nachrichten das Längenfeld der Nachricht zu fuzzen.

Die Veränderung einer verschlüsselten Nachricht oder die Veränderung der MAC einer Nachricht ergibt wenig Sinn, da dies mit höchster Wahrscheinlichkeit zu einem Verbindungsabbruch führen würde und keine weitere Codeabdeckung dadurch erzielt werden würde.

Die Anforderung an das Fuzzen eines kryptographischen Protokolls ist, dass in einem Testfall jeweils nur ein Nachrichtentyp verändert werden kann.

3 Entwurf

In dieser Bachelorarbeit soll der Fokus auf den Entwurf und die spätere Implementierung eines Netzwerkfuzzers gelegt werden. Dieser Fuzzer arbeitet mit vordefinierten Testfällen nach dem generationsbasierten Prinzip. Der SSH-Fuzzer soll in dieser Bachelorarbeit als Prototyp entworfen werden. Zu einem späteren Zeitpunkt kann versucht werden, den Entwurf oder die Implementierung soweit zu verallgemeinern, dass der Fuzzer auch auf andere Protokolle angewendet werden kann. Für den Entwurf des Fuzzers stehen in dieser Bachelorarbeit zwei mögliche Ansätze zur Verfügung. Diese sollen im folgenden in Bezug auf die mögliche Verallgemeinerung gegeneinander abgewogen werden. Basierend auf dieser Ausarbeitung soll ein Entwurf eines SSH-Fuzzers erstellt werden.

3.1 Built-In-Methode

Bei der Built-In Methode wird der Entwurf des Fuzzing Tools so ausgelegt, dass der Fuzzer entweder als Server oder als Client zu sehen ist. Der Aufbau in Abbildung 3.1 zeigt einen Fuzzer, der als Client realisiert ist. Anders herum ist es, wenn das zu testende System der Client ist, dann muss der Fuzzer als Serversystem entworfen werden. Dabei wird der Fuzzer genau nach jeweiligem beschriebenen Standard des Protokolls implementiert.

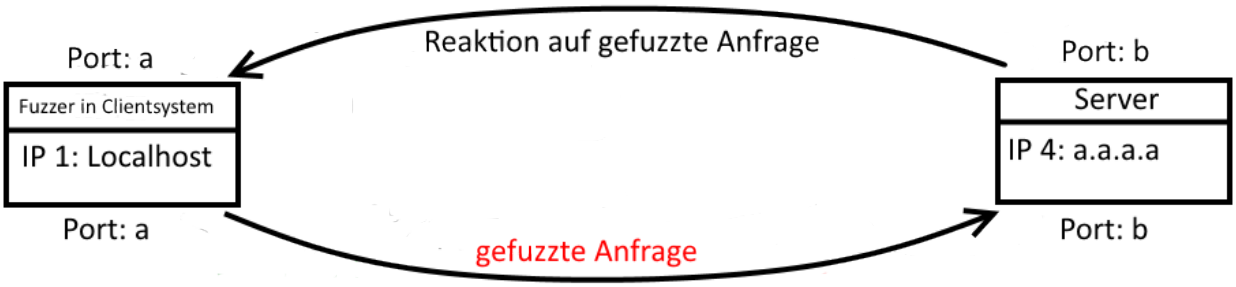

Abb. 3.1: Funktionsweise Built-In Methode

Der Aufbau, der in Abbildung 3.1 aufzeigt wird, lässt sich wie folgt erklären: Der in diesem Beispiel entworfene Fuzzer ist als Client ausgelegt. Bei der Ausführung des Fuzzers baut das Programm eine Verbindung zu dem Server (IP-Adresse x.x.x.x) über die Konfiguration des gegebenen Standards auf. Die Reaktion des Servers auf die gefuzzten Nachrichten in Form von Antworten wird ohne Umweg zu dem Client (IP-Adresse a.a.a.a) zurück gesendet.

In der späteren Implementierung des Fuzzers muss jedes Detail des Standards beachtet werden, damit ein später gewünschter Testfall nicht an dem Aufbau und Ablauf des implementierten Netzwerkprotokolls scheitert. Da der Fuzzer nur auf ein spezielles Protokoll anwendbar ist, kann zu einem späteren Zeitpunkt keine Verallgemeinerung des Fuzzing Tools stattfinden. Weiterhin kommt erschwerend hinzu, dass sobald das zu fuzzende System getauscht werden soll, der Entwurf des Fuzzers neu erstellt werden muss.

Als Vorteil dieser Methode kann der realitätsnahe Aufbau des Systems gesehen werden. Dieser sorgt dafür, dass bei einer kompletten Implementierung ein großer Teil des Codes abgedeckt werden kann.

3.2 Interceptor-Methode

Bei dieser Methode wird der Entwurf des Fuzzing Tools so ausgelegt, dass die zu fuzzende Nachricht über einen Interceptor, der sich zwischen Client und Server befindet, verändert wird. Dieser Entwurf ist in Abbildung 3.2 zu sehen. Der zu entwerfende Interceptor kann dabei in beide Richtungen angewandt werden, da er nicht speziell als Client oder Server ausgelegt ist.

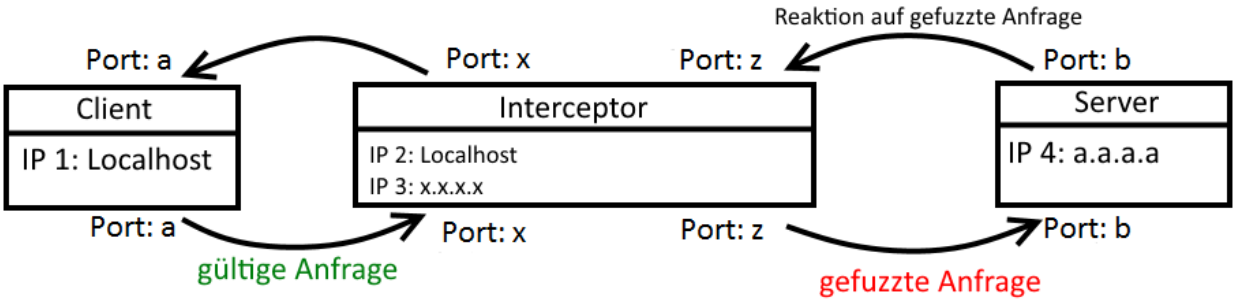

Abb. 3.2: Funktionsweise Interceptor-Methode

Der Aufbau, der in Abbilung 3.2 zu sehen ist, erklärt sich wie folgt: Hierbei ist das zu testende System der Server. Die Grundlage dieser Methode ist, dass der Interceptor und der Client sich auf einem Betriebssytem befinden. Der Client kann so konfiguriert werden, dass er bei Verbindungsaufbau nicht wie sonst versucht, einen Sendekanal über den Socket mit der IP-Adresse IP a.a.a.a und dem Port b des Servers zu öffnen, sondern einen Sendekanal über den Socket mit der IP-Adresse IP localhost und Port x des Interceptors öffnet. Der Interceptor muss bei ankommenden Anfragen auf diesem Socket einen Sendekanal über einen Socket mit der IP-Adresse IP x.x.x.x und Port z mit dem zu fuzzenden System aufbauen (IP a.a.a.a und Port b). Die jeweiligen Antworten des Servers muss der Interceptor wieder an den Client zurück senden, da nur der Client auf die zurück gesendeten Nachrichten reagieren kann.

Die spätere mögliche Implementierung des Interceptors ist dabei einfacher als die Implementierung eines Built-In Entwurfes. Bei der Interceptor-Methode muss nicht ein kom-

pletter Server oder Client nach einem definierten Standard für ein Netzwerkprotokoll, entworfen werden. Weiterhin ist es zu einem späteren Zeitpunkt noch möglich, eine Verallgemeinerung des Fuzzing-Tools durchzuführen. Da der Interceptor als Schnittstelle implementiert werden kann, an den später die zu fuzzenden Daten übergeben werden können.

Nachteil der Interceptor-Methode ist, dass das zustandsabhängige Fuzzen schwierig ist. Der Interceptor kann selbst keine Nachrichten generieren, sondern kann nur auf die versendeten Nachrichten des Clients oder des Servers zurückgreifen. Diese senden ihr Nachrichten aber nach einer festen Vorgabe, die dem normalen Ablauf der jeweiligen Implementierung folgt.

3.3 Ausgewählte Methode

Nachdem die beiden Methoden vorgestellt wurden, gilt es nun, eine Methode für die spätere Implementierung eines Prototypen zu wählen und für diese Methode einen genaueren Entwurf zu erarbeiten. Da die Interceptor-Methode in Bezug auf die Verallgemeinerung des Fuzzers zu einem späteren Zeitpunkt die besseren Ansatzmöglichkeiten bietet und nicht wie bei der Bulit-In Methode der komplette Code ausgetauscht werden muss, soll in dieser Arbeit die Methode des Interceptors weiterverfolgt und implementiert werden. Der Prototyp des SSH Fuzzers soll zum einen den Algorithmenaustausch und den Schlüsselaustausch und zum anderen den kryptographischen Teil des SSH-Protokolls fuzzen.

3.4 Entwurf eines SSH-Protokoll Fuzzers

Der zu entwerfende Prototyp des SSH-Fuzzers besteht in diesem Entwurf aus zwei getrennten Systemen. Ein System soll dabei das SSH-Client- und Fuzzersystem beherbergen und das zweite System soll den zu testenden SSH-Server beinhalten. An dieses Serversystem soll später der Prototyp seine gefuzzten Nachrichten schicken und die jeweilige Antwort des Systems empfangen und richtig interpretieren. Der grobe Aufbau der beiden Systeme ist in Abbildung 3.3 zu sehen.

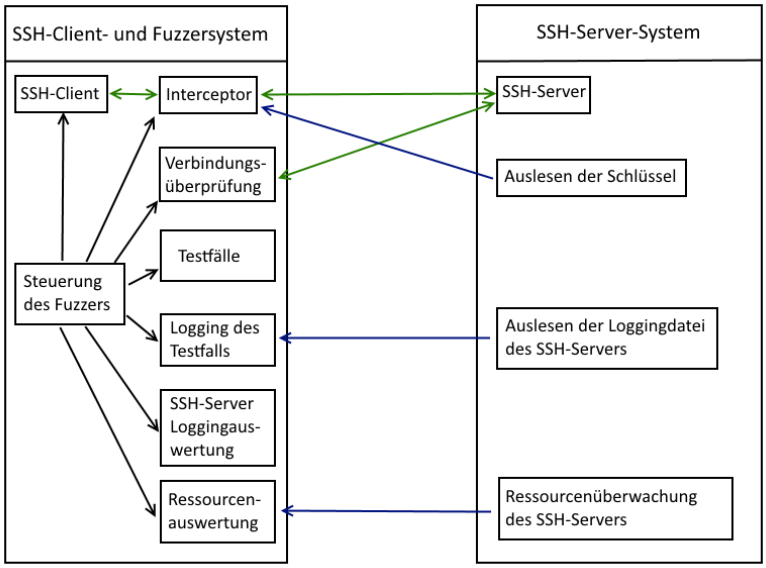

Abb. 3.3: Entwurf des SSH Prototyps

Die in Abbildung 3.3 gezeigten Funktionen und Beziehung zwischen den zu entwerfenden Systemen sollen in den jeweils folgenden Unterkapiteln erläutert werden.

3.4.1 Das SSH-Client- und Fuzzersystem

Dieser Teil des Entwurfs soll einen SSH Client beschreiben, der die zu fuzzenden Nachrichten erzeugt. Zudem wird der Entwurf des Fuzzersystems erstellt. Die zu entwerfenden Funktionblöcke des Fuzzers sind die Steuerung der einzelnen Funktionen des Fuzzers, die einzelnen Testfälle, der Interceptor, die Verbindungsüberprüfung und das Logging der Testfälle.

3.4.1.1 Die Steuerlogik

Das Herz des Fuzzers bildet die Steuerlogik. Dieser Teil des Fuzzers ist dafür verantwortlich, dass die einzelnen Funktionen des Fuzzers bei der Ausführung der Testfälle richtig aufgerufen werden. Die Steuerlogik soll dabei so entworfen werden, dass für jeden Testfall eine bestimme Abfolge der abzuarbeiteten Teile eingehalten wird.

Diese Abfolge ist wie folgt: Die Steuerung soll zu Beginn eines kompletten Tests die vorhandenen Testfälle laden. Anschließend soll die Steuerung in eine Schleife gehen, in der jeweils der Aufruf des Interceptors, der Aufruf der Verbindungsüberprüfung und die Funktion, die für das Logging des einzelnen Testfalles verantwortlich ist, aufgerufen werden. Diese Schleife soll so oft wiederholt werden wie Testfälle vorhanden sind. Falls keine weiteren Testfälle vorhanden sind, soll die Steuerung das Programm beenden.

Zudem soll dem Anwender die Möglichkeit gegeben werden, gegebenenfalls Testfälle in einem Fehlerfall zu reproduzieren. Hierfür sollen der Steuerlogik beim Aufruf zwei Parameter übergeben werden. Der erste Parameter dient dazu den Starttestfall festzulegen, ab dem die Steuerlogik anfangen soll, Testfälle auszuführen. Der zweite übergebene Parameter dient dazu, den Endtestfall festzulegen.

Der Ablauf der Steuerung ist in dem Aktivitätsdiagramm in Abbildung 3.4 zu sehen.

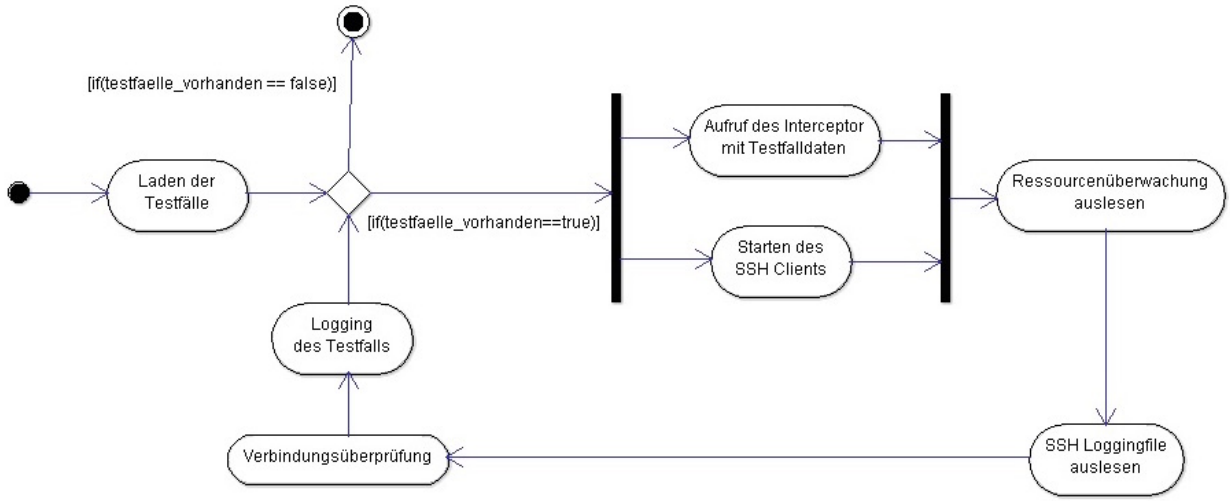

Abb. 3.4: Entwurf der Steuerung

3.4.1.2 Die einzelnen Testfälle des Prototyps

Die ersten Testfälle des Prototyps des Fuzzers sollen nicht quantitativ angelegt werden, sondern gezielt auf einzelne ausgesuchte Nachrichten und Felder abzielen. Mit den ausgewählten Testfällen soll eine größtmögliche Codeabdeckung erzielt werden. Die entworfenen Testfälle basieren auf der Analyse der Standards des SSH-Protokolls (Version 2). In diesen Standards wurde nach den einzelnen verwendeten Nachrichten in den Schichten des Transport- und Authentifikationsprotokoll gesucht. In den gefundenen Nachrichten wurden dann einzelne Felder herausgesucht, die sich zur Veränderung anbieten. Die festgelegten Testfälle des Entwurfes sind in Tabelle 3.1 zu finden.

Die Wahl der einzelnen Testfälle und deren Fuzzingdaten aus Tabelle 3.1 lässt sich wie folgt begründen:

1. Testfall: Der Fokus dieses Testfalles liegt auf der Aushandlung der verwendbaren Algorithmen zur Generierung der einzelnen Keys. Die Trennung der Algorithmen erfolgt über das Trennzeichen Komma (Hexwert: 2c). Hierbei soll ein beliebiges Stringfeld so manipuliert werden, dass mehrere Kommas hintereinander in das Stringfeld eingefügt werden

Tab. 3.1: Entwurf Testfälle SSH

Testfall	MSG_ID_SSH	Startposition Fuzzing	Länge Fuzzing Daten	Fuzzingdaten	Art des Fuzzing
1	20	108	14	2c2c2c2c2c2c2c	Einfügen
2	20	0	8	ffffffff	Ersetzen
3	20	0	8	0d0a0d0a	Ersetzen
4	30	12	8	00000000	Ersetzen
5	5	0	8	ffffffff	Ersetzen
6	50	116	8	passwd:0d0a0d0a	Ersetzen
7	50	170	6	ffffff	Ersetzen
8	50	176	6	ffffff	Ersetzen

sollen. Es soll somit getestet werden, ob die Implementierung des Servers dieses korrekt verarbeiten kann. Zudem muss beachtet werden, dass Daten in die vorhandene SSH-Nachricht mit eingebaut werden. Somit muss das Längenfeld des SSH-Paketes und die Längenangabe der Schlüsselaustauschalgorithmen mitangepasst werden. Testfall 1 ist in Abbildung 3.5 nochmals dargestellt.

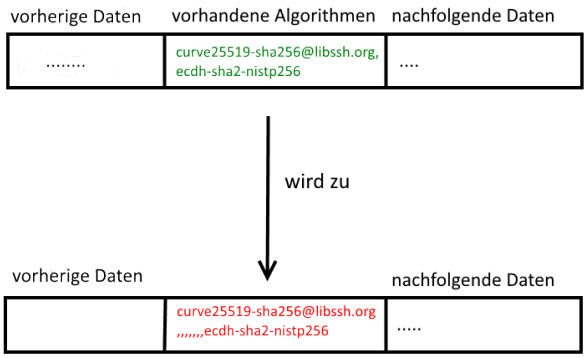

Abb. 3.5: Testfall 1

2. Testfall: Der Fokus dieses Testfalles liegt auf der Manipulation von Längenfeldern. Das Längenfeld der SSH-Nachricht soll mit der größtmöglichen Zahl gefuzzt werden. Der einzufügende Wert ist hier 0xffffffff. Dadurch soll überprüft werden, ob die Implementierung des Servers den Fehler im Längenfeld erkennt und die Verbindung ordungsgemäß abbricht. Es müssen keine Anpassungen an dem Längenfeld der SSH-Nachricht statt finden, da die Daten nicht eingefügt werden, sondern ersetzt werden. Testfall 2 ist in Abbildung 3.6 nochmals dargestellt.

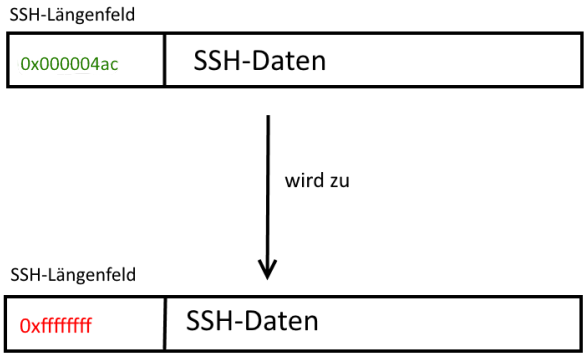

Abb. 3.6: Testfall 2

3. Testfall: Der Fokus dieses Testfalles liegt ebenfalls auf der Manipulation eines Längenfeldes. Es ändert sich aber der einzufügende Wert. Dieser Hexadezimalwert ist hier 0x0d0a0d0a. Die Wahl fiel auf diesen Wert, da das SSH-Protokoll am Anfang eines jeden Verbindungsaufbaus eine Client-Hello Nachricht versendet. Diese Nachricht endet immer mit einem LineFeed (0x0a) und Carriage Return (0x0d). Bei der Manipulation dieser Nachricht soll versucht werden, das vorläufige Ende der Nachricht zu simulieren. Testfall 3 ist in Abbildung 3.7 nochmals dargestellt.

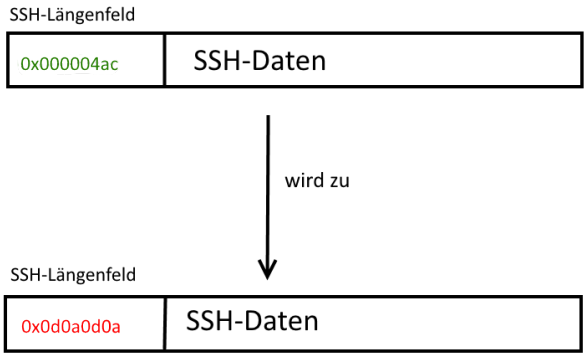

Abb. 3.7: Testfall 3

4. Testfall: Der Fokus dieses Testfalles liegt wieder in der Manipulation eines Längenfeldes. Der zu fuzzende Nachrichtentyp des SSH-Protokolls ist die ECDH KEY EXCHANGE Init Nachricht des Clients. Hierbei gibt es ein Längenfeld, dass die Länge des Multiprecision Integerwertes angibt. Dieser soll hier auf den Hexadezimalwert 0x00000000 gesetzt werden. Es müssen keine Anpassungen an dem Längenfeld der SSH-Nachricht stattfinden, da die Daten nicht eingefügt werden, sondern ersetzt werden. Testfall 4 ist in Abbildung 3.8 nochmals dargestellt.

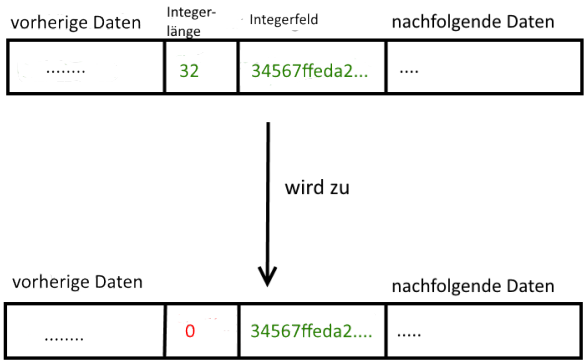

Abb. 3.8: Testfall 4

5. Testfall: Dieser Testfall des SSH-Fuzzers soll versuchen die SSH-Längenangabe eines verschlüsselten Protokolls zu verändern. Dadurch soll der verwendete Verschlüsselungsalgorithmus getestet werden. Die Daten, die eingefügt werden sollen sind: 0xffffffff. Testfall 5 ist in Abbildung 3.9 nochmals dargestellt.

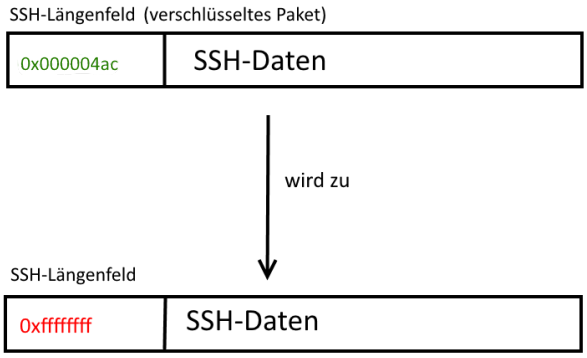

Abb. 3.9: Testfall 5

6. Testfall: Der Fokus dieses Testfalles liegt auf der Manipulation eines Strings innerhalb des SSH Authentication Requests. Es soll die Authentifikationmethode mit einem Passwort getestet werden. Zur Identifikation des Passwortfuzzings in der SSH_MSG_USERAUTH-_REQUEST wird das Schlüsselwort *passwd:* in den Fuzzdaten vorangestellt. Für das Fuzzing soll in den String, der das Passwort enthält ebenfalls der Hexadezimalwert 0x0d0a0d0a eingebaut werden. Dieser Wert wurde aus den selben Gründen wie in Testfall Nummer drei gewählt. Es muss ebenfalls keine Veränderung des SSH Paketes vorgenommen werden, da wieder Daten nur ersetzt werden und nicht eingefügt werden sollen. Testfall 6 ist in Abbildung 3.10 nochmals dargestellt.

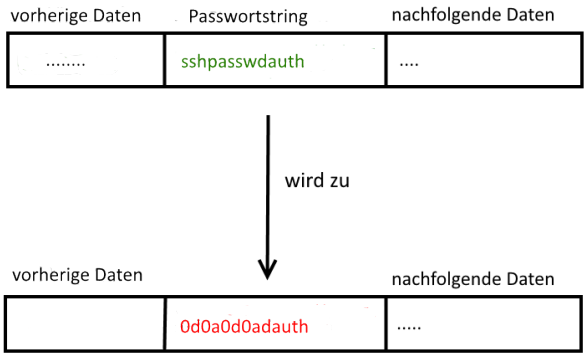

Abb. 3.10: Testfall 6

7. Testfall: Der Fokus dieses Testfalles liegt auf der Manipulation des SSH Authentication Requests. Hierbei soll die Authentifizierungsmethode Public Key gefuzzt werden. Dafür wird in der Nachricht die übertragene Länge des enthaltenen Exponent auf den Wert 0xffffffff gesetzt. Testfall 7 ist in Abbildung 3.11 nochmals verdeutlicht

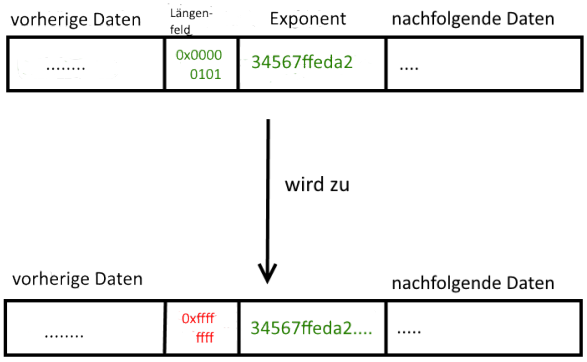

Abb. 3.11: Testfall 7

8. Testfall: Der Fokus dieses Testfalles liegt auf der Manipulation des SSH Authentication Requests. Hierbei soll die Authentifizierungsmethode Public Key gefuzzt werden. Dafür wird in der Nachricht der Wert des übetragenen Modulus verändert. Dieser wird auf den Wert 0xffffffff gesetzt. Testfall 8 ist in Abbildung 3.12 nochmals dargestellt.

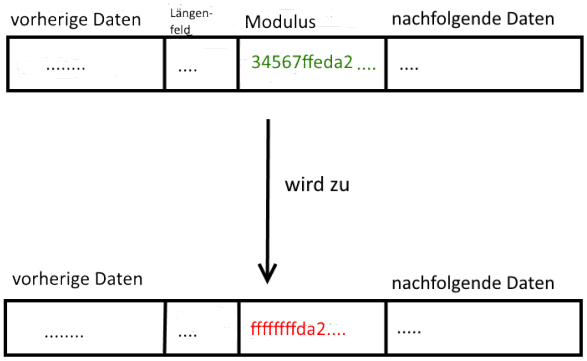

Abb. 3.12: Testfall 8

Die beschriebenen Testfälle sollen in der späteren Implementierung in einer Konfigurationsdatei enthalten sein. Diese soll dann von der Steuerung des Fuzzers ausgelesen werden. Die Konfigurationsdatei hat den Vorteil, dass ein Anwender, der weitere Testfälle hinzufügen möchte, nur Wissen über die Syntax der Konfigurationsdatei besitzen muss, nicht aber spezielle Programmierkenntnisse.

Diese definierten Testfälle sollen nun in der späteren Implementierung an einen Interceptor übergeben werden, der im folgenden Kapitel beschrieben werden soll.

3.4.1.3 Entwurf des Interceptors

Der Interceptor ist nach dem Steuerblock des Fuzzers der wichtigste Teil des Programms. Dieser Programmteil enthält die meiste Logik des Fuzzers. Der Funktionsblock des Interceptors ist dafür verantwortlich, dass Pakete, die vom SSH-Client verschickt worden sind, bei Bedarf mit gefuzzten Daten manipuliert werden können und diese anschließend an den Server weitergeleitet werden. Dazu soll das in Kapitel 3.4.1.4 vorgestellte Prinzip des Interceptors verwendet werden. Zusätzlich wird auch in diesem Block das Auslesen der Keys beschrieben.
Damit der Interceptor weiß, wann, was und an welcher Stelle er Pakete manipulieren soll, benötigt er die folgenden Übergabeparameter bei seinem Aufruf:

- SSH_MSG_ID

- Postition des zu fuzzenden Feldes

- Länge der Fuzzingdaten

- Fuzzingdaten

- Ersetzen/Einfügen

- Authentikationsmethode

Der vorletzte Übergabeparameter ist nötig, damit der Interceptor unterscheiden kann, ob er nur Daten ersetzen oder einfügen muss. Wird dem Interceptor ersetzen übergeben, muss dieser die Länge des SSH-Paketes nicht ändern. Wird der Interceptor mit dem Übergabewert einfügen gestartet, muss dieser die Längenfelder des SSH-Paketes anpassen.
Der Ablauf des Interceptorblocks ist in Abbildung 3.13 in einem Aktivitätsdiagramm dargestellt.

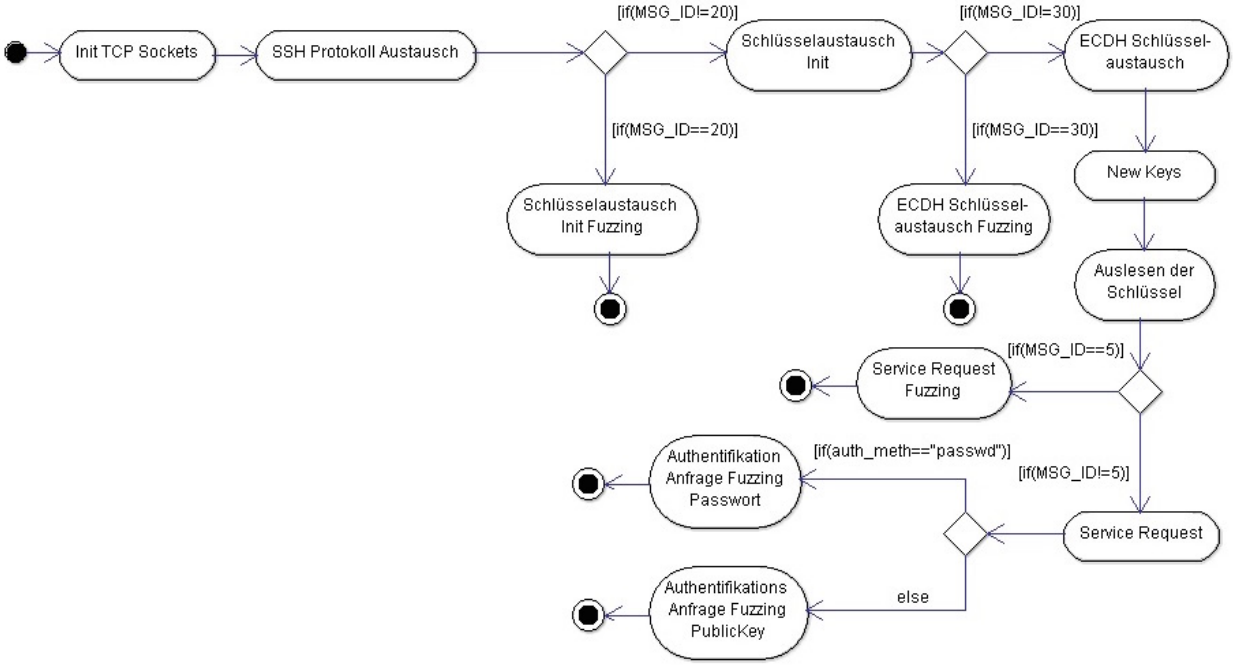

Abb. 3.13: Interceptor

Der erste Schritt des Interceptors ist, zwei gültige TCP Sockets für die Client- und die Serverseite aufzubauen. Dabei gilt es folgende Einstellungen für den Socket der Clientseite zu berücksichtigen: Dieser muss eine SSH-Verbindung über die IP-Adresse 127.0.0.1 (localhost) entgegennehmen. Der Serversocket muss wiederum über die IP-Adresse des zu testenden SSH-Server und über Port 22 aufgebaut werden.

Anschließend ist es nötig, dass der Interceptor auf dem Clientsocket auf eine SSH Verbindung wartet. Der Interceptor durchläuft darauf den Block des Protokoll Exchange. In diesem Block leitet der Interceptor die Protokoll Exchange Nachricht von Client und Server an den jeweils anderen Kommunikationspartner weiter.

Als nächstes soll geprüft werden, ob die ausgelesene MSG_ID mit der zu fuzzenden MSG_ID gleich 20 ist (entspricht der SSH_MSG_KEXINIT). Ist dies der Fall muss der

Interceptor in den Block Key Exchange Init Fuzzing wechseln und die übergebenen Fuzzingdaten an der übergebenen Position einbauen. Die Funktion des Interceptors darf nach dem Versenden der gefuzzten Nachricht an den Server nicht den Socket zum SSH-Servers schließen, da sonst ein mögliches Fehlverhalten des SSH-Servers nicht erkannt werden kann. Erst nach einer Kulanzzeit von einer Sekunde, darf der Client die Verbindung zum Server trennen. Ist die Bedingung nicht erfüllt, darf der Interceptor die Nachricht nicht verändern und muss die SSH_MSG_KEXINIT Nachricht von Client und Server an den jeweils anderen Kommunikationspartner weiterleiten.

Anschließend soll geprüft werden, ob die ausgelesene MSG_ID mit der zu fuzzenden MSG_ID gleich 30 ist (entspricht der SSH_MSG_KEXDH_INIT). Ist dies der Fall muss der Interceptor in den Block ECDH Key Exchange Fuzzing wechseln und die übergebenen Fuzzingdaten an der übergebenen Position einbauen. Die Funktion des Interceptors darf nach dem Versenden der gefuzzten Nachricht an den Server nicht den Socket zum SSH-Servers schließen, da sonst ein mögliches Fehlverhalten des SSH-Servers nicht erkannt werden kann. Erst nach einer Kulanzzeit von einer Sekunde, darf der Client die Verbindung zum Server trennen. Ist die Bedingung nicht erfüllt, darf der Interceptor die Nachricht nicht verändern und muss die SSH_MSG_KEXDH_INIT Nachricht des Client an den Server weiterleiten. Der Inteceptor muss dann wiederum die Nachricht SSH_MSG_KEXDH_REPLY an den Client weiterleiten.

Der nächste Block des Interceptors ist zum einen dafür verantwortlich, dass die SSH_MSG-_NEWKEYS an den anderen Kommunikationspartner weitergeleitet werden und zum anderen zu prüfen, ob die SSH_MSG_NEWKEYS von beiden Seiten gesendet wurde. Ist dies der Fall kann das Auslesen der Sessionkeys erfolgen, da ab jetzt die beiden Kommunikationspartner verschlüsselt kommunizieren und die folgenden zu fuzzenden Nachrichten im Klartext vorliegen müssen.

Nachdem die Schlüssel der Session ausgelesen worden sind, muss geprüft werden, ob die Nachricht SSH_MSG_SERVICE_REQUEST gefuzzt werden soll. Ist dies der Fall wird der Block Service Request Fuzzing aufgerufen und die Fuzzingdaten an der angegebenen Position eingebaut. Ergibt die Prüfung der MSG-ID, dass diese Nachricht nicht gefuzzt werden soll, müssen alle Nachrichten bis zum Fuzzing der jeweiligen Authentisierungsmethode unverändert weitergeleitet werden.

Als Letztes muss nicht mehr geprüft werden, ob die ausgelesene MSG_ID mit der zu fuzzenden MSG_ID gleich 50 ist (entspricht der SSH_MSG_USERAUTH_REQUEST), da der Ablauf des Interceptors mit dem Fuzzen der MSG_ID 50 endet. Es muss aber unterschieden werden, ob ein Fuzzing der Passwort Authentifizierung oder der Publickey Authentifizierung stattfindet. Hierzu soll in dem Übergabewert Authentifikationsmethode das Schlüsselwort *passwd:* enthalten sein. Wird dieses Schlüsselwort erkannt, soll der Fuzzer die Passwort Authentifizierung fuzzen. Wird dieses Schlüsselwort nicht übergeben, soll der Fuzzer die Publickey Authentifizierung fuzzen. Der Interceptor geht dann in den jeweiligen Block des Auth Request Fuzzing und die übergebenen Fuzzingdaten sollen an der übergebenen Position einbaut werden. Die Funktion des Interceptors darf nach dem Versenden der gefuzzten Nachricht an den Server nicht den Socket zum SSH-Servers schließen, da

sonst ein mögliches Fehlverhalten des SSH-Servers nicht erkannt werden kann. Erst nach einer Kulanzzeit von einer Sekunde, darf der Client die Verbindung zum Server trennen.

Mit dem Fuzzing dieser fünf Nachrichten des Clients ist der Interceptor komplett. Danach folgende Nachrichten sind für das kryptographische Fuzzing nicht weiter relevant.

3.4.1.4 Der SSH-Client

Das SSH-Client Programm muss auf dem gewählten Betriebssystem verfügbar sein. Der SSH Client soll dabei auf dem gleichen Betriebssystem wie der Fuzzer sein. Diese beiden Bedingungen an das Betriebssystem erfüllt Ubuntu LTS 14.04 und soll in der späteren Implementierung verwendet werden. Damit eine Verbindung über den localhost zum Fuzzer möglich ist, muss in der Konfigurationsdatei des SSH Client die Porteinstellung des Clients geändert werden. Dieser ist standardmäßig Port 22 und sollte hier oberhalb der Grenze der well-known Ports liegen (>1023) liegen. Der Port sollte über der Zahl 1023 liegen, damit kein Konflikt mit anderen Protokollen entsteht. Desweiteren muss der SSH-Client den Schlüsselaustauschalgorithmus Elliptic Curve Diffie Hellmann zur Verfügung stellen, da eine ECDH-Konfiguration getestet werden soll. Der Client muss so konfiguriert werden, dass die MAC *umac-128-ctr* und der Verschlüsselungsalgorithmus *aes-128-ctr* verwendet werden, da diese vom gegebenen Entschlüsselungs- und Verschlüsselungsprogramm verwendet wird. Zusätzlich müssen die Sessionkeys einer SSH Verbindung in einem Textfile ausgegeben werden. Hierzu kann der Source Code des OpenSSH- Client an entscheidenden Stellen geändert werden.

Die Möglichkeit die Schlüssel über einen Diffie-Hellman Man-in-the-Middle zu bestimmen, würde auch bestehen. Da der Schlüsselaustausch so realitätsnah wie möglich durch den SSH-Fuzzer getestet werden soll, ist der Schlüsselaustausch über die Serverauthentifikation abgesichert. Dadurch ist ein Man-in-the-Middle Angriff nicht möglich.

3.4.1.5 Entwurf der Verbindungsüberprüfung

Nachdem der jeweilige Testfall durch den Interceptor ausgeführt wurde, gilt es eine Verbindungsüberprüfung zum SSH-Server durchzuführen, um zu kontrollieren, ob der Server möglicherweise durch einen Testfall nicht mehr erreichbar und abgestürzt ist. Dazu sollen in der späteren Implementierung zwei Verbindungsüberprüfungen stattfinden.

Die erste Verbindungsüberprüfung ist eine Portüberprüfung des Port 22 des SSH-Servers. Hierdurch wird kontrolliert, ob der SSH Daemon auf dem Port 22 lauscht und auf TCP Handshakes wartet. Ist dies der Fall, so ist die zweite Verbindungsüberprüfung mit einem Kommando über eine gültige SSH-Verbindung durchzuführen. Diese soll im folgenden Abschnitt erläutert werden. Wenn der Fall eintritt, dass der Port auf dem Server geschlossen ist, soll die zweite Verbindungsüberprüfung nicht mehr ausgeführt werden, da diese ebenfalls negativ ausfallen würde. Das Pass oder Fail der Portüberprüfung ist in jedem Fall in das Logging des Testfalles mitaufzunehmen.

Die zweite Verbindungsüberprüfung besteht im Wesentlichen darin, dass über den SSH-Client versucht wird, eine gültige Verbindung mit dem SSH- Server aufzubauen. Hierbei soll versucht werden, ein Kommando, wie zum Beispiel “whoami”, auf dem Server auszuführen und dessen Reaktion auf den Befehl auszuwerten. Gelingt dies so, ist der SSH-Server weiterhin erreichbar. Falls die zweite Verbingungsüberprüfung misslingt, so ist der SSH-Daemon in einem Zustand, in dem er SSH-Anfragen nicht mehr richtig verarbeiten kann. Nur für den Fall, dass die zweite Verbindungsüberprüfung auch wirklich ausgeführt wird, ist diese in das Logging mitaufzunehmen. Ansonsten ist ein Vermerk dem Logging hinzuzufügen, dass der Test nicht durchgeführt wurde.

In Abbildung 3.14 ist der Ablauf der Verbindungsüberprüfung in einem Aktivitätsdiagramm dargestellt.

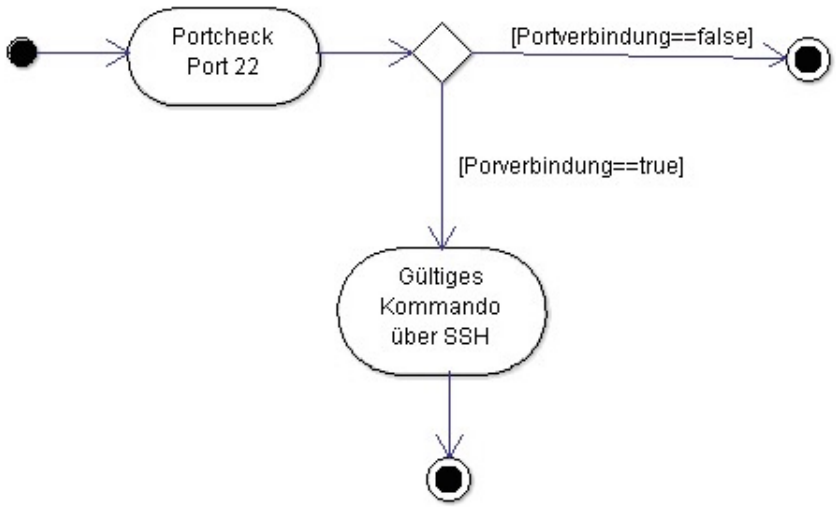

Abb. 3.14: Verbindungsüberprüfung

3.4.1.6 Entwurf der Auswertung des SSH-Loggingfiles

Die Loggingdatei des SSH-Daemons soll über den Client ausgewertet werden. Die Auswertung der Loggingdatei zu jedem Testfall soll über eine Änderungsverfolgung realisiert werden. Hierzu sollen alle hinzugekommenen Logginginformationen seit der Ausführung des Testfalles ausgelesen werden. Die Auswertung erfolgt immer erst nach der Ausführung eines Testfalles. Der Rückgabewert der Auswertung soll der Steuerung des Fuzzers übergeben werden. Dieser Rückgabewert eines Strings soll dann an den Block der Erstellung der Loggingdatei des Testfalles übergeben werden. Das Loggingfiles des SSH-Daemon auf der Serverseite soll über eine Netzwerkfreigabe auf dem Clientsystem eingebunden werden. Diese Netzwerkfreigabe soll über den NFS Dienst realisiert werden. Die Loggingdatei *authlog* ist unter dem Verzeichnis */var/log/* zu finden.

3.4.1.7 Entwurf der Auswertung der Ressourcenüberwachung

Die Auswertung der geschriebenen Datei des Servers, soll über eine Netzwerkfreigabe dem Fuzzer zur Verfügung gestellt werden. Falls die angegebene Speicherressource 10 Prozent über dem Normalwert von 2550 Kilobyte liegt, soll die komplette Zeile als Rückgabewert an die Steuerung des Fuzzing Tools übergeben werden. Der Normalwert einer Speicherauslastung von 2550 KByte ergibt sich durch die Mittelung aus mehreren erfassten Speicherauslastungen. Falls der Grenzwert nicht überschritten wird, soll als Rückgabewert *Grenzwert nicht überschritten* zurück gegeben werden.

3.4.1.8 Entwurf des Logging der Testfälle

Das Logging der Testfälle gliedert sich im Entwurf in jeweils zwei Teile. Der erste Teil befasst sich mit der Struktur der Archivierung der Testfälle. Der zweite Teil soll sich mit dem Aufbau eines einzelnen Logging Files und dem Erstellen einer CSV-Datei, in der alle Testfallergebnisse festgehalten werden sollen, auseinander setzen.

Der erste der beiden Schritte zu einem übersichtlichen Logging ist, dass die einzelnen Testfälle strukturiert archiviert werden. Dazu ist die in Abbildung 3.15 ersichtliche Ordnerstruktur entworfen worden.

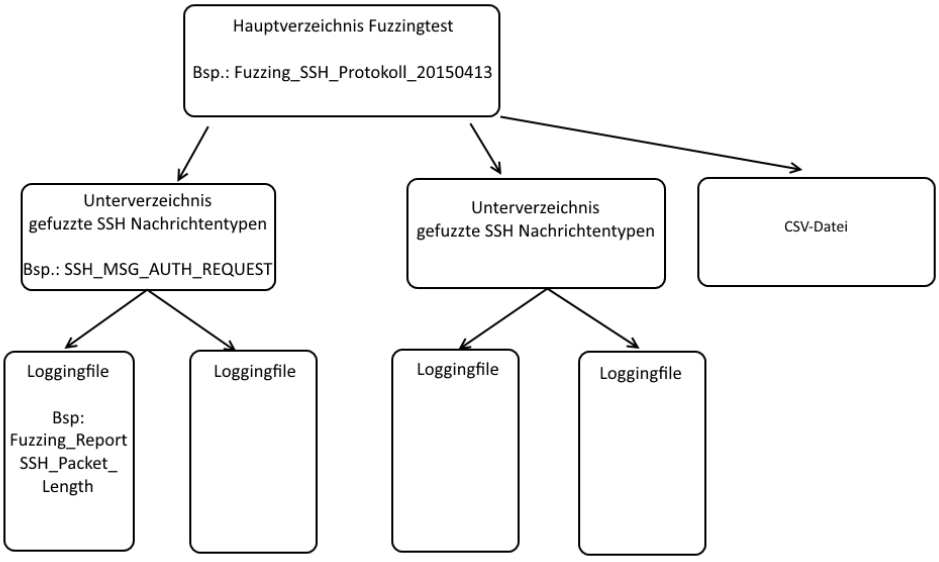

Abb. 3.15: Aufbau der Ordnerstruktur des Fuzzers

Das Hauptverzeichnis eines jeden durchgeführten Testes soll den Titel Fuzzing_SSH_Protokoll_<DATUM> tragen. An diesen Titel soll zusätzlich das Datum, an dem der Test ausgeführt wurde, im Format JJJJMMTT angehängt werden. Das Verzeichnis sollte dann zum Beispiel den Namen: “Fuzzing_SSH_Protokoll_20150407” tragen. Dieses Verzeichnis soll einmalig erstellt werden und bei wiederholtem Aufruf des

Logging Funktionsblockes muss geprüft werden, ob das Verzeichnis schon erstellt wurde. Wenn dies schon geschehen ist, darf dieses Verzeichnis nicht mehr erstellt werden.

In diesem Verzeichnis befinden sich wiederum weitere Verzeichnisse, die als Namen die gefuzzten Nachrichtentypen tragen. Ein Beispiel für einen Name eines solchen Verzeichnisses wäre “SSH_MSG_AUTH_REQUEST”. Dieses Verzeichnis soll initial erstellt werden.

Der Ablauf des Programmteiles zur Erstellung der Ordnerstruktur ist in Abbildung 3.16 in einem Aktivitätsdiagramm dargestellt.

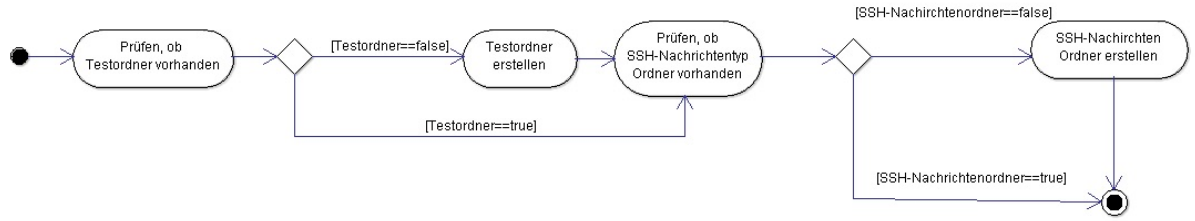

Abb. 3.16: Ablauf des Ordnerfunktionsblock

Innerhalb dieser Verzeichnisebene befinden sich nun die Loggingfiles der einzelnen Testfälle. Der Titel dieser Loggingdateien soll sich immer aus zwei Teilen zusammensetzen. Der erste immer gleichbleibende Teil enthält das folgende Präfix: Fuzzing_Report_ . An dieses Präfix soll sich das gefuzzte Feld anschließen. Am Schluss des Titels soll ein Zeitstempel des Testfalles angehängt werden. Der Aufbau des Inhaltes einer Loggingdatei ist in Listing 4.1 zu sehen.

List. 3.1: Aufbau der Loggingdatei eines Testfalles

```
1  ################################################################################
2  ########################SSH-Fuzzing-Test########################################
3  Testfall Nr.xy                          Datum:DD.MM.YYYY
4                                          Uhrzeit:HH:MM:SS
5
6  ################################################################################
7
8  Fuzzing Informationen:
9
10 Gefuzzter SSH Nachrichtentyp    : [SSH-Nachrichtentyp]
11 Position Fuzzingdaten           : [Fuzzingposition]
12 Fuzzingdaten                    : [Fuzzingdaten]
13 Gefuzzte Nachricht              : [Gefuzzte Nachricht]
14
15 ################################################################################
16 Verbindungsüberprüfung Ergebnisse:
17 Ergebnis Portcheck: [pass or fail]
18 Ergebnis SSH-Verbindungsüberprüfung: [pass, fail, not used]
19
20 ################################################################################
21 Auszug der Loggingdatei des SSH-Daemons:
22
23 [Auszug Loggingdatei zum Zeitpunkt es ausgeführten Testfalles]
24 ################################################################################
```

```
25 Auszug  des Ressourcenverbrauchs des SSH-Daemon:
26
27 SSH-Daemon Prozessorauslastung: [CPU-Auslastung]
28 SSH-Daemon Speicherauslastung:  [Ressourcenauslastung] [Grenzwert
29 über- oder unterschritten]
```

Im linken oberen Teil des Dokumentes soll die Testfallnummer, die aus den einzelnen Testfällen entnommen werden, dokumentiert werden. Rechts davon soll sich ein Datums- und Zeitstempel befinden. Anschließend an diese beiden Informationen folgen nun die spezifischen Informationen zum Fuzzing. Die erste Informationsangabe ist hier der gefuzzte Nachrichtentyp. Die darunter sich befindende Information gibt an, welches genau Feld innerhalb eines Nachrichtentyps gefuzzt wurde. In der nächsten Zeile sind die Fuzzingdaten angegeben, die in diesem Testfall verwendet wurden. Die letzte Information bezüglich des Fuzzings ist die gefuzzte Nachricht, die durch den Interceptor erzeugt wurde. Der darunter liegende Teil der Loggingdatei enthält Informationen zu der Verbindungsüberprüfung. Zum einen ist hier das Ergebnis des Portchecks zu sehen. Dies kann entweder den Wert Fail oder Pass sein. Der Begriff Pass hat die Bedeutung, dass der Portcheck erfolgreich war und der Port 22 erreichbar war. Der Ausdruck Fail hat die Bedeutung, dass der Portcheck nicht erfolgreich war und somit der Port nicht erreichbar war. Falls es zu einer gültigen SSH-Verbindungsüberprüfung gekommen ist, gibt es hier genau die gleichen Begriffe wie bei der Verbindungüberprüfung über den Portcheck. Pass bedeutet hier, dass der Befehl über einen SSH-Kanal erfolgreich ausgeführt werden konnte und Fail bedeutet, dass der Befehl nicht erfolgreich ausgeführt werden konnte. Die sich am Schluss der Loggingdatei befindenden Informationen geben Aufschluss über die Logging Info, die der SSH-Server in seinem Loggingfile abgelegt hat.

Zusätzlich soll in eine CSV (Comma-separated values) Datei die gewonnen Informationen gespeichert werden. Somit liegen alle gewonnen Informationen zur statistischen Auswertung vor. Das Erstellen einer CSV-Datei erleichtert das spätere Analysieren der ausgeführten Testfälle. Die CSV-Datei kann zum Beispiel nach Schlüsselworten durchsucht werden. Hinter diesen Schlüsselworten stehen gegebene Erwartungshaltungen. Durch diese erste automatisierte Filterung, kann eine langwierige manuelle Analyse aller ausgeführten Testfälle vermieden werden. Loggingdateien und der Schreibvorgang in die CSV-Datei ergeben das in Abbildung 3.17 zu sehende Aktivitätsdiagramm für den Loggingfunktionsblock des Fuzzers.

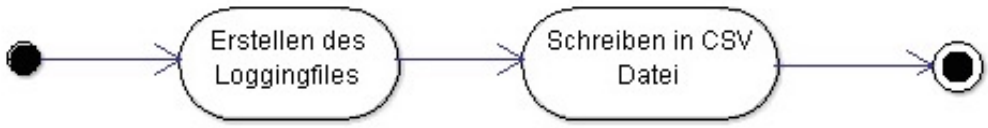

Abb. 3.17: Ablauf des Loggingfunktionsblock

3.4.2 Das SSH-Serversystem

Dieser Teil des Entwurfs soll einen SSH-Server und die einzelnen Funktionsblöcke, die für die Informationsgewinnung nötig sind, beschreiben. Die zu entwerfenden Funktionsblöcke des Fuzzers sind der SSH Daemon des Systems, die Informationsgewinnung des Exponenten zur Schlüsselgenerierung, das Auslesen des Loggingfiles des SSH-Daemons und die Ressourcenüberwachung des SSH-Daemons.

3.4.2.1 Der SSH Daemon

Unter dem SSH Daemon wird die zu testende Implementierung des SSH Protokolls verstanden. Diese ist unter OpenBSD die Implementierung OpenSSH. Die Standardkonfiguration der OpenSSH Implememententierung soll erhalten bleiben, da so reale Testbedingungen für den Fuzzer geschaffen werden.

3.4.2.2 Auslesen des Loggingfiles des SSH Daemons

Zur Auswertung der ausgeführten Testfälle soll danach die Loggingdatei des SSH Daemons ausgewertet werden. Das Logging des SSH Daemons ist standardmäßig in der Implementierung von OpenSSH aktiviert. Das Loglevel des SSH-Servers muss eingeschaltet sein und soll auf das Debuglogging eingestellt werden. Die Datei in die der SSH-Server die Logginginformationen ablegt, soll über das Netzwerk auf dem Clientsystem eingebunden werden, da die Dokumentation zu jedem Testfall auf dem Client erfolgen soll.

3.4.2.3 Auslesen der Ressourcen des SSH Daemons

Das Auslesen der Ressourceninformationen über den SSH Daemons des Serversystems soll dazu ausgenutzt werden, um abweichende höhere benötigte Ressourcen des SSH Daemons festzustellen. Dazu kann unter dem OpenBSD Betriebssystem der Befehl *top* verwendet werden. Der Befehl *top* zeigt eine Übersicht der auf dem System laufenden Prozesse und der ausgelasteten Systemressourcen an. Die Ausgabe des Befehls soll unter dem OpenBSD Betriebssystem in eine Datei umgelenkt werden. Falls in die Datei schon einmal geschrieben wurde, soll das Geschriebene gelöscht werden und die neue Ausgabe darin gespeichert werden. Die Datei soll dann wiederum über eine Netzwerkfreigabe über den Client eingebunden werden. Diese Netzwerkfreigabe ist ab Systemstart einzurichten. Das Fuzzingprogramm übernimmt dann die weitere Verarbeitung der revelvanten Informationen.

4 Implementierung

Dieses Kaptiel befasst sich mit der praktischen Umsetzung des Entwurfes aus Kapitel 3. Ziel der Implementierung ist ein funktionsfähigen Prototyp, der die SSH- Implementierung des Servers testet. Der erste Schritt der Implementierung des Fuzzers ist die beiden für die Testumgebung benötigten Systeme korrekt aufzusetzen.

4.1 Aufsetzen der Testumgebungen

Zur späteren Implementierung werden zwei Systeme aufgesetzt werden. Das erste beschriebene System dient zur Realisierung des SSH-Clients und des Fuzzers. Dieses System wird mit dem Betriebssystem Linux Ubuntu 14.04 LTS realisiert. Das zweite System ist der zu testendene SSH-Server. Hierfür wird das Betriebssystem OpenBSD gewählt. Beide Systemewerden in einer virtuellen Umgebung aufgesetzt. Dazu dient das Programm VMWare Player Version 6.0.4 von der Firma VMWare.

4.1.1 Linux Ubuntu SSH-Client-Fuzzer System

Das Linux Ubuntu SSH-Client/Fuzzer System dient zur späteren Umsetzung des SSH-Fuzzers, in dem Nachrichten verändert werden. Die im folgenden beschriebenen Konfigurations- und Installationsschritte werden benötigt, um den später zu realisierenden Fuzzer korrekt zu implementieren. Der erste Schritt zur richtigen Konfiguration des Linux Systemes sind die Einstellungen der virtuellen Umgebung. Dazu sind im verwendeten System die in Tabelle 4.1 zu sehenden Einstellungen vorgenommen worden.

Tab. 4.1: Konfiguration Virtuelle Maschine Ubuntu System

Einstellung	Wert
RAM-Speicher	2GB
Prozessoren	4
Festplattenspeicher	10GB
Netzwerkkonfiguration	NAT
IP-Adresse	192.168.222.134

Auf dieser konfigurierten Basis kann nun das Linux System installiert werden.

4.1.1.1 Installation des Grundsystems

Für die Installation von Linux Ubuntu 14.04.2 LTS 32-bit wird die ISO Datei des Systems benötigt. Diese Datei wird von Ubuntu unter der folgenden Link `http://www.ubuntu.com/download/desktop` bereit gestellt. Die Installation des Systems verläuft standardmäßig ohne Veränderungen an der Installation. Der Benutzer ssh-client-fuzzer wird auf dem installierten System eingerichtet. Unter diesem Benutzer sollen später der Fuzzer ausgeführt werden. Zusätzlich zu der Standardinstallation ist es notwendig, dass gewisse Pakete nachinstalliert werden. Diese werden über den Befehl:

sudo apt-get install <Paketname>

nachinstalliert.

Die benötigen Pakete sind in Tabelle 4.2 mit einer kleinen Beschreibung des Verwendungszwecks hinterlegt.

Tab. 4.2: Ubuntu Pakete

Paketname	Beschreibung
perl	Dient zum Ausführen von perl Skripten
nfs-common	Paket zur Einbindung von NFS Freigaben
nmap	Programm zur Untersuchung von Netzwerkports
wireshark	Programm zur Untersuchung von Netzwerktraffic
vmware-tools	Programm zur Verwendung von Shared Folders
openjdk-7-jdk	Java Delevelopment Kit
libconfig-json-perl	Programm zur Verwendung von Json mit Perl

Weiterhin werden für die Skriptssprache Perl Erweiterungen benötigt. Diese werden über die sogenannte CPAN Shell installiert. Die CPAN Shell wird mit folgendem Befehl aufgerufen:

perl -MCPAN -e shell

In der sich geöffneten CPAN Shell wird mit dem Befehl *install <Paketname>* die in Tabelle 4.3 geforderten Pakete installiert.

Tab. 4.3: CPAN Pakete

Paketname	Beschreibung
IO::Socket::INET	Erweiterung zur Nutzung von TCP- und UDP Sockets
Json	Nutzung der Funktionen zum Parsen von Json Datenstrukturen

4.1.1.2 Konfiguration des SSH-Client

Die Konfiguration des SSH-Client wird über die Konfigurationsdatei ssh.conf unter dem Verzeichnis */etc/* durchgeführt.

Für das Umleiten der SSH-Nachrichten an den Fuzzer wird in der Konfigurationsdatei der Wert, der den Port definiert, von 22 auf einen frei belegbaren Port verändert. Hierfür wurde der Port 5000 gewählt. Desweiteren muss gesichert sein, dass die Werte der Zeilen RSAAuth und PassAuth mit yes belegt sind. Erst dann ist es dem Client/Fuzzer möglich, sich über ein Passwort oder einen Public Key zu authentifizeren.

Für die Authentisierungsmethode PublicKey wird zudem ein Schlüsselpaar benötigt, mit dem sich der Client gegenüber dem Server authentifizieren kann. Dieses Schlüsselpaar beinhaltet einen privaten und einen öffentlichen Schlüssel. Beide Schlüssel werden auf dem Client erstellt. Dazu wird über die Kommandozeile der folgende Befehl ausgeführt:

ssh-keygen -t rsa -b 2048

Die Option -t gibt an von welcher Art der Schlüssel sein soll. Auf diesem System wird ein Schlüsselpaar mit der RSA-Methode erstellt. Die Option -b definiert die Bitlänge des Schlüssels. Diese ist bei diesem Schlüsselpaar 2048 bit. Dies entspricht einem Wert, der oft in der Praxis zur Anwendung kommt. Nachdem die Erstellung des Schlüsselpaars abgeschlossen ist, muss der öffentliche Schlüssel auf den Server kopiert werden. Die genaue Pfadangabe erfolgt in Kapitel 4.1.2.2.

Damit der SSH-Client bei einem Verbindungsaufbau die Sessionkeys ausgibt, muss ein veränderter SSH-Client kompiliert werden. Dazu sind die im folgenden beschriebenen Schritte notwendig.

Zuerst muss der Sourcecode der benötigten Programme heruntergeladen werden. Für die Implementierung OpenSSH 6.6.1 findet man die benötigten Dateien unter der Adresse: *http://www.openssh.com/portable.html*

Für die Implementierung OpenSSL 1.0.2a findet man die benötigten Dateien unter der Adresse: *http://openssl.org/source/*

Als erstes müssen die benötigten Header des Programms OpenSSL Programms für OpenSSH installiert werden. Dazu muss in das Verzeichnis gewechselt werden, an dem der entpackte OpenSSL-Ordner gespeichert wurde. Hier muss die Konfiguration des Kompiliervorgangs vorgenommen werden. Dazu muss der Befehl mit den den angegebenen Optionen ausgeführt werden:

./config –prefix=/opt/openssl-1.0.2a

Nachdem die Konfiguration erfolgreich abgeschlossen wurde, kann nun der Befehl *make* ausgeführt werden.

Über den Befehl *make tests* wird überprüft, ob der Befehl make erfolgreich ausgeführt wurde.

Durch die Ausführung des Befehls *sudo make install* wird das Programm nun in das unter der Konfiguration angegebene Verzeichnis installiert. Damit stehen nun die benötigten Headerdateien für die OpenSSH-Implementierung bereit.

Bevor die Konfiguration der OpenSSH Installation begonnen werden kann, muss noch ein Paket installiert werden, da aus diesem eine Headerdatei benötigt wird. Dieses Paket ist mit dem folgenden Befehl zu installieren:

sudo apt-get install zlib1g-dev

Nachdem das Paket erfolgreich installiert wurde, muss in das entpackte Verzeichnis, das den Source Code zu OpenSSH enthält, gewechselt werden. Hier muss die Konfiguration des Kompiliervorgangs vorgenommen werden. Dazu muss der Befehl mit den angegebenen Optionen ausgeführt werden:

./configure –prefix=/usr –sysconfdir=/etc/ssh –with-ssl-dir=/opt/openssl-1.0.2a

Anschließend muss die Datei *kex.c* im gleichen Verzeichnis geöffnet werden. In dieser Datei müssen nun Veränderungen vorgenommen werden, damit die Sessionkeys über die Standarderrorausgabe ausgegeben werden. Dazu muss ein zusammenhängendes ifdef (Zeile 584) und endif (Zeile 587) auskommentiert werden (Listing 4.1). In diesem Codeabschnitt wird eine Funktion namens *dump_digest* (Listing 4.2) aufgerufen. Diese kann aber erst verwendet werden, wenn ein zweites zusammenhängendes ifdef (Zeile 673) und endif (Zeile 689) auskommentiert wurde.

List. 4.1: kex.c DEBUG_KEX

```
1
2 //#ifdef DEBUG_KEX
3         fprintf(stderr, "key '%c'== ", c);
4         dump_digest("key", digest, need);
5 //#endif
```

List. 4.2: kex.c dump_digest-Funktion

```
1
2  //#if defined(DEBUG_KEX) || defined(DEBUG_KEXDH) || defined(DEBUG_KEXECDH)
3  void
4  dump_digest(char *msg, u_char *digest, int len)
5  {
6          int i;
7
8          fprintf(stderr, "%s\n", msg);
9          for (i = 0; i < len; i++) {
10                 fprintf(stderr, "%02x", digest[i]);
11                 if (i%32 == 31)
12                         fprintf(stderr, "\n");
13                 else if (i%8 == 7)
14                         fprintf(stderr, " ");
15         }
16         fprintf(stderr, "\n");
17 }
18 //#endif
```

Nachdem die Konfiguration erfolgreich abgeschlossen wurde und die Datei kex.c angepasst wurde, kann nun der Befehl *make* ausgeführt werden.

Das SSH Programm kann nun mit folgendem Befehl gestartet werden:

./ssh <Benutzername>@<IP-Adresse> 2> /tmp/key

Die Umleitung der Standarderrorausgabe erfolgt in das Verzeichnis */tmp/key*. Aus dieser Datei wird der Interceptor später die Sessionkeys auslesen.

4.1.1.3 NFS Freigabe

Die NFS Freigabe wird für das Auswerten der SSH Loggingdatei und für das Auslesen Ressourcenauswertung benötigt. Die NFS Freigabe soll während des Bootvorgangs des Linuxsystems eingebunden werden. Dazu muss die Datei *fstab* bearbeitet werden. Sie liegt unter dem Verzeichnis */etc/*.

Die Zeilen 14 und 15 der fstab Datei sind die für die Einbindung der NFS Freigaben relevant. Der Aufbau der beiden Einträge ist gleich. Die Angabe *192.168.222.133:/var/log* gibt an unter welcher Adresse sich das einzubindende Verzeichnis auf dem Server befindet. Der Abschnitt mit der Angabe */media/test* gibt der Benutzer an, in welchem Verzeichnis die NFS-Freigabe auf dem Client eingebunden werden soll. Die letzte Angabe mit dem Inhalt *nfs rw,nolock 0 0* definiert das verwendete Protokoll NFS, die Schreibrechte read und write und kein Lock.

Damit die Verbindung zwischen Client und Server zustande kommt, müssen in der Datei *host.allow* und *host.deny* in dem Verzechnis */etc/* Anpassungen vorgenommen werden.

In der Datei *host.deny* muss die folgende Zeile hinzugefügt werden:

rpcbind : ALL

Hierbei wird dem Daemon rcpbind verboten eine TCP Verbindung über jede IP-Adresse aufzubauen.

In der Datei *host.allow* muss die folgende Zeile hinzugefügt werden, damit die Netzwerkfreigabe funktioniert:

rpcbind : NFS 192.168.222.133

Hierbei wird dem Daemon rpcbind erlaubt eine TCP Verbindung über das NFS Protokoll zum Serversystem 192.168.222.133 zuzulassen.

4.1.1.4 Zeitsynchronisation mit NTP

Die Zeitsynchronisation des Clientssystem und des Serversystems ist für die spätere Auswertung der Testfälle vonnöten. Hierzu wird das Network Time Protokoll (NTP) verwendet. Hierbei soll das Linuxsystem als NTP-Server dienen. Dazu muss im ersten Schritt ein Paket, das standardmäßig installiert ist, deinstalliert werden. Hierbei handelt es sich um das Paket: *ntpdate*. Es wird mit dem Befehl

sudo apt-get remove ntpdate

deinstalliert. Anstatt des Paketes *ntpdate* muss das Paket *ntp* installiert werden. Dies geschieht mit dem Befehl:

sudo apt-get intall ntp

Falls ein DHCP-Server im Netzwerk konfiguriert ist, dürfen über diesen keine NTP-Informationen bezogen werden. Damit dies nicht geschieht, muss man in der Datei *etc/dhcp3/ dhclient.conf* die Zeile *request ntp-servers* auskommentieren.

In der Datei */etc/ntp.conf* muss nun die folgende Veränderung vorgenommen werden:

Die Zeilen mit den Angaben zu bisherigen NTP-Servern muss gelöscht werden und durch die Zeilen *server 127.127.1.0* und *fudge 127.127.1.0 stratum 4* ersetzt werden. Die Adresse 127.127.1.0 ist für das NTP Protokoll reserviert und bezieht sich auf die Localtime des Systems.

4.1.1.5 Schnittstelle zur Ver- und Entschlüsselung

Die Schnittstelle zur Ver- und Entschlüsselung der SSH-Nachrichten ist nötig, da nach dem Austausch der New Keys Nachricht von Client und Server die Kommunikation verschlüsselt stattfindet.

Für die Ver- und Entschlüsselung der Daten gibt es ein Modul von der secuvera GmbH. Dieses Modul stellt ein API zur Verfügung , welche die Eingabe und Ausgabe von Daten über eine JSON Datenstruktur regelt.

Das Bindeglied zwischen dem bereitgestellten Modul und der Funktionen in Perl ist ein Shellskript. Dieses Skript leitet die Eingabedaten der Perlfunktionen an das Modul weiter. Die Antwort des Modules wird wiederum über das Shellskript an die Perlfunktion zurückgegeben.

Das entschlüsseln von Daten, die der Interceptor abgefangen hat, wird über die Funktion *testsuitedecrypt()* realisiert. Dieser Funktion muss zusätzlich der Enc-Key und der aktuelle IV-Key übergeben werden. Es werden die entschlüsselten Daten zurück gegeben.

Das verschlüsseln von gefuzzten Nachrichten wird über die Funktion *testsuitedecrypt()* realisiert. Dieser Funktion muss zusätzlich der Enc-Key, der MAC-Key und der aktuelle IV-Key übergeben werden. Die Funktion gibt zum einen die verschlüsselten Daten und zum anderen die MAC der Daten zurück. Diese beiden Rückgabewerte werden anschließend zusammengefügt.

4.1.2 OpenBSD SSH-Server System

Das OpenBSD SSH-Server System dient zur späteren Umsetzung des SSH-Servers, den es zu testen gilt. Die im folgenden beschriebenen Konfigurations- und Installationsschritte werden benötigt, um den später zu realisierenden Fuzzer korrekt zu implementieren. Der erste Schritt zur richtigen Konfiguration des OpenBSD-Systems sind die Einstellungen der virtuellen Umgebung. Dazu sind im verwendeten System die in Tabelle 4.4 zu sehenden Einstellungen vorgenommen worden:

Tab. 4.4: Konfiguration Virtuelle Maschine OpenBSD

Einstellung	Wert
RAM-Speicher	256MB
Prozessoren	1
Festplattenspeicher	10GB
Netzwerkkonfiguration	NAT
IP-Adresse	192.168.222.133

Auf dieser konfigurierten Basis kann nun das Betriebssystem OpenBSD installiert werden.

4.1.2.1 Installation des Grundsystems

Für die Installation von OpenBSD 5.5 32-bit wird die ISO Datei des Systems benötigt. Diese Datei wird von OpenBSD unter dem folgenden Link `http://www.openbsd.org/ftp.html` bereit gestellt. Die Installation des Systems verläuft standardmäßig ohne Veränderungen an der Installation. Die in Tabelle 4.5 dargestellten Benutzer werden auf diesem Sysytem eingerichtet.

Tab. 4.5: Benutzer auf Serversystem

Benutzer	Passwort
sshpubkeyauth	sshpubkeyauth
sshpasswdauth	sshpasswdauth

4.1.2.2 Konfiguartion SSH-Server

Der Benutzer *ssh-client-fuzzer* des Clientssystems wird sich auf dem Serversystem über SSH mit dem Benutzer *sshpubkeyauth* und *sshverbindtest* anmelden. Dazu wird der PublicKey des Benutzers *ssh-client-fuzzer* in dem Ordner */home/sshpubkeyauth/.ssh* und */home/sshverbindtest/.ssh* kopiert.

4.1.2.3 NFS-Freigabe

Die NFS-Freigabe wird für das Auswerten der SSH-Loggingdatei und für die Ressourcenüberwachung benötigt. Dazu muss die Datei *exports* bearbeitet werden. Sie liegt unter dem Verzeichnis */etc/*. In der Datei *exports* sind die Freigaben der Verzeichnisse in den Zeilen 8 und 9 zu finden. Die Freigabe in Zeile 8 bezieht sich auf das Verzeichnis */tmp/*. Diese Angabe ist im ersten Teil der Zeile zu finden. Der nächste Abschnitt gibt an, für welche Rechner das Verzeichnis einzusehen ist. Die Angabe *-network = 192.168.222.0* gibt an, dass alle IP-Adressem mit Präfix 192.168.222. auf das Verzeichnis zugreifen können. Zudem wird über die nächste Angabe *-mask = 255.255.255.0* die Subnetmaske festgelegt. Die in Zeile 9 implementierte Freigabe bezieht sich auf das Verzeichnis */var/log*, da an diese Stelle der SSH-Server seine Loggingdateien schreibt. Die Angaben zum Netzwerk und der Subnetmaske in Zeile 9 sind die gleichen wie in Zeile 8.

Damit der Client auf die NFS Freigabe zugreifen kann, muss über die hosts.allow und hosts.deny die Erlaubnis erteilt werden eine TCP-Verbindung zum Server zu erlauben.

Zuerst werden in der Datei hosts.deny alle möglichen Verbindungsanfragen kategorisch abgelehnt. Durch den Zusatz *ALL : 192.168.222.129* wird dem Client erlaubt auf den Host zuzugreifen.

Damit der NFS-Dienst beim Starten des Betriebsystems verfügbar ist, müssen Anpassungen an der Datei *rc.conf.local* vorgenommen werden.

4.1.2.4 Zeitsynchronisation über NTP

Die Zeitsynchronisation des Clientssystem und des Serversystems ist für die spätere Auswertung der Testfälle vonnöten. Hierzu wird das Network Time Protokoll (NTP) verwendet. Unter OpenBSD wird in der Datei *ntpd.conf* unter dem Verzeichnis */etc/* die folgende Zeile unter dem Abschnitt *sync to a single server* hinzugefügt:

192.168.222.134

Zusätzlich muss in der Datei *rc.conf.local* in dem Verzeichnis */etc/* die folgende Zeile eingefügt werden:

$ntpd\_flags = s$"

Dadurch wird bei jedem Systemstart die Zeit über das NTP-Protokoll mit dem angegebenen Server synchronisiert.

4.2 Implementierung des SSH-Fuzzers

Im Folgenden werden alle entwickelten Blöcke des SSH-Fuzzers beschrieben. Dabei wurden die meisten Blöcke mit der Skriptsprache Perl unter der Version 5.18.2 realisiert.

4.2.1 Implementierung der Testfälle

Die Informationen, welche der Interceptor als Übergabeparameter benötigt, damit dieser weiß, an welcher Position er die Fuzzdaten einfügen oder einsetzen muss, sind in der Testfalldatenbank implementiert. Die Testfalldatenbank ist in der Datei *testfaelle.json* implementiert und befindet sich in dem Projektverzeichnis */Implementierung/src*.

Anhand eines Testfalles wird die implementierte JSON-Datenstruktur im nachstehend beschrieben. Ein Auschnitt der Datei ist in Listing 4.3 zu sehen.

List. 4.3: Ausschnitt testfaelle.json

```
1  [
2          {"testcase1": {
3                                      "testcase":"1",
4                                      "MSG_ID":"20",
5                                      "position_fuzz":"108",
6                                      "length_fuzz":"14",
7                                      "fuzz_data":"2c2c2c2c2c2c2c",
8                                      "insert_or_replace":"insert"
9                                    }
10         }
11         ,
12 ...
13 ]
```

In diesem Teil der JSON-Datenstruktur wird jeder Testfall in einem Array abgelegt. Dieses Array wird in einer JSON-Datenstruktur durch eckige Klammern definiert. Jedes Element des Arrays wird in der Json Datenstruktur durch ein Komma getrennt. In diesem Array wird für jeden definierten Testfall ein Objekt erstellt. Ein Objekt wird in JSON durch geschweifte Klammern dargestellt. Jedes dieser Testfallobjekte bekommt einen Schlüsselwert zugewiesen. Dieser setzt sich aus dem Wort “testcase” und einer fortlaufenden Nummerierung, die bei 1 beginnt, zusammen. Hinter dem Doppelpunkt folgt der Wert, welcher dem Schlüsselwert zugewiesen ist. Da zu einem Testfall mehrere Werte hinterlegt werden, werden diese wiederum in einem Objekt definiert. In diesem Objekt sind die in Tabelle 4.6 dargestellten Schlüsselwerte und die dazugehörigen Werte enthalten.

Die angegebene Position, an der die Fuzzingdaten eingebaut werden, ist in jedem Testfall statisch.

Da der beschriebene Aufbau für die folgenden Testfälle 2-8 in der JSON-Datenstruktur genau gleich ist, wird auf die weiteren Testfälle nicht weiter eingegangen.

Tab. 4.6: JSON-Testfallobjekt

Schlüsselwert	Inhalt
testcase	1
MSG_ID	20
position_fuzz	108
length_fuzz	14
fuzz_data	2c2c2c2c2c2c2c
insert_or_replace	insert

4.2.2 Implementierung des Interceptors

Die Logik des Interceptors wird in der Datei *interceptor.pl* implementiert. Die Datei befindet sich unter dem Projektverzeichnis */Implementierung/src*. Diese Datei wird entweder von der Steuerlogik aufgerufen oder direkt vom Benutzer gestartet. Dieser Fall ist implementiert worden, damit auch ein einzelner Testfall durch den Interceptor ausgeführt werden kann.

Das Schaubild in Abbildung 4.1 soll nochmal das Prinzip, nachdem der Interceptor implementiert ist, zeigen.

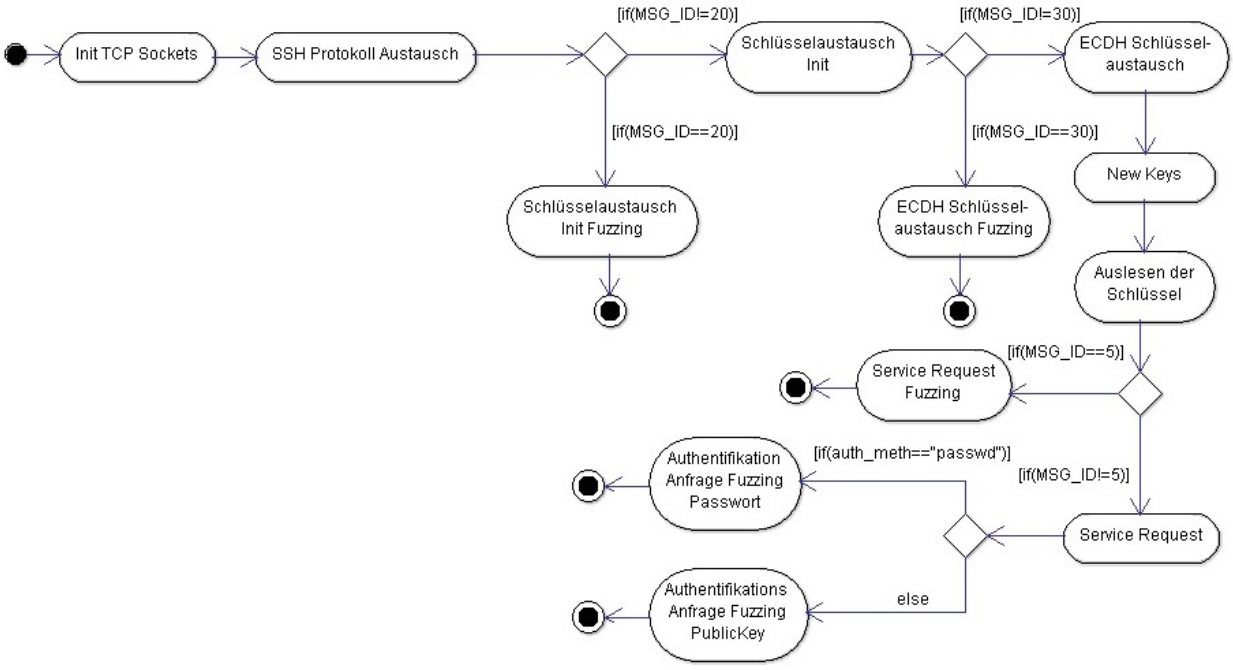

Abb. 4.1: Funktionsprinzip des Interceptors

Als erstes beginnt der Interceptor in der Funktion *Uebergabewerte_einlesen()* mit dem

Speichern der Übergabewerte in globalen Variablen. In den Übergabewerten sind die Informationen enthalten, die für das Fuzzing der bestimmten Nachricht benötigt werden.

Darauf wartet der Fuzzer solange, bis ein Socket auf dem Port 5000 über den Localhost geöffnet wird. Ist dies der Fall, dann wird zusätzlich die Verbindung in Richtung des SSH-Servers aufgebaut. Dies geschieht in der Funktion *SSH_Verbindung_annehmen()*.

Der Interceptor beginnt die SSH-Verbindung durch den Austausch der SSH-Nachrichten *SSH_ProtokollExchange* und empfängt danach die SSH-Nachricht *SSH_KeyExchange_Init* des Clients.

Die darauf folgende if-Abfrage prüft den ersten möglichen Fuzzingfall in der Key Exchange Init Nachricht ab. Entspricht die ausgelesene MSG ID der übergebenen MSG ID des Testfalles, gibt die Funktion *MSG_ID_pruefen()* eine 0 zurück und der Fall des Fuzzings wird aufgerufen. Im Falle, dass der Rückgabewert nicht 0 ist, wird der Verbindungsaufbau fortgeführt.

Im Falle, dass die Key Exchange Init Nachricht gefuzzt wird, wird die Funktion *SSH_KeyExchangeInit_Fuzz()* aufgerufen. Innerhalb dieser Funktion wird wiederum die Funktion *Fuzzdaten_Einbauen()* aufgerufen. Diese setzt die gewünschten Fuzzingdaten an die übergebene Position ein. Zudem wird in dieser Funktion überprüft, ob in diesem Testfall die Fuzzdaten ersetzt oder eingefügt werden. Im Falle des Einfügens wird auf das Längenfeld der SSH-Nachricht die Länge der eingefügten Daten addiert. In beiden Fällen wird der gefuzzte String an die Funktion *SSH_KeyExchangeInit_Fuzz()* zurückgegeben.

In diesem Fall des Key Exchange Init Fuzz wird das Längenfeld des Strings, der den zu fuzzenden String enthält, ebenfalls angepasst. Hierbei durchsucht die Funktion hinter der zu fuzzenden Position den gefuzzten String nach dem Längenfeld. Ist die Position des Längenfeldes gefunden, wird auch diese um die Länge der einzufügenden Daten angepasst.

Zum Abschluss dieser Funktion wird die gefuzzte Nachricht mit der Funktion *Nachrichtsenden_Server()* an den Server gesendet. Die darauf folgende sleep-Funktion sorgt dafür, dass der Interceptor das Ergebnis nicht durch einen zu früh geschlossenen Socket verfälscht. Der Interceptor beendet sich danach mit dem Kommando exit.

Tritt der Fall auf, dass in einer nachfolgenden Nachricht gefuzzte Daten eingebaut werden, wird die Funktion *SSH_KeyExchangeInit()* aufgerufen. Diese schickt die nicht gefuzzte Key Exchange Init Nachricht an den Server weiter. Dieser antwortet mit der Key Exchange Reply Nachricht an den Interceptor. Dieser leitet die Nachricht wiederum an den Client weiter.

Darauf folgt der Empfang der SSH Elliptic Curve Diffie-Hellman Key Exchange Init Nachricht. Diese Nachricht des Clients wird in der Funktion *SSH_ECDH_KeyExchangeInit_Empfangen()* empfangen. Diese empfangene Nachricht wird auf die gleiche Weise abgeprüft wie die empfangene *SSH_KeyExchangeInit* Nachricht. Das Einbauen oder das unveränderte Weiterleiten der Nachricht erfolgt nach dem gleichen Prinzip wie in den Funktionen *SSH_KeyExchangeInit_Fuzz()* oder *SSH_KeyExchangeInit()*.

Als Rückgabewert liefert diese Funktion die empfangene Nachricht des Servers. In diesem String ist nicht nur die SSH Elliptic Curve Diffie-Hellman Key Exchange Reply Nachricht enthalten, sondern auch gleichzeitig die SSH New Keys des Servers. Diese zeigt an, dass der Server mit der nächsten Nachricht die errechneten Schlüssel einsetzt und nur noch verschlüsselt mit dem Client kommuniziert. Anschließend wird die SSH New Keys Nachricht des Clients empfangen. Beide Nachrichten werden darauf auf ihre enthaltene MSG ID untersucht und mit der geforderten MSG ID 21 verglichen. Ist das Ergebnis dieser Prüfung falsch, wird der Interceptor beendet. Ist das Ergebnis dieser Prüfung wahr, dann wird mit der Auslesen der benötigten Schlüssel begonnen.

Das Auslesen der Schlüssel findet in der Funktion *Keys_auslesen()* statt. Hierfür wird die erzeugte Datei *key* des gepatchten Clients, die im Verzeichnis */tmp* zu finden ist, geöffnet und die sechs erzeugten Schlüssel ausgelesen. Diese ausgelesene Schlüssel werden als Array der main-Funktion zurückgegeben.

Im nächsten Schritt wird die erste verschlüsselte Nachricht in der Funktion *SSH_Service-_Request_Client_entschluesseln()* empfangen und durch die bereitgestellte Funktion *testsuiteDecrypt()* mit den Schlüsseln: *enc_key_client_to_server* und *inital_IV_key_client_-to_server* entschlüsselt.

Anschließend wird der *inital_IV_key_client_to_server* mit der Funktion *hIV_add* um die entschlüsselten Blöcke inkrementiert.

Die entschlüsselte Nachricht wird über das gleich aufgebaute . Ist dies der Fall wird die Funktion *SSH_Service_Request_Fuzz()* aufgerufen. In dieser Funktion wird die gefuzzte Nachricht erstellt. Diese wird anschließend der Funktion *testsuiteEncrypt()* übergeben. In dieser Funktion wird die gefuzzte Nachricht wieder verschlüsselt und die neu berechnete MAC wird an die verschlüsselte Nachricht angehängt.

Der zurückgegebene String der Funktion *testsuiteEncrypt()* wird mit der Funktion *Nachrichtsenden_Server()* an den Server gesendet. Darauf folgt die sleep-Funktion zum korrekten Abbau der Verbindung und der Interceptor wird danach beendet.

Falls die Prüfung der if-Abfrage falsch ergibt, wird der Austausch der verschlüsselten Nachrichten fortgesetzt. Nun folgt die Prüfung der zu fuzzenden Authentifikationsmethode in einer if-Abfrage. Enthält die Variable *auth_method* den String “passwd”, wird das Fuzzing der Passwortauthentisierung durchgeführt. Anderenfalls wird die Publickeyauthentisierung durchgeführt.

Im Falle des Fuzzings der Passwortauthentisierung werden die entsprechenden Nachrichten solange zwischen Client und Server ausgetauscht, bis die Nachricht SSH Userauth Request die Methode password enthält. Zusätzlich wird nach jedem Empfangen einer verschlüsselten Nachricht des Clients der IV-Vektor um die theoretisch zu entschlüsselnden Paketblöcke durch die Funktion *hIV_add()* inkrementiert. Die entsprechende SSH Userauth Request Nachricht wird vom Interceptor durch die Funktion *testsuiteDecrypt()* mit dem Verschlüsselungsschlüssel und dem aktuellen IV-Vektor entschlüsselt.

Die Funktion *SSH_Userauth_Request_Fuzz()* beinhaltet die gleiche Funktionalität wie die Funktion *SSH_Service_Request_Fuzz()*. Anstatt der Service Request Nachricht wird aber ide Userauth Request Nachricht gefuzzt.

Tritt der Fall auf, dass die Publickeyauthentisierung gefuzzt wird, wird der else Teil ausgeführt. Im else-Teil werden die entsprechenden Nachrichten solange zwischen Client und Server ausgetauscht, bis die Nachricht SSH-Userauth-Request die Methode Publickey enthält. Zusätzlich wird nach jedem Empfangen einer verschlüsselten Nachricht des Clients der IV-Vektor um die theoretisch zu entschlüsselnden Paketblöcke durch die Funktion *hIV_add()* inkrementiert. Die entsprechende SSH Userauth Request Nachricht wird vom Interceptor durch die Funktion *testsuiteDecrypt()* mit dem Verschlüsselungsschlüssel und dem aktuellen IV-Vektor entschlüsselt.

Die Funktion *SSH_Userauth_Request_Fuzz()* beinhaltet die gleiche Funktionalität wie die Funktion *SSH_Service_Request_Fuzz()*. Anstatt der Service Request Nachricht wird aber die Userauth Request Nachricht gefuzzt.

4.2.3 Implementierung der Verbindungsüberprüfung

Die Logik der Verbindungsüberprüfung wird in der Datei *ueberpruefung.pm* implementiert. Die Datei befindet sich unter dem Projektverzeichnis */Implementierung/src*. Die Verbindungsüberprüfung wird in einer externen Datei realisiert, da sich somit nicht der gesamte Code in einer Datei befindet. Der Start der Verbindungsüberprüfung erfolgt über den Aufruf der Funktion *verbindung_ueberpruefung()*. Der Aufruf der Funktion geschieht in der Steuerlogik des Fuzzers.

Zu Beginn der Funktion werden die beiden Variablen *socket_test* und *ssh_test* auf den Wert fail gesetzt, da zu der Initalisierung der Variablen die beiden Verbindungstests noch nicht durchgeführt wurden.

Als nächstes erfolgt die Überprüfung der Verfügbarkeit des SSH-Sockets auf Port 22 des SSH-Serversystems. Falls es dem Perlskript gelingt einen Socket auf Port 22 mit der IP-Adresse 192.168.222.133 (SSH-Server IP Adresse) zu öffnen, wird die Variable *socket_test* auf den Wert pass gesetzt. Wenn des dem Perlskript nicht gelingt einen Socket zu öffnen, dann wird sowohl die Variable *socket_test* als auch die Variable *ssh_test* auf fail gesetzt, da auch davon ausgegangen werden kann, dass der gültige SSH-Verbindungsaufbau misslingt, da der Port 22 bei der Socketüberprüfung nicht geöffnet ist.

Bei gelungener Socketüberprüfung wird zusätzlich versucht, eine gültige SSH-Verbindung aufzubauen. Dazu wird versucht, eine Verbindung vom SSH-Client mit dem Benutzer *sshtest* direkt zum SSH-Server herzustellten. Dabei wird über ein virtuelles Terminal mit diesem Benutzer der Befehl *whoami* ausgeführt. Liefert der Befehl den String *sshtest* an die SSH-Verbindungsüberprüfung zurück, war die Überprüfung erfolgreich und der Wert der Variable *ssh_test* auf pass gesetzt. Enthält der zurückgegebene String nicht den Wert

sshtest, wurde der Befehl nicht korrekt ausgeführt und der Wert der Variable *ssh_test* muss auf fail gesetzt werden.

Zuletzt werden die beiden Variablen *socket_test* und *ssh_test* in ein Array geschrieben. Dieses Array ist auch gleichzeitig der Rückgabewert der Funktion *verbindung_ueberpruefung()*.

4.2.4 Implementierung der Ressourcenüberwachung

Die Ressourcenüberwachung, die nach einem Testfall ausgelesen und ausgewertet wird, teilt sich in zwei Blöcke. Der erste Teil umfasst das Auslesen der CPU- und Speicherauslastung auf dem Serversystem des SSH-Daemons. Der zweite Teil dient dazu, die ausgelesenen Werte auf der Clientseite auszuwerten.

4.2.4.1 Implementierung zum Auslesen der Ressourcen

Die Logik zur Auslesung der Ressourcen wird in der Datei *ressource_read.sh* implementiert. Die Datei befindet sich unter dem Projektverzeichnis */Implementierung/src*.

Sobald das Skript gestartet wird, wird dauerhaft das Auslesen der Ressourcen durchgeführt. Das Auslesen der Ressourcen findet nur dann statt, wenn ein SSH-Daemon auch auf dem Serversystem vorhanden ist. Dabei kann in einer if-Bedingung abgefragt werden, ob sich unter den laufenden Prozessen ein SSH-Daemon befindet. Diese Abfrage geschieht mit der Auswertung der Ausgabe des Befehls *ps aux* mit dem Befehl *grep*. Der Befehl *grep* schließt zuerst seinen eigenen Befehl als Ergebnis aus und durchsucht dieses dann nach dem Schlüsselwort “sshd:”.

Falls ein SSH-Daemon in der Prozessübersicht gefunden wird, wird dessen Prozess-ID ausgelesen. Dazu wird erneut der Befehl *ps aux* mit den dazugehörigen Befehlen von *grep* aufgerufen. Die Prozzes-ID ist an zweiter Stelle des ausgegebenen Strings zu finden. Um die Prozess-ID herauszufiltern, wird der Befehl *awk 'print $2'* verwendet.

Mit der gefilterten Prozess-ID wird nun der Aufruf des Befehls *top -p $pid* durchgeführt. Diese Ausgabe muss nun auf die Werte der CPU- und der Speicherauslastung gefiltert werden. Dies geschieht mit der Befehlskette *tail -n 3 | awk 'print $6,$10'* . Das Ergebnis dieses Filters wird in der Variable *ress* abgelegt.

Da es vorkommen wird, dass die Variable *ress* keinen Inhalt hat, da kein SSH-Daemon Prozess gefunden wurde, wird die Variable *ress* nur dann in einer Datei abgespeichert, wenn diese auch einen Wert enthält. Das Speichern erfolgt in der Datei */tmp/ressource_sshd*, die von der Ressourcenauswertung auf der Clientseite ausgelesen wird.

4.2.4.2 Implementierung der Ressourcenauswertung

Die Logik zur Auswertung der Ressourcen wird in der Datei *ressourcenueberwachung.pm* implementiert. Die Datei befindet sich unter dem Projektverzeichnis */Implementierung/src*. Die Ressourcenauswertung wurde in einer externen Datei realisiert, da sich nicht der gesamte Code in einer Datei befindet. Der Start der Ressourcenauswertung erfolgt über den Aufruf der Funktion *ressourcenueberwachung()*. Der Aufruf der Funktion geschieht in der Steuerlogik des Fuzzers.

Zuerst wird die Funktion *file_read()* aufgerufen. In dieser Funktion werden die gespeicherten Ressourceninformationen der CPU und des Speichers von dem SSH-Daemon über eine Netzwerkfreigabe auf dem Client ausgelesen. Die CPU-Auslastung wird in der Variable *ress_cpu* und die Speicherauslastung in der Variable *ress_speicher* abgelegt.

Als nächstes erfolgt der Aufruf der Funktion *ress_speicher()*. In dieser Funktion wird die ausgelesene Speicherauslastung mit dem definierten Grenzwert verglichen. Falls die ausgelesene Speicherauslastung größer als der festgelegte Grenzwert ist, wird der Grenzwert überschritten und es wird 0 zurückgegeben. Wenn die ausgelesene Speicherauslastung kleiner als der Grenzwert ist, dann wird der Grenzwert nicht überschritten und es wird 1 zurückgegeben.

Zuletzt werden sowohl die beiden Variablen *ress_cpu* und *ress_speicher* als auch die Information, ob der Grenzwert überschritten wurde, in einem Array zurückgegeben.

4.2.5 Implementierung der Loggingauswertung

Die Logik zur Auswertung der SSH-Loggingdatei zu jedem einzelnen Testfall und der Erstellung der CSV-Datei wurde in der Datei *logging_auswertung.pm* implementiert. Die Datei befindet sich unter dem Projektverzeichnis */Implementierung/src*. Die Auswertung der SSH-Loggingdatei wurde in einer externen Datei realisiert, da sich nicht der gesamte Code in einer Datei befindet. Der Start der Auswertung der SSH-Loggingdatei erfolgt über den Aufruf der Funktion *authlog_auslesen()*. Der Aufruf der Funktion geschieht in der Steuerlogik des Fuzzers.

Im ersten Schritt zur Auswertung der Datei *authlog*, die in dem Verzeichnis */media/test* liegt, wird die zuletzt hinzugefügte Zeile ausgelesen. Dies geschieht mit dem Befehl *tail -n 1*. Diese Zeile wird in einer Variable mit dem Namen *pid_sshd* gespeichert. Diese Variable wird der Funktion *pid_auslesen()* bei deren Aufruf übergeben.

In der Funktion *pid_auslesen()* wird die Prozess-ID der übergebenen Zeile herausgefiltert und an die Hauptfunktion zurück gegeben.

Der Befehl *grep* wird mit der gefilterten Prozess-ID als Suchstring aufgerufen und liefert als Ergebnis alle Zeilen, die der zugehörige SSH-Prozess in die Datei geloggt hat. Diese Zeilen werden in einem String gespeichert und zurückgegeben.

4.2.6 Implementierung der Ordnerstruktur

Die Logik zur Erstellung der Ordnerstruktur wurde in der Datei *ordnerstruktur.pm* implementiert. Die Datei befindet sich unter dem Projektverzeichnis */Implementierung/src*. Die Erstellung der Ordnerstruktur wurde in einer externen Datei realisiert, da sich nicht der gesamte Code in einer Datei befindet. Die Erstellung der Ordnerstruktur erfolgt über den Aufruf der Funktion *ordnerstruktur_erstellen()*. Der Aufruf der Funktion geschieht in der Steuerlogik des Fuzzers.

Als erstes erfolgt das Speichern des Übergabewertes, der die *SSH_MSG_ID* enthält, in einer lokalen Variable. Die *SSH_MSG_ID* wird später benötigt, um das Unterverzeichnis zu erstellen.

Darauf erfolgt der Aufruf der Funktion *hauptverzeichnis_erstellen()*. In dieser Funktion wird das zugehörige Hauptverzeichnis des ganzen Tests erstellt. Das Hauptverzeichnis wird unter dem Verzeichnis */home/test/Fuzzing* angelegt. Für die Namensgebung des Hauptverzeichnisses wird das Datum benötigt. Dieses wird über die Perlfunktion *localtime()* abgerufen. Die Datumsinformationen werden an den fixen String *Fuzzing_SSH_Protokoll_* in der Reihenfolge Jahr, Monat, Datum angehängt. Anschließend wird überprüft, ob der definierte String, der den Namen des Hauptordners enthält, in Form eines Ordners schon vorhanden ist. Ist dies nicht der Fall, wird der Befehl *mkdir* mit dem zusammengestellten String aufgerufen. In beiden Fällen wird der erstellte String an die Hauptfunktion *ordnerstruktur_erstellen()* zurück gegeben, da bei der Erstellung des Unterverzeichnisses in das Hauptverzeichnis gewechselt werden muss.

Nachdem die Funktion *hauptverzeichnis_erstellen()* beendet wurde, wird die Funktion *unterverzeichnis_erstellen()* aufgerufen. In dieser Funktion wird zu der übergebenen *SSH MSG ID* ein jeweiliger Unterordner erstellt. Durch eine if-elseif Bedingung wird die übergebene *SSH_MSG_ID* mit den möglichen vorkommenden Nachrichtentypen 5, 20, 30 und 50 abgeglichen. Die jeweils zu vergebenden Ordnernamen sind in Tabelle 4.7 dargestellt.

Tab. 4.7: Unterverzeichnis basiernd auf

SSH_MSG_ID	Ordnername
5	SSH_MSG_SERVICE_REQUEST
20	SSH_MSG_KEXINIT
30	SSH_MSG_DIFFIE_HELLMANN_KEX_INIT
50	SSH_MSG_USERAUTH_REQUEST

Falls der String, der den Verzeichnisnamen enthält, in Form eines Verzeichnisses schon vorhanden ist, wird der Ordner nicht erstellt. Wenn er nicht vorhanden ist, wird der Ordner erstellt. In beiden Fällen wird der String des Hauptverzeichnisses und des Unterverzeichnisses zurück gegeben.

4.2.7 Implementierung des Testfallloggings

Die Logik zur Erstellung der Loggingdatei zu jedem einzelnen Testfall und der Erstellung der CSV-Datei wird in der Datei *logging.pm* implementiert. Die Datei befindet sich unter dem Projektverzeichnis */Implementierung/src*. Das Testfalllogging wurde in einer externen Datei realisiert, da sich nicht der gesamte Code in einer Datei befindet. Der Start des Testfallloggings erfolgt über den Aufruf der Funktion *logging_testfall()*. Der Aufruf der Funktion geschieht in der Steuerlogik des Fuzzers.

Zuerst werden die übergebenen Variablen in lokalen Variablen gespeichert. Bei den übergebenen Variablen handelt es sich um die gesammelten Informationen während der Ausführung eines Testfalles. Die übergebenen Variablen sind in Tabelle 4.8 dargestellt.

Tab. 4.8: Logginginformationen

Variablenname	Verwendung
testfall_hash_ref	Referenz auf Testfallinformationen
res_results_ref	Referenz auf Ressourceninformationen
logg_results	Authloginformationen
verbind_results_ref	Referenz auf Verbindungsüberprüfung
gefuzzte_nachricht	Versendete gefuzzte Nachricht

Im darauf folgenden Schritt wird die Funktion *erstellen_loggingdatei* aufgerufen. In dieser Funktion werden nun alle Informationen, die als Referenz übergeben wurden, dereferenziert. Desweiteren wird die Information, ob der Grenzwert eingehalten wurde oder nicht, für die Erstellung des Loggingfiles in ein lesbaren String gewandelt. Zusätzlich wird mit dem Aufruf der Perlfunktion *localtime()* die aktuelle Zeit und das aktuelle Datum in einem Array abgespeichert. Aus diesem Array wird Zeit in Form von Stunden-, Minuten- und Sekundenangaben und das Datum in Form von des Jahres, des Monats und des Tages in einzelnen Variablen gespeichert. Das Logging wird mit diesen Datums- und Zeitstempel versehen.

Der Name der jeweiligen Testfalldatei wird in der Variable *dateiname* festgelegt. Diese beinhaltet zum einen den String mit dem Inhalt “Fuzzing_Report_SSH_Test_” und zum anderen wird an diesen String die gerade ausgeführte Testfallnummer angehängt. Der Befehl *open* öffnet einen Filehandler. Der Filehandler wird über die Variable *FH* angesprochen. Zusätzlich wird bei der Ausführung des Befehls *open* die Variable *dateiname* angegeben. Dadurch wird der Filehandler an diese Datei gebunden. Das Schreiben des Loggings erfolgt durch den Befehl *print*. Unter Anhabe des Filehandlers nach jedem Printbefehl, wird die programmierte Ausgabe an den Filehandler umgeleitet. Dieser schreibt

die erhaltenen Informationen in die Datei. Das Schreiben der Datei erfolgt nach dem vorgestellten Muster aus Kapitel 3.4.1.8.

Im nächsten Schritt wird die Funktion *erstellen_csvdatei* aufgerufen. Diese Funktion erstellt für jeden Testfall eine Zeile in einer CSV-Datei mit den übergebenen Informationen. Auch in dieser Funktion werden nun alle Informationen, die als Referenz übergeben wurden, dereferenziert. Desweiteren wird die Information, ob der Grenzwert eingehalten wurde oder nicht, für die Erstellung des Loggingfiles in ein lesbaren String gewandelt. Zusätzlich wird mit dem Aufruf der Perlfunktion *localtime()* die aktuelle Zeit und das aktuelle Datum abgespeichert.Die Zeit wird in Form von Stunden-, Minuten- und Sekundenangaben und das Datum in Form von des Jahres, des Monats und des Tages in einzelnen Variablen gespeichert.

Die gewonnenen Informationen aus der Ressourcenauswertung, Verbindungsüberprüfung und Auswertung der SSH-Loggingdatei werden nun in der Variable *csv_string* abgelegt. Das Format der CSV-Datei setzt voraus, dass zwischen jedem Eintrag ein Komma stehen muss. So wird in diesem Fall jede gewonnene Information durch ein Komma getrennt. Am Ende der Variable *csv_string* wird ein New Line Befehl hinzugefügt. Der New Line Operator dient in dieser CSV-Datei dazu, dass jeder ausgeführte Testfall eine eigene Zeile erhält.

4.2.8 Implementierung der Steuerlogik

Die Logik der Steuerung des Testfalles wird in der Datei *fuzzer.pl* implementiert. Die Datei befindet sich unter dem Projektverzeichnis */Implementierung/src*. Mit dem Aufruf dieser ausführbaren Datei wird der Fuzzer durch den Benutzer gestartet.

Nachdem der Fuzzer gestartet wird, beginnt dieser mit dem Auslesen der JSON-Datei, in welcher sich die Testfälle befinden. Als nächstes liest die Steuerlogik des Fuzzers die über den Aufruf des Skripts übergebenen Werte ein. Bei dem ersten übergebenen Wert handelt es sich um den Startwert. Dieser gibt an bei welchem Testfall begonnen werden soll. Als zweites wird der Stoppwert eingelesen. Dieser kann dazu genutzt werden, dass der Fuzzer an einem definierten Testfall aufhört und sich beendet.

Im nächsten Schritt wird überprüft, ob ein Start- oder ein Stoppwert angegeben wurde. Ist dies nicht der Fall wird der Startwert automatisch auf 0 und der Stoppwert auf die Anzahl an verfügbaren Testfällen gesetzt.

Anschließend beginnt die Steuerlogik in einer for-Schleife, die zu bearbeitenden Testfälle auszuführen. Zuerst werden die einzelnen Informationen des Testfalles in lokalen Variablen gespeichert. Nachdem dies geschehen ist, wird der Fuzzer mit den benötigten Informationen aufgerufen. Anschließend wartet der Fuzzer eine Sekunde, da sonst der Aufruf der jeweiligen SSH-Verbindung ins Leere laufen würde. Im Falle, dass die Passwortauthentifizierung gefuzzt werden soll, muss die Steuerlogik die Verbindung des SSH-Client mit dem Programm *sshpass* und dem Benutzer *sshpasswdauth* aufbauen, da über dieses Programm

das Passwort gesetzt werden kann. In allen anderen Testfällen wählt die Steuerlogik das Programm *ssh* mit dem Benutzer *sshpubkeyauth*, um eine Verbindung zum Interceptor aufzubauen. Beide Programme verbinden sich dabei auf Port 5000 des Localhost. Zusätzlich wird die Standarderrorausgabe in eine Datei mit dem Namen *key* umgeleitet. Diese enthält die benötigten Schlüssel, falls im verschlüsselten Modus gefuzzt werden soll.

Die Steuerlogik empfängt als nächstes die gefuzzte Nachricht über den Dateizeiger. Nachdem diese Nachricht empfangen wurde, wird der Dateizeiger für den Interceptor geschlossen.

Anschließend beginnt die Steuerlogik mit dem Auslesen und Auswerten der Ressourcenübwachung für den zuvor ausgeführten Testfall mit dem Aufruf der Funktion *ressourcenueberwachung()*. Danach wird die Funktion *authlog_auslesen()* aufgerufen. Dadurch werden die geloggten Zeilen aus der Datei *authlog* ausgelesen und in einer lokalen Variable gespeichert. Die letzte Informationsgewinnung startet durch den Aufruf der Funktion *verbindung_pruefen()*. Hierbei wird der Portcheck und der SSH-Check durchgeführt. Die Ergebnisse dieses Testes werden in einem Array abgelegt.

Anschleßend beginnt die Steuerlogik mit dem Erstellen der Ordnerstruktur durch den Aufruf der Funktion *ordnerstruktur_erstellen()*. Dieser Funktion wird die MSG ID des zuvor ausgeführten Testfalles übergeben, damit diese das richtige Unterverzeichnis erstellen kann. Die Funktion liefert als Rückgabewert das aktuelle Unterverzeichnis, in die das Textfile geschrieben werden soll.

Zuletzt wird die Funktion *logging_testfall()* aufgerufen. Dabei werden dieser Funktion die gewonnenen Informationen übergeben. Zusätzlich zu diesen Informationen wird das Unterverzeichnis mit übergeben.

Falls der festgelegte Stoppwert der Steuerlogik noch nicht erreicht wird, wird die Steuerlogik das gleiche Prozedere für den nächsten Testfall durchführen. Falls der Stoppwert dann erreicht wird, beendet sich die Steuerlogik und somit ist das Fuzzing der SSH-Implementierung beendet.

5 Test des Fuzzers

Die Implementierung des SSH-Fuzzers wird nun in diesem Kapitel auf seine Funktionalität getestet. Dafür werden zuerst die einzelnen Blöcke Interceptor, Testfalldatenbank, Verbindungsüberprüfung, Ressourcenüberwachung, Loggingauswertung und Logging auf ihre Funktionalität geprüft und anschließend wird die Steuerlogik des SSH-Fuzzers, die alle einzelnen Blöcke aufruft, getestet.

5.1 Test des Interceptors

Für den kompletten Test des Interceptors müssen 10 verschiedene Tests durchgeführt werden. Durch die ersten 5 Testfälle wird der komplette Code des Interceptors durchlaufen und so auf seine Funktionalität geprüft. Das Ergebnis des jeweiligen Tests wird in Form der gefuzzten Nachricht im Klartext überprüft. Hierbei muss überprüft werden, ob der Fuzzer die gewollten Daten in der richtigen Nachricht und an der richtigen Position ersetzt oder eingefügt hat. In den nächsten 5 Testfällen wird mit der Angabe von falschen Fuzzinginformationen geprüft, ob der Interceptor diese Abfangen kann und das Fuzzing daraufhin abbricht.

Es gilt dabei zu beobachten, welche Nachrichten der Interceptor an den SSH-Server schickt. Um diese Nachrichten zu beobachten, wird während des Tests ein Mitschnitt der Nachrichten mit dem Programm Wireshark durchgeführt. Der Mitschnitt erfolgt auf dem Hostrechner der virtuellen Maschine auf dem Interface "VM-Netz8”. Der Aufbau dieser Messung ist in Abbildung 5.1 dargestellt.

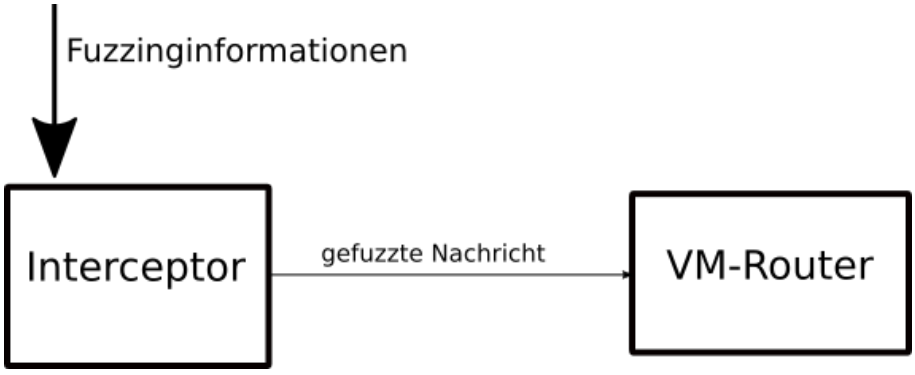

Abb. 5.1: Aufbau der Messung

Die durchzuführenden Tests sind im folgenden aufgelistet. Diese werden anschließend in den dazugehörigen Unterkapiteln beschrieben.

- Test des SSH Key Exchange Init Fuzzings
- Test des SSH Elliptic Curve Diffie Hellman Key Exchange Init Fuzzings
- Test des SSH Service Request Fuzzings
- Test des SSH Userauth Request Passwort Fuzzings
- Test des SSH Userauth Request Publickey Fuzzings
- Tests mit falschen Fuzzinginformationen

5.1.1 Test des SSH Key Exchange Init Fuzzings

Dieser Test wird die in Abbildung 5.2 gezeigten Codeblöcke des Interceptors abdecken und auf seine Richtigkeit überprüfen.

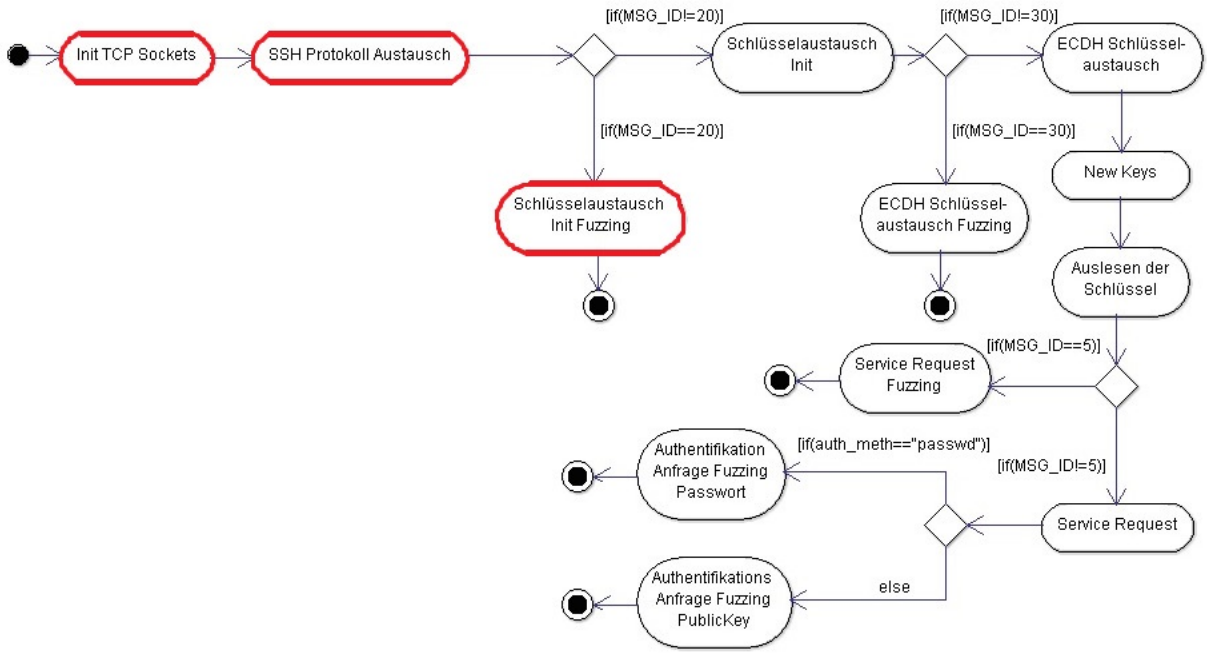

Abb. 5.2: Test 1: Key Exchange Init Fuzzing

Im ersten Test wird der Fuzzer mit den in Tabelle 5.1 gezeigten Daten aufgerufen.

Tab. 5.1: Testfall 1 Daten

Feld	Wert
SSH MSG ID	20
Pos. Fuzzing	300
Länge Fuzzing	8
Fuzzingdaten	0b0b0b0b
Art des Fuzzings	insert

Das erwartete Ergebnis ist, dass der Interceptor in der Key Exchange Init Nachricht die Fuzzingdaten 0b0b0b0b beginnend an der Position 300 bis zur Position 307 einfügt.

Das Ergebnis dieses Testfalles ist in Abbildung 5.3 zu sehen. Der Interceptor hat in der richtigen Nachricht die Daten eingefügt (rot eingerahmt). Die hexadezimal codierte MSG-ID 14 entspricht im Dezimalsystem einer 20. Zusätzlich hat der Interceptor die Daten an der richtigen Stelle eingebaut (grün eingerahmt). Da der Interceptor die Daten in der richtigen Nachricht und an der richtigen Stelle eingebaut hat, gilt der Test als bestanden.

```
0040          00 00 04 f0 07 14  af ca bf 73 ed 4b f2 4a
0050  5c c0 37 4e e2 f8 37 6f  00 00 00 bb 65 63 64 68
0060  2d 73 68 61 32 2d 6e 69  73 74 70 32 35 36 2c 65
0070  63 64 68 2d 73 68 61 32  2d 6e 69 73 74 70 33 38
0080  34 2c 65 63 64 68 2d 73  68 61 32 2d 6e 69 73 74
0090  70 35 32 31 2c 64 69 66  66 69 65 2d 68 65 6c 6c
00a0  6d 61 6e 2d 67 72 6f 75  70 2d 65 78 63 68 61 6e
00b0  67 65 2d 73 68 61 32 35  36 2c 64 69 66 66 69 65
00c0  2d 68 65 6c 6c 6d 61 6e  2d 67 72 6f 75 70 2d 65
00d0  78 63 68 61 6e 67 65 2d  0b 0b 0b 0b 73 68 61 31
```

Abb. 5.3: Test 1: Ausschnitt des Wiresharkmitschnittes

5.1.2 Test des SSH ECDH Key Exchange Init Fuzzings

Dieser Test wird die in Abbildung 5.4 gezeigten Codeblöcke des Interceptors abdecken und auf seine Richtigkeit überprüfen.

Im zweiten Testfall wird der Fuzzer mit den in Tabelle 5.2 gezeigten Daten aufgerufen.

Das erwartete Ergebnis ist, dass der Interceptor in der ECDH Key Exchange Init Nachricht die Fuzzingdaten aaaaaaaaaaaaa beginnend an der Position 150 bis zur Position 159 ersetzt.

Das Ergebnis dieses Testfalles ist in Abbildung 5.5 zu sehen. Hierbei hat der Interceptor den Fuzzingwert in der richtigen Nachricht (rot eingerahmt) und an der richtigen Stelle

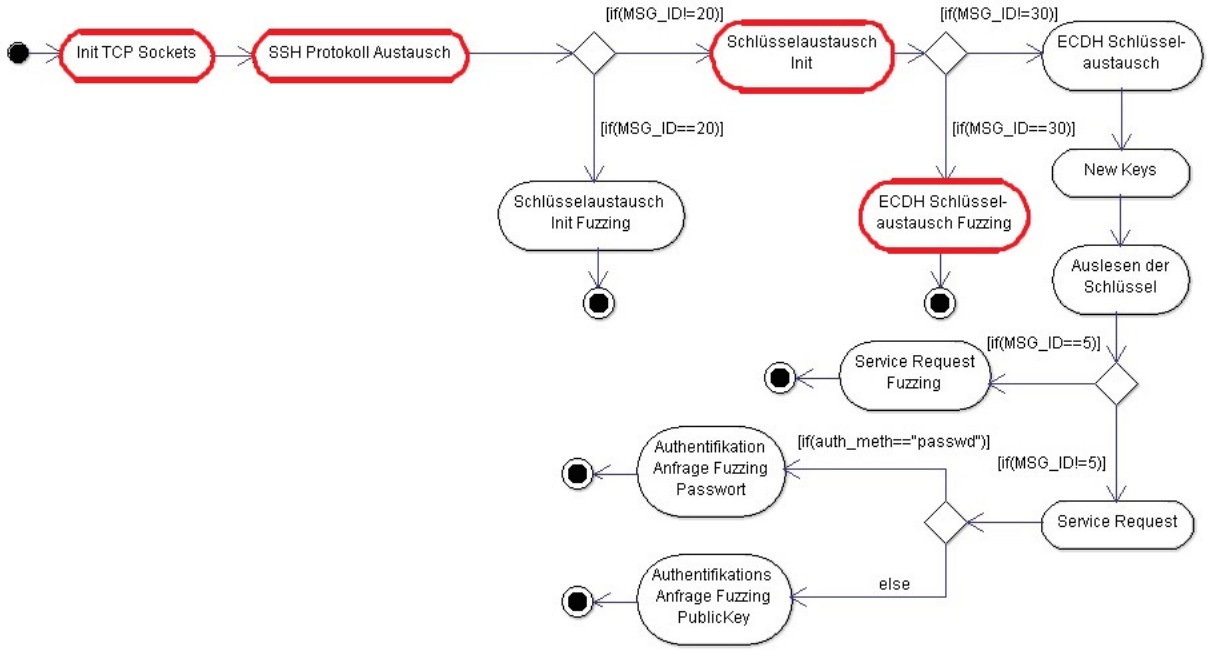

Abb. 5.4: Test 2: ECDH Key Exchange Init Fuzzing

Tab. 5.2: Testfall 2 Daten

Feld	Wert
SSH MSG ID	30
Pos. Fuzzing	150
Länge Fuzzing	10
Fuzzingdaten	aaaaaaaaaa
Art des Fuzzings	replace

eingebaut (grün eingerahmt). Da bei diesem Test nur Daten ersetzt worden sind, muss kein Längenfeld angepasst werden. Damit ist dieser Test bestanden.

```
      00 00 00 4c 05 1e  00 00 00 41 04 9d cc 07
13 6f fa 08 9e b6 d7 fe  d8 14 80 57 4b a4 73 52
48 a8 a2 9d 3c 76 ea c5  2c 8a 70 00 bb 11 d4 77
eb fa 30 82 20 d6 3f 87  cd fb 24 08 de a4 bb 32
be 4b b8 97 1a d4 ac 73  08 fc 33 47 dd aa aa aa
aa aa aa aa aa aa aa
```

Abb. 5.5: Test 2: Ausschnitt des Wiresharkmitschnittes

5.1.3 Test des SSH Service Request Fuzzings

Dieser Test wird die in Abbildung 5.6 gezeigten Codeblöcke des Interceptors abdecken und auf seine Richtigkeit überprüfen.

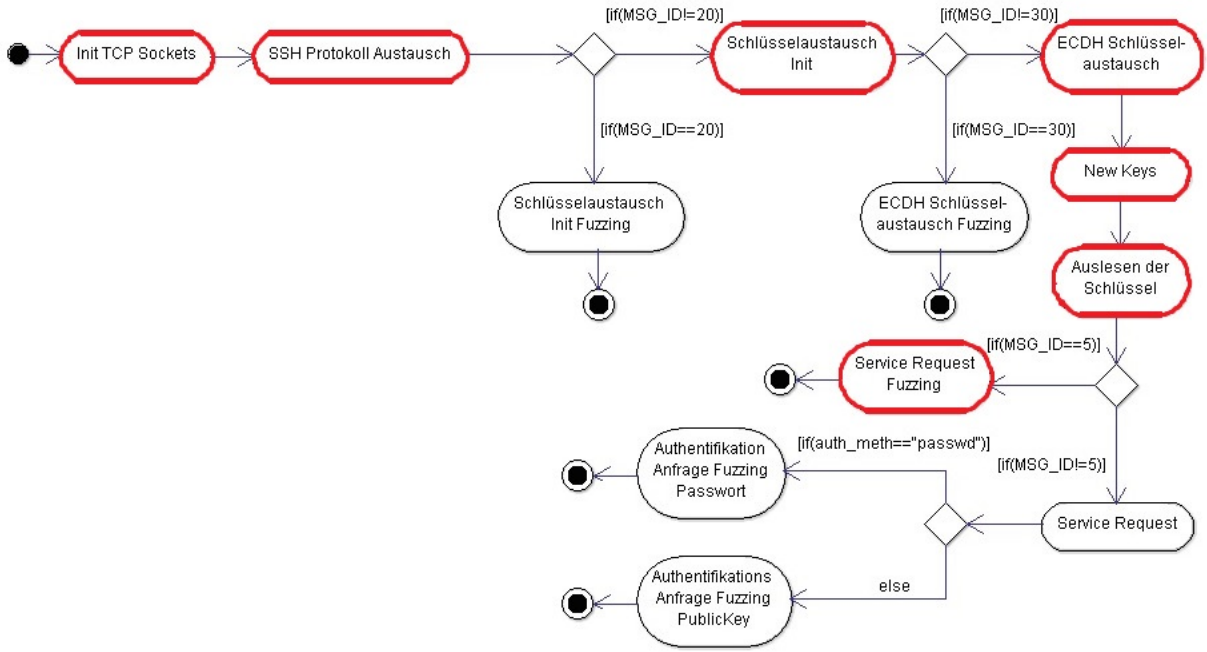

Abb. 5.6: Test 3: Service Request Fuzzing

Im dritten Testfall wird der Fuzzer mit den in Tabelle 5.3 gezeigten Daten aufgerufen.

Tab. 5.3: Testfall 3 Daten

Feld	Wert
SSH MSG ID	5
Pos. Fuzzing	50
Länge Fuzzing	14
Fuzzingdaten	56565656787878
Art des Fuzzings	replace

Das erwartete Ergebnis ist, dass der Interceptor in der Service Request Nachricht die Fuzzingdaten 56565656787878 beginnend an der Position 50 bis zur Position 63 ersetzt.

In diesem Testfall kann das Ergebnis des Tests nicht mehr in Wireshark nachvollzogen werden, da die gefuzzte Nachricht verschlüsselt ist. Die verschlüsselte Nachricht ist mit der Funktion *testsuitedecrypt()* entschlüsselt worden. Die entschlüsselte Nachricht und somit das Ergebnis des Tests ist in Abbildung 5.7 zu sehen.

Hierbei hat der Interceptor den Fuzzingwert in der richtigen Nachricht (rot eingerahmt) und an der richtigen Stelle eingebaut (grün eingerahmt). Da bei diesem Test nur Daten ersetzt worden sind, muss kein Längenfeld angepasst werden. Damit ist dieser Test bestanden.

```
0000001C0A050000000C7373682D757365726175746 85C5
336565656567878711389B1B96B0718E9AC015751A4621
DE
```

Abb. 5.7: Test 3: Ausschnitt der entschlüsselten Daten

5.1.4 Test des SSH Userauth Request Passwort Fuzzings

Dieser Test wird die in Abbildung 5.8 gezeigten Codeblöcke des Interceptors abdecken und auf seine Richtigkeit überprüfen.

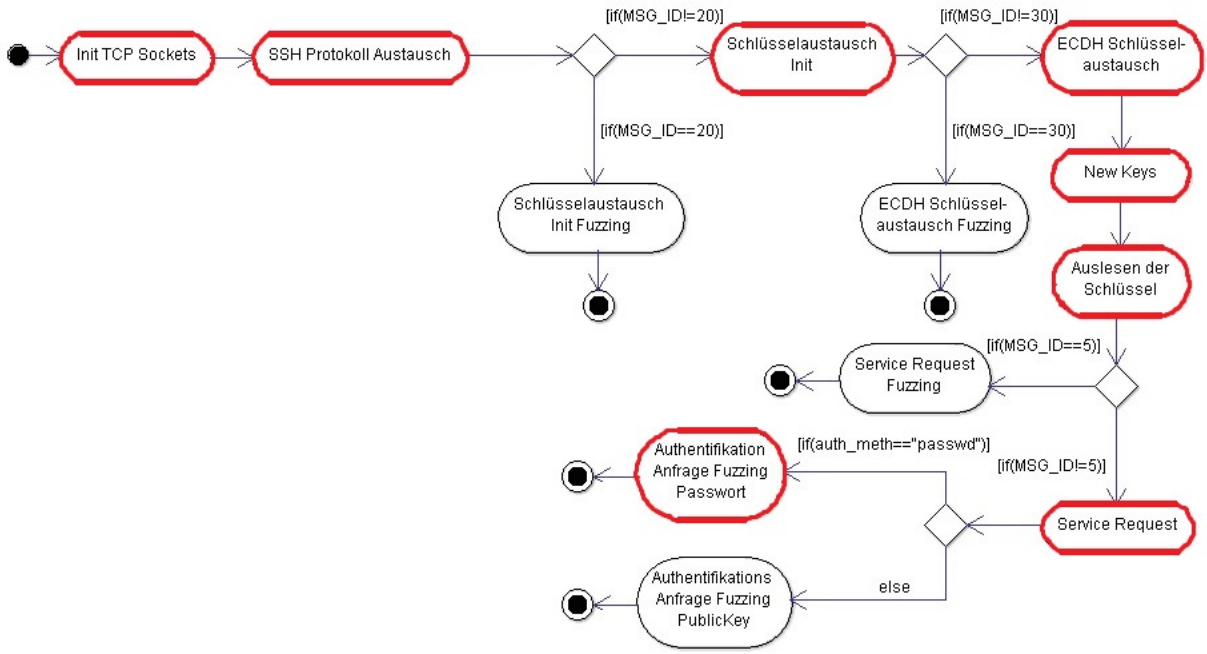

Abb. 5.8: Test 4: Userauth Request Passwort Fuzzing

Im vierten Testfall wird der Fuzzer mit den in Tabelle 5.4 gezeigten Daten aufgerufen.

Das erwartete Ergebnis ist, dass der Interceptor in der Userauth Request Nachricht die Fuzzingdaten 56565656787878 beginnend an der Position 230 bis zur Position 243 ersetzt.

In diesem Testfall kann das Ergebnis des Tests nicht mehr in Wireshark nachvollzogen werden, da die gefuzzte Nachricht verschlüsselt ist. Die verschlüsselte Nachricht ist mit der

Tab. 5.4: Testfall 4 Daten

Feld	Wert
SSH MSG ID	50
Pos. Fuzzing	230
Länge Fuzzing	14
Fuzzingdaten	56565656787878
Art des Fuzzings	replace
Auth. Methode	passwd

Funktion *testsuitedecrypt()* entschlüsselt worden. Die entschlüsselte Nachricht und somit das Ergebnis des Tests ist in Abbildung 5.9 zu sehen.

Der Interceptor hat den Fuzzingwert in der richtigen Nachricht (rot eingerahmt) und an der richtigen Stelle eingebaut (grün eingerahmt). Da bei diesem Test nur Daten ersetzt worden sind, muss kein Längenfeld angepasst werden. Damit ist dieser Test bestanden.

```
0000007C39320000000D737368706173737646175746800
000000E7373682D636F6E6E656374696F6E000000087061
7373776F7264000000000D737368706173737646175746
8563FC58ABBA64B486B7785A0575E62D4DF603F726B9E6E
A495ADC38252592E68F8EA112C29CDBC3D1DCF50A356565
656787878 1D28B3D8DBBA456739F77C15855C9EA1EBE18C
095DF3
```

Abb. 5.9: Test 4: Ausschnitt der entschlüsselten Daten

5.1.5 Test des SSH Userauth Request Publickey Fuzzings

Dieser Test wird die in Abbildung 5.10 gezeigten Codeblöcke des Interceptors abdecken und auf seine Richtigkeit überprüfen.

Im fünften Testfall wird der Fuzzer mit den in Tabelle 5.5 gezeigten Daten aufgerufen.

Das erwartete Ergebnis ist, dass der Interceptor in der Userauth Request Nachricht die Fuzzingdaten 0c0c beginnend an der Position 20 bis zur Position 23 ersetzt.

In diesem Testfall kann das Ergebnis des Tests nicht mehr in Wireshark nachvollzogen werden, da die gefuzzte Nachricht verschlüsselt ist. Die verschlüsselte Nachricht ist mit der Funktion *testsuitedecrypt()* entschlüsselt worden. Die entschlüsselte Nachricht und somit das Ergebnis des Tests ist in Abbildung 5.11 zu sehen.

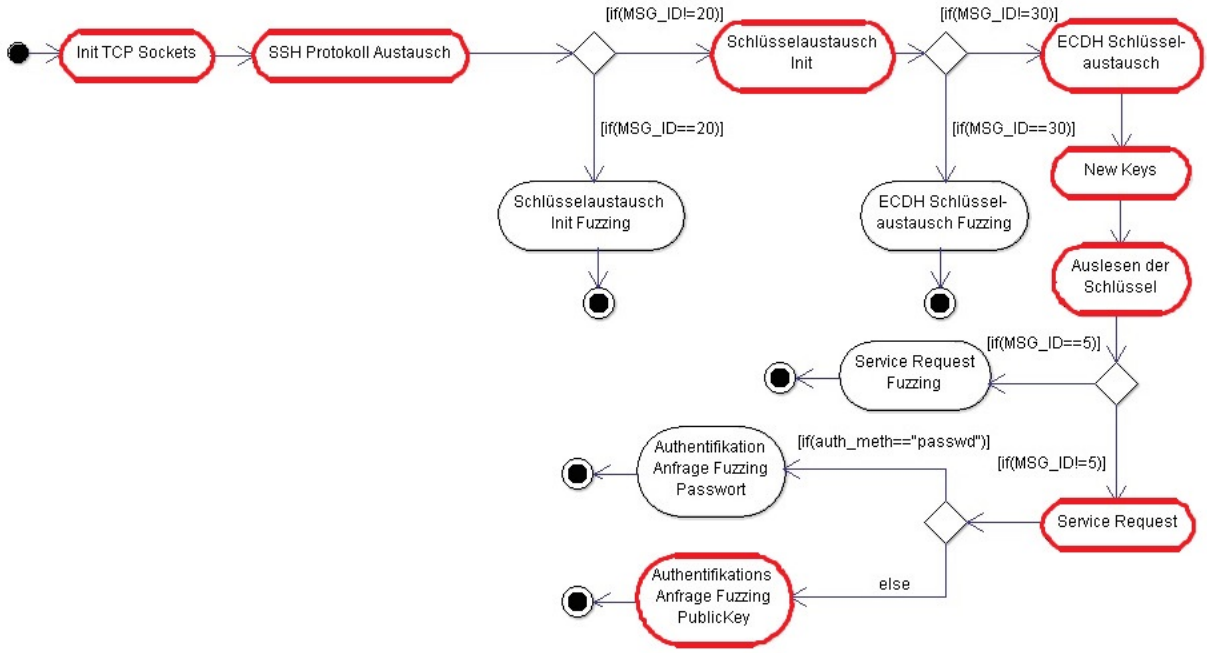

Abb. 5.10: Test 5: Userauth Request Publickey Fuzzing

Tab. 5.5: Testfall 5 Daten

Feld	Wert
SSH MSG ID	50
Pos. Fuzzing	20
Länge Fuzzing	4
Fuzzingdaten	0c0c
Art des Fuzzings	replace
Auth. Methode	-

Der Interceptor hat die Fuzzingdaten in der richtigen Nachricht (rot eingerahmt) und an der richtigen Stelle eingebaut (grün eingerahmt). Da bei diesem Test nur Daten ersetzt worden sind, muss kein Längenfeld angepasst werden. Der Test ist damit bestanden.

```
0000027C1032 0000000D 0c0c…
```

Abb. 5.11: Test 5: Ausschnitt der entschlüsselten Daten

5.1.6 Test mit falscher MSG-ID

Für den Test des Interceptors wird eine falsche MSG-ID dem Interceptor übergeben. Da der Interceptor nur die MSG-ID 5, 20, 30 und 50 fuzzen kann, sind auch nur diese für den Aufruf des Fuzzers zugelassen. Im Test wird die MSG-ID 4 dem Interceptor übergeben.

Das erwartetete Ergebnis des Tests ist, dass der Interceptor die falsche MSG-ID erkennt und sich beendet. Zusätzlich muss der Interceptor eine Ausgabe erzeugen, in der enthalten ist, dass der Wert der MSG-ID nicht in Ordnung gewesen ist. Der Interceptor darf die gefuzzte Nachricht nicht versenden.

Während des Tests wird ein Wiresharkmitschnitt erstellt, damit geprüft werden kann, ob keine Nachricht durch den Interceptor versendet wird.

Der Wiresharkmitschnitt zeigt, dass keine Pakete vom Interceptor versendet wurden.

Zusätzlich ist in Listing 5.1 die Ausgabe über die Konsole zu sehen. Hier ist die Information enthalten, dass die MSG-ID nicht korrekt ist.

List. 5.1: Error falsche MSG-ID

```
1 Error: falsche MSG ID
```

Das erwartete Ergebnis stimmt mit dem gemessenen Ergebnis überein und das Abfangen einer falschen MSG-ID ist erfolgreich.

5.1.7 Test mit falscher Fuzzingdatenlänge

Für den Test des Interceptors wird eine falsche Fuzzingdatenlänge dem Interceptor übergeben. Die angegebene Länge muss mit der wirklichen Länge der Fuzzingdaten verglichen werden. Um dies zu prüfen, wird dem Interceptor eine Länge der Fuzzingdaten von sieben und Fuzzingdaten mit der Länge von acht übergeben.

Das erwartetete Ergebnis des Tests ist, dass der Interceptor die falsche Lägenangabe erkennt und sich beendet. Zusätzlich muss der Interceptor eine Ausgabe erzeugen, in der enthalten ist, dass der Wert der Längenangabe nicht in Ordnung gewesen ist. Der Interceptor darf keine Nachrichten versenden.

Während des Tests wird ein Wiresharkmitschnitt erstellt, damit geprüft werden kann, ob keine Nachricht durch den Interceptor versendet wird.

Der Wiresharkmitschnitt zeigt, dass keine Pakete vom Interceptor versendet wurden.

Zusätzlich ist in Listing 5.2 die Ausgabe über die Konsole zu sehen. Hier ist die Information enthalten, dass die MSG-ID nicht korrekt ist.

List. 5.2: Error falsche Fuzzingdaten

```
1 Error: Länge der Fuzzingdaten stimmt nicht mit der Längenangabe überein.
```

Das erwartete Ergebnis stimmt mit dem gemessenen Ergebnis überein und das Abfangen einer falschen Fuzzingdatenlänge ist erfolgreich.

5.1.8 Test mit falscher insert_or_replace Angabe

Für den Test des Interceptors wird eine falsche insert_or_replace Angabe dem Interceptor übergeben. Die Variable *insert_or_replace* darf nur die Werte insert oder replace besitzen. Die Prüfung der Variable wird durch diesen Test verifiziert. Als Testwert dient der Wert “aaaaaa”.

Das erwartetete Ergebnis des Tests ist, dass der Interceptor den falschen Wert der *insert_or_replace* Variable erkennt und sich beendet. Zusätzlich muss der Interceptor eine Ausgabe erzeugen, in der enthalten ist, dass der Wert der *insert_or_replace* Variable nicht in Ordnung gewesen ist. Der Interceptor darf keine Nachrichten versenden.

Während des Tests wird ein Wiresharkmitschnitt erstellt, damit geprüft werden kann, ob keine Nachricht durch den Interceptor versendet wird.

Der Wiresharkmitschnitt zeigt, dass keine Pakete vom Interceptor versendet wurden.

Zusätzlich ist in Listing 5.3 die Ausgabe über die Konsole zu sehen. Hier ist die Information enthalten, dass die MSG-ID nicht korrekt ist.

List. 5.3: Error falsches insert or replace

```
1 Error: Insert oder replace wurde nicht richtig angegeben
```

Das erwartete Ergebnis stimmt mit dem gemessenen Ergebnis überein und das Abfangen eines falschen Wertes der Variable *insert_or_replace* ist erfolgreich.

5.1.9 Test mit falscher auth_method Angabe

Für den Test des Interceptors wird eine falsche auth_method Angabe dem Interceptor übergeben. Die Variable *auth_method* darf entweder keinen oder den Wert "passwd:” enthalten. Die Prüfung der Variable wird durch diesen Test verifiziert. Als Testwert dient der Wert "bbbbbb”.

Das erwartetete Ergebnis des Tests ist, dass der Interceptor den falschen Wert der *auth_method* Variable erkennt und sich beendet. Zusätzlich muss der Interceptor eine Ausgabe erzeugen,

in der enthalten ist, dass der Wert der *auth_method* Variable nicht in Ordnung gewesen ist. Der Interceptor darf keine Nachrichten versenden.

Während des Tests wird ein Wiresharkmitschnitt erstellt, damit geprüft werden kann, ob keine Nachricht durch den Interceptor versendet wird.

Der Wiresharkmitschnitt zeigt, dass keine Pakete vom Interceptor versendet wurden.

Zusätzlich ist in Listing 5.4 die Ausgabe über die Konsole zu sehen. Hier ist die Information enthalten, dass die MSG-ID nicht korrekt ist.

List. 5.4: Error falsche auth_method

```
1 Error: Angabe der Authmethode stimmt nicht
```

Das erwartete Ergebnis stimmt mit dem gemessenen Ergebnis überein und das Abfangen eines falschen Wertes der Variable *auth_method* ist erfolgreich.

5.1.10 Test mit zu großer Positionsangabe

Der Test für die Angabe der zu fuzzenden Position kann erst dann durchgeführt werden, wenn die zu fuzzende Nachricht empfangen ist. Bevor die Fuzzingdaten eingebaut werden, wird überprüft, ob die angegebene zu fuzzende Position größer ist als die eigentliche Länge der empfangenen Nachricht. Für den Test dieser Überprüfung wird eine Position mit dem Wert 4000 übergeben.

Das erwartetete Ergebnis des Tests ist, dass der Interceptor den zu großen Wert der Variable *pos_fuzz* erkennt und sich beendet. Zusätzlich muss der Interceptor eine Ausgabe erzeugen, in der enthalten ist, dass der Wert der *pos_fuzz* Variable nicht in Ordnung gewesen ist. Der Interceptor darf die zu fuzzenden Daten nicht in den empfangenen String einbauen.

Während des Tests wird ein Wiresharkmitschnitt erstellt, damit geprüft werden kann, ob keine Nachricht durch den Interceptor versendet wird.

Der Wiresharkmitschnitt zeigt, dass keine Pakete vom Interceptor versendet wurden.

Zusätzlich ist in Listing 5.5 die Ausgabe über die Konsole zu sehen. Hier ist die Information enthalten, dass die MSG-ID nicht korrekt ist.

List. 5.5: Error falsche Fuzzingposition

```
1 Error: Angegebene Position ist größer als die empfangene Nachricht
```

Das erwartete Ergebnis stimmt mit dem gemessenen Ergebnis überein und das Abfangen eines zu großen Wertes der Variable *pos_fuzz* ist erfolgreich.

5.2 Test der Verbindungsüberprüfung

Für den kompletten Test der Verbindungsüberprüfung müssen drei verschiedene Tests durchgeführt werden. Der erste der drei Tests beinhaltet den Fall, dass das SSH-Serversystem sowohl den Port 22 geöffnet hat. Auf diesem Port wird eine funktionsfähige OpenSSH Implementierung auf eine Verbindung warten. Der erwartete Ausgang des Tests ist, dass sowohl der Portcheck, als auch der SSH-Verbindungscheck erfolgreich sind. Somit müssen beide Variablen den Wert pass enthalten.

Der Aufruf der Verbindungsüberprüfung unter den vorliegenden Bedingungen ergab die in Tabelle 5.6 dargestellten Ergebnisse.

Tab. 5.6: Ergebnis Test 1 der Verbindungsüberprüfung

Test	Ergebnis
Socket Test	pass
SSH-Test	pass

Das Ergebnis des ersten Tests entspricht der gegebenen Erwartungshaltung.

Der zweite der drei Tests beinhaltet den Fall, dass das SSH-Serversystem den Port 22 geschlossen hat. Der erwartete Ausgang des Tests ist, dass sowohl der Portcheck, als auch der SSH-Verbindungscheck nicht erfolgreich sind. Somit müssen beide Variablen den Wert fail enthalten.

Der Aufruf der Verbindungsüberprüfung unter den vorliegenden Bedingungen ergab die in Tabelle 5.7 dargestellten Ergebnisse.

Tab. 5.7: Ergebnis Test 2 der Verbindungsüberprüfung

Test	Ergebnis
Socket Test	fail
SSH-Test	fail

Das Ergebnis des zweiten Tests entspricht der gegebenen Erwartungshaltung.

Der letzte Tests beinhaltet den Fall, dass das SSH-Serversystem den Port 22 geöffnet hat, aber keine gültige SSH-Verbindung über Port 22 aufgebaut werden kann. Der erwartete Ausgang des Tests ist, dass der Portcheck erfolgreich und der SSH-Verbindungscheck nicht erfolgreich ist. Somit muss die Portcheck-Variable den Wert pass und die SSH-Verbindungscheck-Variable den Wert fail enthalten.

Tab. 5.8: Ergebnis Test 3 der Verbindungsüberprüfung

Test	Ergebnis
Socket Test	pass
SSH-Test	fail

Der Aufruf der Verbindungsüberprüfung unter den vorliegenden Bedingungen ergab die in Tabelle 5.8 dargestellten Ergebnisse.

Das Ergebnis des dritten Tests entspricht der gegebenen Erwartungshaltung. Somit erfüllen alle drei ausgeführten Tests die Erwartungshaltung und der Block der Verbindungsüberprüfung wird als funktionsfähig deklariert.

5.3 Test der Ressourcenüberwachung

Für den kompletten Test der Ressourcenüberwachung sind drei verschiedene Einzeltests nötig. Im ersten Test wird überprüft, ob das Shellskript auf dem SSH-Serversystem richtig arbeitet. Der zweite Test verifiziert, dass die Funktion der Ressourcenauswertung auf Seiten des Clients richtig funktioniert. Der dritte Test beinhaltet eine fehlerhafte geschriebene Ressourcendatei. Mit diesem Test wird überprüft, ob die Ressourcenüberwachung einen falschen Parameter erkennt und basierend darauf keine Auswertung vornimmt.

Der Test selbst ruft die Funktion *ressourcenueberwachung()* in der Perl-Datei *ressourcenueberwachung.pm* auf. Über diese Funktion werden die Tests der Ressourcenüberwachung gestartet.

Der erste Test beinhaltet, dass festgestellt wird, ob die Ausgabe des Befehls top durch das Shellskript richtig gefiltert wird. Die erwartete Ausgabe der Filterung sollte wie folgt aussehen:

2576K 0.00%

Die Filterung des Befehls durch das Shellskript erzeugt folgende Ausgabe:

2576K 0.00%

Somit entspricht der gefilterte Wert der erwarteten Ausgabe.

Das Schreiben der Informationen in die Datei ressource_sshd unter dem Verzeichnis */tmp* ist durch die Ausgabe der geschriebenen Datei mit dem Befehl more verifiziert worden. Die Ausgabe der Datei ist in Abbildung 5.12 zu sehen.

Der zweite Test betreffend der Ressourcenüberwachung ist die Verifikation der Funktionalität des Perlskripts auf dem Fuzzersystem. Hierfür werden jeweils zwei verschiedene

```
more ressource_sshd
2576K 0.00%
```

Abb. 5.12: Inhalt der Datei ressource_sshd

ausgelesene Werte an die Funktion übergeben. Der erste übergebene Wert ist unterhalb des definierten Grenzwertes und der zweite liegt darüber. Im ersten Fall ist die Erwartungshaltung, dass eine eins zurück gegeben wird und die ausgelesenen Werte korrekt zurück gegeben werden. Für den zweiten Fall ist die Erwartungshaltung, dass zum einen eine null zurück gegeben wird und zum anderen die ausgelesenen Werte korrekt zurück gegeben werden.

Der Aufruf der Funktion mit dem ersten Fall erzeugte folgenden Rückgabewert.

1 0.00 2376

Der Aufruf der Funktion mit dem zweiten Fall erzeugte folgenden Rückgabewert.

0 0.00 2900

Beide Ausgaben entsprechen den erwarteten Ausgaben.

Für den dritten Test wird der in Listing 5.6 gezeigte Inhalt in die Datei ressource_read auf dem SSH-Server geschrieben.

List. 5.6: Inhalt korrupte ressourcen_sshd

```
1 23fg 0.10\%
```

Das erwartete Ergebnis dieses Tests ist, dass die Ressourcenüberwachung einen Error ausgibt und die Funktion der Ressourcenüberwachung beendet wird. Diese Errorausgabe muss den String Ërror: Speicherressource ist keine natürliche Zahl” enthalten.

List. 5.7: Error Ressourcenauswertung

```
1 Error: Speicherressource ist keine natürliche Zahl
```

Das Listing 5.7 zeigt die Ausgabe der Funktion *ressourcenueberwachung()* nachdem der Test ausgeführt wurde. Die Errorausgabe der Funktion zeigt, dass die korrupte natürliche Zahl richtig erkannt wurde. Der Test ist damit bestanden.

5.4 Test der Loggingauswertung

Für den Test der Loggingauswertung des SSH-Fuzzers wird die Datei authlog, deren Inhalt vorher bekannt ist, durch den Block der Loggingauswertung gefiltert. Der Auszug der Datei ist in Listing 5.8 zu sehen.

Der Test selbst ruft die Funktion *authlog_auslesen()* in der Perl-Datei *logging_auswertung.pm* auf. Über diese Funktion werden die Tests der Ressourcenüberwachung gestartet.

List. 5.8: Ungefilterte authlog-Datei

```
1  May 28 22:24:22 SSH-Server sshd [26619]: Accepted pubkey for sshtest
2  from 192.168.222.134 port 59029 ssh2: RSA fc:d5:77:95:ed:a8:09:b3:37:0f:b0:1e:27:30:6d:97
3  May 28 22:24:22 SSH-Server sshd [26619]: User child is on pid 9382
4  May 28 22:24:22 SSH-Server sshd [9382]: Starting session: command for
5  sshtest from 192.168.222.134 port 59029
6  May 28 22:24:22 SSH-Server sshd [9382]: Received disconnect from
7  192.168.222.134: 11: disconnected by user
8  May 28 22:24:22 SSH-Server sshd [20049]: Did not receive
9  identification string from 192.168.222.134
10 May 28 01:06:42 SSH-Server sshd [6587]: Connection from 192.168.222.134 port 59053
11 May 28 01:06:55 SSH-Server sshd [6587]: fatal: Incorrect size for
12 server Curve 25519 pubkey: 0 [preauth]
```

Als Ergebnis der Loggingauswertung wird erwartet, dass alle geloggten Informationen mit der Prozess-ID 6587 zurück gegeben werden. Dies sind im Testfall die letzten zwei Zeilen.

List. 5.9: gefilterte authlog-Datei

```
1  May 28 01:06:42 SSH-Server sshd [6587]: Connection from 192.168.222.134 port 59053
2  May 28 01:06:55 SSH-Server sshd [6587]: fatal: Incorrect size for
3  server Curve 25519 pubkey: 0 [preauth]
```

In Listing 5.9 ist das Ergebnis der Loggingauswertung zu sehen. Dieses Ergebnis entspricht der Erwartungshaltung zu diesem Test. Somit funkioniert die Loggingauswertung und kann für den SSH-Fuzzer verwendet werden.

5.5 Test des Loggings

Der Test des Loggings der gewonnen Informationen teilt sich in zwei einzelne Testfälle auf. Hierbei wird zuerst das Erstellen der Ordnerstruktur überprüft und als nächstes erfolgt die Überprüfung der Erstellung der Textdatei zu jedem Testfall und der CSV-Datei, die die gewonnen Informationen des Fuzzers zentral speichert.

Zur Überprüfung der Erstellung der Ordnerstruktur wird unter dem Verzeichnis */home/test/Fuzzing* die Ordnerstruktur testweise erstellt. Die Funktion der Erstellung der

Ordnerstruktur wird zweimal aufgerufen, um zu prüfen, dass der zweite Aufruf den gewollten Unterordner im gleichen Hauptverzeichnis erstellt.

Zuerst erfolgt der Testaufruf der Funktion mit der MSG ID 20 und darauf mit der MSG ID 30. Das erwartete Ergebnis ist, dass sich durch den Aufruf ein Hauptverzeichnis mit jeweils zwei Unterverzeichnissen erstellt. Die Unterverzeichnisse müssen die Namen *SSH_MSG_KEXINIT* und *SSH_MSG_DIFFIE_HELLMANN_KEX_INIT* tragen.

In Listing 5.10 ist das Ergebnis des Aufrufs auf der Kommandozeile durch Aufruf des Befehls ls zu sehen. Dieses Ergebnis entspricht dem erwarteten Ergebnis des Tests und somit kann der Funktionsblock der Ordnererstellung in dem SSH-Fuzzer verwendet werden.

List. 5.10: Test Ordnerstruktur

```
1 SSH_MSG_DIFFIE_HELLMANN_KEX_INIT    SSH_MSG_KEXINIT
```

Für die Überprüfung des korrekten Loggings in Form einer Textdatei und einer CSV Datei werden der Funktion zur Erstellung dieser beiden Dateien Dummyinformationen übergeben. Diese Dummyinformationen sind in Tabelle 5.9 festgehalten.

Tab. 5.9: Dummyinformationen

Typ der Info	Wert
SSH-Nachrichtentyp	20
Fuzzingdaten	2c2c2c2c2c2c2c
gefuzzte Nachricht	DUMMY-gefuzzte Nachricht
Ergebnis Portcheck	pass
Ergebnis SSH-Check	pass
Auszug Logging sshd	DUMMY-Logging
Prozessauslastung	0.10%
Speicherauslastung	2516

Die durch diese Dummyinformationen erzeugte Textdatei und CSV Datei sind jeweils in Listing 5.11 und Listing 5.12 zu sehen. Die erzeugte Textdatei und CSV Datei entsprechen der Erwartungshaltung des Tests.

List. 5.11: Inhalt der Textdatei

```
1 ################################################################################
2 ########################SSH-Fuzzing-Test########################################
3 Testfall Nr.1                          Datum:27.5.2015
4                                        Uhrzeit:18:47:13
5
6 ################################################################################
7
8 Fuzzing Informationen:
```

```
9
10 Gefuzzter SSH Nachrichtentyp    : 20
11 Position Fuzzingdaten           : 108
12 Fuzzingdaten                    : 2c2c2c2c2c2c2c
13 Gefuzzte Nachricht              : DUMMY-gefuzzte Nachricht
14
15 ##############################################################################
16 Verbindungsüberprüfung Ergebnisse:
17 Ergebnis Portcheck:                      pass
18 Ergebnis SSH-Verbindungsüberprüfung:     pass
19
20 ##############################################################################
21 Auszug der Loggingdatei des SSH-Daemons:
22 DUMMY-Logging
23
24 ##############################################################################
25 Auszug  des Ressourcenverbrauchs des SSH-Daemon:
26
27 SSH-Daemon Prozessorauslastung: 0.10\%
28 SSH-Daemon Speicherauslastung:  2516 KByte (Grenzwert nicht überschritten)
```

List. 5.12: Inhalt der CSV-Datei

```
1 1,2015.5.27,18:47:13,20,108,2c2c2c2c2c2c2c, DUMMY-gefuzzte
      Nachricht,pass,pass,DUMMY-Logging,0.10,2516
```

Somit kann der Funktionsblock des Loggings für den SSH-Fuzzer verwendet werden.

5.6 Test der Steuerlogik

Der Test der Steuerlogik bildet den Abschluss des Kapitels. In diesem Test wird nun das komplette System getestet. Insbesondere gilt es hier die Start- und Stoppwerte zu testen.

Hierfür werden die folgenden einzelnen Tests durchgeführt:

- Test mit allen verfügbaren Testfällen
- Test mit einem Startwert, aber keinem Stoppwert
- Test mit einem Stoppwert, aber keinem Startwert
- Test mit einem Start- und Stoppwert
- Test mit falschen Start- und Stoppwerten

Die Erwartungshaltung an das zu testende System ist, dass sobald ein Start- oder Stoppwert angegeben ist, das System diese Werte einhalten muss. Sobald kein Start- und Stopp-

wert angebenen ist, muss das System alle verfügbaren Testfälle, die sich in der JSON-Datei befinden, ausführen.

List. 5.13: Ausführung aller Tests

```
./fuzzer.pl
Testfall1....gestartet
Testfall1....erfolgreich
Testfall2....gestartet
Testfall2....erfolgreich
Testfall3....gestartet
Testfall3....erfolgreich
Testfall4....gestartet
Testfall4....erfolgreich
Testfall5....gestartet
Testfall5....erfolgreich
Testfall6....gestartet
Testfall6....erfolgreich
Testfall7....gestartet
Testfall7....erfolgreich
Testfall8....gestartet
Testfall8....erfolgreich
```

Listing 5.13 zeigt die Ergebnisse für den Test mit allen verfügbaren Testfällen. Zum Zeitpunkt des Tests befinden sich acht Testfälle in der Testfalldatenbank. Diese werden in diesem Test alle erfolgreich ausgeführt.

List. 5.14: Ausführung ab Startwert

```
./fuzzer.pl 3
Testfall3....gestartet
Testfall3....erfolgreich
Testfall4....gestartet
Testfall4....erfolgreich
Testfall5....gestartet
Testfall5....erfolgreich
Testfall6....gestartet
Testfall6....erfolgreich
Testfall7....gestartet
Testfall7....erfolgreich
Testfall8....gestartet
Testfall8....erfolgreich
```

Listing 5.14 zeigt die Ergebnisse für den Test mit einem Startwert, aber keinem Stoppwert. In diesem Fall liegt der angegbene Startwert bei drei. Somit bleiben für die Steuerlogik nur noch sechs Testfälle, die ausgeführt werden müssen. Die Steuerlogik muss Testfälle 3-8 ausführen. Dies wird durch die Steuerlogik ausgeführt. Somit gilt dieser Test als bestanden.

List. 5.15: Ausführung bis Stoppwert

```
./fuzzer.pl 1 5
Testfall1....gestartet
Testfall1....erfolgreich
```

```
4  Testfall2....gestartet
5  Testfall2....erfolgreich
6  Testfall3....gestartet
7  Testfall3....erfolgreich
8  Testfall4....gestartet
9  Testfall4....erfolgreich
10 Testfall5....gestartet
11 Testfall5....erfolgreich
```

Listing 5.15 zeigt die Ergebnisse für den Test mit einem Stoppwert, aber keinem Startwert. In diesem Fall liegt der angegbene Stoppwert bei fünf. Somit bleiben für die Steuerlogik nur noch fünf Testfälle, die ausgeführt werden müssen. Die Steuerlogik muss Testfälle 1-5 ausführen. Dies wird durch die Steuerlogik ausgeführt. Somit gilt dieser Test als bestanden.

List. 5.16: Ausführung ab Startwert und bis Stoppwert

```
1 ./fuzzer.pl 3 5
2 Testfall3....gestartet
3 Testfall3....erfolgreich
4 Testfall4....gestartet
5 Testfall4....erfolgreich
6 Testfall5....gestartet
7 Testfall5....erfolgreich
```

Listing 5.16 zeigt die Ergebnisse für den Test mit einem Stoppwert, und einem Startwert. In diesem Fall liegt der angegbene Startwert bei drei und der Stoppwert bei fünf. Somit bleiben für die Steuerlogik nur noch drei Testfälle, die ausgeführt werden müssen. Die Steuerlogik muss Testfall drei, vier und fünf ausführen. Dies wird durch die Steuerlogik ausgeführt.

Der anschließende Test der Steuerlogik beinhaltet die Eingabe von falschen Start- und Stoppwerten, die außerhalb der enthaltenen Testfälle der Datenbank liegen.

Das erwartete Ergebnis des Tests ist, dass die Steuerlogik die Überschreitung des Start- oder Stoppwertes erkennt, eine Fehlermeldung ausgibt und sich danach beendet.

Das gemessene Ergebnis ist in einem Auszug der Kommandozeile in Listing 5.17 und 5.18 zu sehen. Diese Auszüge zeigen die Ausgabe der Errormeldungen. Dieser Test gilt somit als bestanden.

List. 5.17: Error: Startwert größer Anzahl Testfälle

```
1 Error: Startwert ist größer als die Anzahl an möglichen Testfällen
```

List. 5.18: Error: Zahl kleiner oder gleich 0

```
1 Error: Stoppwert ist größer als die Anzahl an möglichen Testfällen
```

Der nächste Test der Steuerlogik beinhaltet die Eingabe einer Zahl für die Start- und Stoppwerte, die kleiner oder gleich der Zahl 0 ist.

Das erwartete Ergebnis des Tests ist, dass die Steuerlogik die falsche Eingabe erkennt, eine Fehlermeldung ausgibt und sich danach beendet.

Das gemessene Ergebnis ist in einem Auszug der Kommandozeile in Listing 5.19 zu sehen. Dieser Auszug zeigt die Ausgabe der Errormeldung. Dieser Test gilt somit als bestanden.

List. 5.19: Error: Zahl kleiner oder gleich 0

```
1 Error: 0 oder negative Zahl als Startwert definiert
```

Der letzte Test der Steuerlogik beinhaltet die Eingabe eines Stoppwertes, der kleiner als der Startwert ist.

Das erwartete Ergebnis des Tests ist, dass die Steuerlogik die falsche Eingabe erkennt, eine Fehlermeldung ausgibt und sich danach beendet.

Das gemessene Ergebnis ist in einem Auszug der Kommandozeile in Listing 5.20 zu sehen. Dieser Auszug zeigt die Ausgabe der Errormeldung. Dieser Test gilt somit als bestanden.

List. 5.20: Error: Stoppwert kleiner wie Startwert

```
1 Error: Stoppwert ist kleiner als Startwert
```

6 Ergebnisse

Im Kapitel Ergebnisse wird eine detaillierte Analyse und Bewertung der ausgeführten Testfälle durchgeführt. Zu der Analyse werden die gewonnen Informationen des SSH-Fuzzers genutzt und zusätzlich wird zu jedem Testfall ein Wiresharkmitschnitt erstellt. Es wird für jeden Testfall festgestellt, ob die gefuzzte Nachricht erfolgreich an den Server gesendet wurde. Die erwartete Reaktion des SSH-Servers ist in jedem Testfall, dass eine SSH-Disconnect Nachricht versendet wird. Die SSH-Disconnect Nachricht sollte vom SSH-Server versendet werden, so dass eine saubere Trennung der Verbindung auf Applikationsschicht geschieht. Zusätzlich sollte der SSH-Server die Verbindung auf TCP Ebene mit einem FIN,ACK beenden. Wird diese Erwartungshaltung nicht erfüllt, werden weiter Untersuchungen des speziellen Testfalles durchgeführt.

In den Testfällen 5-8 müssen die verschlüsselten gefuzzten SSH-Nachrichten und die verschlüsselten SSH-Disconnect Nachrichten für die Analyse entschlüsselt werden. Hierfür könnte das Tool *ssh_decoder* [23] von Julien Tinnes benutzt werden. Über dieses Tool ist es möglich, mitgeschnittene SSH-Sessions mit einem Brute Force Angriff auf den Schlüsselaustausch zu entschlüsseln. In dieser Arbeit wäre das Tool einsetzbar, wenn durch die ausgelesenen Schlüssel die mitgeschnittenen Daten entschlüsselt werden könnten. Diese Eigenschaft konnte nicht festgestellt werden. Deswegen wird für das Entschlüsseln der Daten die vorliegende Funktion *testsuitedecrypt()* aus der Implementierung genutzt. Diese Funktion wird manuell für jede zu entschlüsselnde Nachricht mit den passenden Schlüsseln aufgerufen. Als Resultat liefert diese Funktion die Nachricht im Klartext.

6.1 Testfälle ohne Fehlverhalten

In diesem Kapitel werden die Testfälle ohne Fehlverhalten analysiert und bewertet. Zur Analyse der Testfälle dienen die gewonnen Ergebnisse aus der Ressourcenüberwachung, Loggingüberwachung, Verbindungsüberprüfung und des Wiresharkmitschnittes. Alle Ergebnisse der jeweiligen Quelle werden einheitlich in einer Tabelle dargestellt. Die Abildungen der Ergebnisse sind im Anhang unter den jeweils angegebenen Kapiteln zu finden. Alle Ergebnisse sind auch unter dem Verzeichnis */Ergebnisse* zu finden.

Als erstes kann anhand der Inhalte der gefuzzten Nachrichten aus Tabelle 6.1 in der Spalte “Inhalt der Nachricht” die korrekte Ausführung der jeweiligen Testfälle festgestellt werden. Zusätzlich sind im Testfall 1 die SSH-Längenfelder richtig angepasst worden.

Der SSH-Server reagiert auf alle gefuzzten Nachrichten mit der Nachricht SSH-Disconnect. Die SSH-Disconnect Nachricht ist in Tabelle 6.1 in der Spalte “Verbindungsabbau” zu finden.

Nachdem die SSH-Disconnect Nachrichten versendet wurden, wird der geöffnete Socket der TCP Verbindung mit den FIN,ACK Nachrichten in jedem Testfall geschlossen. Der Interceptor auf Clientseite sendet die TCP-RST Nachricht erst nachdem die FIN,ACK Nachrichten des Servers versendet wurden. Somit werden die erzielten Ergebnisse nicht durch ein frühzeitiges Versenden der TCP-RST Nachricht verfälscht. Die FIN,ACK Nachrichten sind ebenfalls in der Tabelle 6.1 in der Spalte “Verbindungsabbau” zu finden.

Tab. 6.1: Wireshark Mitschnitte

Testfall	Inhalt der Nachricht	Verbindungsabbau
Testfall 1	Zu finden unter A.1	Zu finden unter A.2
Testfall 2	Zu finden unter A.3	Zu finden unter A.4
Testfall 3	Zu finden unter A.5	Zu finden unter A.6
Testfall 5	Zu finden unter A.9	Zu finden unter A.10
Testfall 6	Zu finden unter A.11	Zu finden unter A.12
Testfall 7	Zu finden unter A.13	Zu finden unter A.14
Testfall 8	Zu finden unter A.15	Zu finden unter A.16

Die Ergebnisse des Wiresharkmitschnittes spiegeln sich auch in den gewonnen Informationen des Fuzzers wieder. So loggt der SSH-Server in die Datei authlog in allen Testfällen ein Disconnect. Alle Ausschnitte der Loggingdateien sind in der Tablle 6.2 zu sehen.

Tab. 6.2: Authlogauswertung Ergebnisse

Testfall	Authlog-Datei
Testfall 1	Zu finden unter A.2
Testfall 2	Zu finden unter A.5
Testfall 3	Zu finden unter A.8
Testfall 5	Zu finden unter A.14
Testfall 6	Zu finden unter A.17
Testfall 7	Zu finden unter A.20
Testfall 8	Zu finden unter A.23

Die Ressourcenauswertung ergibt, dass in keinem der Testfälle die CPU Auslastung eine Auffälligkeit zeigt. Desweiteren liegen die ausgelesenen Grenzwerte der Speicherauslastung nicht zehn Prozent über dem definierten Grenzwert von 2550 Kilobyte. Alle Ergebnisse der Ressourcenauswertung sind in Tabelle 6.3 zusammengefasst.

Tab. 6.3: Ressourcenauswertung Ergebnisse

Testfall	Ressource Speicher	Ressource CPU	Grenzwert überschritten
Testfall 1	2348KB [A.1]	0,05% [A.1]	Nein [A.1]
Testfall 2	2360KB [A.4]	0,00% [A.4]	Nein [A.4]
Testfall 3	2368KB [A.7]	0,00% [A.7]	Nein [A.7]
Testfall 5	1952KB [A.13]	0,00% [A.13]	Nein [A.13]
Testfall 6	2024KB [A.16]	0,00% [A.16]	Nein [A.16]
Testfall 7	2044KB [A.19]	0,00% [A.19]	Nein [A.19]
Testfall 8	1972KB [A.22]	0,00% [A.22]	Nein [A.22]

Zudem ergibt die Verbindungsüberprüfung, dass der SSH-Server weiterhin erreichbar ist. Ein Ausschnitt aller gewonnenen Verbindungsinformationen sind in Tabelle 6.4 zu sehen.

Tab. 6.4: Verbindungsüberprüfung Ergebnisse

Testfall	Portcheck	SSH-Check
Testfall 1	OK [A.3]	OK [A.3]
Testfall 2	OK [A.6]	OK [A.6]
Testfall 3	OK [A.9]	OK [A.9]
Testfall 5	OK [A.15]	OK [A.15]
Testfall 6	OK [A.18]	OK [A.18]
Testfall 7	OK [A.21]	OK [A.21]
Testfall 8	OK [A.24]	OK [A.24]

Das Verhalten des SSH-Servers in allen Testfällen entspricht somit der definierten Erwartungshaltung. Damit zeigt die Analyse aller Testfälle, dass die OpenSSH-Implementierung in allen Fällen richtig reagiert und die Verbindung zum Fuzzer trennt. Alle Testfälle lösen somit kein fehlerhaftes Verhalten in der OpenSSH-Implementierung aus.

6.2 Testfälle mit Fehlverhalten

In Testfall 4 der Implementierung wird in der Nachricht SSH Diffie-Hellman Key Exchange des Clients das Längenfeld der Multi Precision Integerzahl e auf null gesetzt. Die Ergebnisse sind unter dem Verzeichnis *Ergebnisse/Testfall4* zu finden.

Abbildung A.7 zeigt die gefuzzte SSH SSH Diffie-Hellman Key Exchange Nachricht aus dem Wiresharkmitschnitt. In diesem Ausschnitt ist der veränderte Integerwert in dem

Längenfeld der SSH-Nachricht zu erkennen. Dieser wurde auf den Wert 0 (0x00000000) geändert.

Der SSH-Server reagiert auf die gefuzzte Nachricht nicht mit der Nachricht SSH-Disconnect. Diese ist nicht in Abbildung A.8 zu sehen.

Diese Abbildung zeigt den sequentiellen Abbau der SSH-Verbindung. Der SSH-Server versendet in diesem Testfall keine SSH-Disconnect Nachricht. Der geöffnete Port 22 wird aber dennoch über ein FIN,ACK, das vom Serversystem ausgeht, geschlossen. Das nicht Versenden der SSH-Disconnect Nachricht zeigt sich auch in dem Logging des SSH-Servers. Dieses ist in Abbildung A.11 zu erkennen.

Die Ressourcenauswertung ergibt, dass der CPU Wert bei 0,00 Prozent des Gesamtsystems liegt. Dies ist als nicht auffällig zu interpretieren. Desweiteren liegt der ausgelesene Grenzwert der Speicherauslastung nicht zehn Prozent über dem defnierten Grenzwert von 2550 Kilobyte. Beide Werte sind in Abbildung A.10 abgebildet.

Die Verbindungsüberprüfung in Listing A.12 zeigt, dass der SSH-Server nach diesem Testfall immer noch erreichbar ist.

Die Erwartungshaltung, dass eine SSH-Disconnect Nachricht versendet wird, ist nicht erfüllt worden. Daher müssen weiter Untersuchungen dieses Testfalles stattfinden. Dazu wird nun im folgenden der Quellcode der OpenSSH Implementierung der Version 6.6 herangezogen. In diesem Quellcode wird nun nach dem geloggten String *fatal: Incorrect size for server Curve25519 pubkey: 0* gesucht.

Die Suche ergibt, dass das Schreiben dieser Lognachricht in der Datei *kexc25519s.c* in Zeile 92 geschieht. Vor oder nach dieser Zeile wird aber wie keine SSH-Disconnect Nachricht versendet.

Testfall 4 hat somit ein fehlerhaftes Verhalten in der OpenSSH-Implementierung ausgelöst. Das fehlerhafte Verhalten wird als nicht kritisch eingestuft, da der SSH-Server die Verbindung trotzdem auf der TCP-Ebene trennt und keine Möglichkeit besteht, nicht erlaubte Daten aus der OpenSSH-Implementierung auszulesen.

7 Schluss

Zuerst werden hier die erzielten Ergebnisse der durchgeführten Arbeit aufgeführt und bewertet. Danach wird ein Ausblick auf mögliche Verbesserungen des Fuzzers auf theoretischer und praktischer Ebene gegeben.

7.1 Ergebnisse der Arbeit

Die Aufgabe eines Enwurfes und der Realisierung eines kryptographischen Fuzzingtools ist erfolgreich durchgeführt worden. Dazu wurden im Grundlagenteil durch Einarbeitung in die Themen Fuzzing und SSH allgemein gültige Möglichkeiten und Anforderungen für das Fuzzing eines kryptographischen Protokolls aufgestellt.

Die Möglichkeiten und Anforderungen wurden erfolgreich dazu genutzt, einen Entwurf für einen Fuzzer der OpenSSH-Implementierung anzufertigen. Dieser Entwurf beinhaltet sowohl Testfälle für den unverschlüsselten als auch für den verschlüsselten Phase des SSH-Protokolls. Die Testfälle dienen hierbei nicht dazu eine hundertprozentige Codeabdeckung zu erzielen, sondern erste Erfahrugen auf dem Gebiet des kryptographischen Fuzzings zu sammlen.

Die entstandene Implementierung eines Fuzzers zeigt, dass das Fuzzing einer Implementierung zu einem Mehrwert der Prüfung des Programmes führt. Somit kann das kryptographische Fuzzing einer Implementierung als sinnvoll erachtet werden. Durch das nicht Erfüllen einer definierten Erwartungshaltung eines Testfalles, müssen genauere Beobachtungen durchgeführt werden. Durch diese genaueren Beobachtungen erfährt man die Codeabdeckung eines Testfalles an einem praktischen Beispiel. Durch das Über- oder Unterschreiten eines Grenzwertes durch verschiedene Fuzzingdaten innerhalb eines Testfalles, konnten andere Fahrtengänge im zu testenden Code beobachtet werden.

Am Prototyp des SSH-Fuzzer sind Kritikpunkte anzubringen. Zum einen müssen die sleep-Funktionen im Interceptor und der Steuerungslogik durch andere Verfahren ersetzt werden, da so die Leistungsfähigkeit des Fuzzers erhöht werden kann. Zum anderen ist ein weiterer Kritikpunkt, dass in der Analyse und Bewertung der ausgeführten Testfälle die Mitschnitte des Netzwerks manuell durchzuführen waren.

7.2 Ausblick

In diesem abschließendem Kapitel soll ein Ausblick auf Basis der bisher gewonnenen Informationen auf weitere theoretische und praktische Ansätze gegeben werden.

7.2.1 Verschiedene Metrikansätze für den Fuzzer

Bevor die Ansätze zur Verbesserung der Metrik eines Fuzzers erarbeitet werden können, muss zunächst der Begriff Metrik in Zusammenhang mit einem Fuzzer erläutert werden. Im Allgemeinen bietet der Begriff der Metrik die Möglichkeit die Qualität eines Quellcodes zu bewerten. Im Zusammenhang mit einem Fuzzer drückt die Metrik die Güte der zu testenden Implementierung aus. Dadurch können verschiedene Fuzzer anhand ihrer gemessenen Metrik miteinander verglichen werden. Die folgende Metriken sollen im Zusammenhang mit einem Fuzzer betrachtet werden:

- Codeabdeckung der zu testenden Implementierung

- Analyse der zu testenden Schnittstelle, die Eingabedaten verarbeitet

- Messen der möglichen Muster der Eingabewerte

Die Metrik der Codeabdeckung basiert auf dem Artikel “Using code coverage to improve fuzzing results” von Lars Opstad [25]. Zur Verbesserung der Codeabdeckung durch einen Fuzzer wird in einem vorangestellten Schritt die Codeabdeckung, die von nicht gefuzzten Anfragen erzeugt wird, analysiert.

Dafür wird die Schnittstelle, die die später gefuzzten Anfragen entgegegen nimmt, in Codeblöcke unterteilt. Die Trennung der Blöcke erfolgt anhand von if-else oder switch-case Entscheidungen. An einem einfachem Beispiel soll dieses Prinzip veranschaulicht werden. Dazu ist die Unterteilung der Blöcke in Abbilung 7.1 dargestellt.

In diesem Beispiel wird der Code durch eine if-else Verzweigung aufgetrennt. Ingesamt ergeben sich somit sechs Blöcke. Bei Aufruf des Beispielcodes kann entweder der Fall auftreten, dass die Blöcke 1, 2, 3, 4 und 6 durchlaufen werden, oder dass die Blöcke 1, 2, 3, 5 und 6 durchlaufen werden. So braucht man auf jeden Fall beide Fälle, die durch normale Anfragen generiert werden, damit garantiert ist, dass alle Blöcke durchlaufen werden.

Sobald das Unterteilen des Programmcodes abgeschlossen ist, kann begonnen werden, normale Anfragen an das Programm zu senden. Während dieser Phase muss die Schnittstelle durch ein Codeabdeckungstool überwacht werden. Das Codeabdeckungstool überwacht zur Laufzeit des Programms die durchlaufenen Codeteile der Schnittstelle. Die durchlaufenen Codeteile werden später in Zusammenhang mit den ausgeführten Nachrichten gebracht. Diese Ergebnisse werden in einem späteren Bericht festgehalten. Die Grafik in Abbildung 7.2 zeigt eine beispielhafte Auswertung eines solchen Codeabdeckungstools.

```
x+=1;                    // block 1
y+=2;
if(
        sin(x)
         <               // block 2 (although executes after block 3)
         sin(y)   )      // block 3
{                        // block 4
         y+=1;
         result=true;
}
else                     // block 5
{
         result=false;
}
x+=1;                    // block 6
return result;
```

Abb. 7.1: Aufteilung des Codes in Blöcke

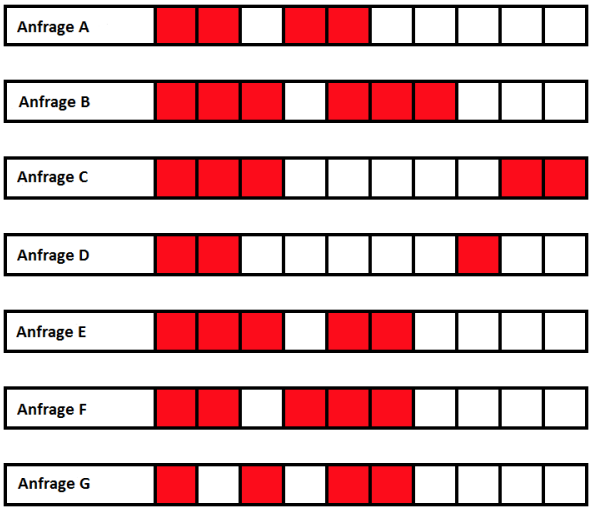

Abb. 7.2: Codeabdeckung der einzelnen Anfragen

In diesem Beispiel wurden insgesamt sieben Anfragen während der Überwachung der Schnittstelle gesendet. Dabei ist zu sehen, dass Anfrage B mit sechs von zehn durchlaufenen Blöcken die größte Codeabdeckung in der Schnittstelle erzeugt hat. Somit wird diese Anfrage als optimale Anfrage definiert. Die restlichen vier Blöcke werden durch weitere Anfragen abgedeckt. Diese fehlenden Blöcke werden in diesem Beispiel durch die Anfragen A, C, D und F abgedeckt. Die Anfragen E und G werden in diesem Schritt entfernt, da sie Blöcke durchlaufen, die durch die optimale Anfrage B abgedeckt sind. Anfrage F kann ebenfalls aus der Liste entfernt werden, da der benötigte Block 4 schon durch Anfrage A

abgedeckt ist. Auf Grund dieser Entscheidungen ergibt sich das in Abbildung 7.3 gezeigte Bild.

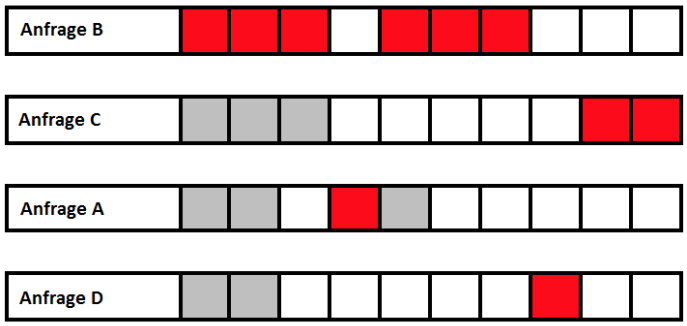

Abb. 7.3: Codeabdeckung durch optimierte Anfragen

Wenn die Analyse zeigt, dass durch die gültigen Anfragen keine vollständige Codeabdeckung erreicht wird, muss eine manuelle Untersuchung der nicht abgedeckten Codeblöcke erfolgen. Durch diese Untersuchung ergeben sich möglichweise weitere Anfragen, die den untersuchten Codeblock durchlaufen. Durch die strukturierte Analyse der maximal möglichen Codeabdeckung der gültigen Anfragen ist gewährleistet, dass diese Codeblöcke durch spätere mutierte Anfragen des Fuzzers abgedeckt sind. Auf Basis dieser gefilterten Anfragen können nun Testfälle mit beliebig vielen Fuzzingdaten für das Fuzzing der Schnittstelle entworfen werden. Beginnt man nun die veränderten Daten in die normalen Anfragen ohne weitere Überlegungen einzubauen, basiert der Fuzzer auf dem mutationsbasierten Ansatz.

Die Metrik der Codeabdeckung kann mit der Metrik der Analyse der zu fuzzenden Schnittstelle kombiniert werden. Nimmt man hierzu als Basis die gefilterten normalen Anfragen nach dem oben beschriebenen Prinzip der Codeabdeckung und führt anschließend eine Analyse der gefilterten Anfragen auf ihre spezifischen Felder und Datentypen durch, basiert der Fuzzer nun auf einem generationsbasierten Ansatz.

Die gewählte Metrik darf sich nicht allein auf die Codeabdeckung beziehen. Der entscheidene Punkt ist, dass durch viele Testfälle ein Codeblock beliebig oft durchlaufen werden kann, aber nur einer der ausgeführten Testfälle einen Fehler auslösen kann. Daher kann ein besserer Wert der Metrik erzielt werden, wenn zusätzlich gemessen wird wieviele Muster an Eingabeparameter der Fuzzer ausgeführt hat. Ein mögliches Muster der Eingabeparameter wären die Datentypen, die in den ausgeführten Testfällen verwendet werden. Je mehr Datentypen für ein zu fuzzendes Feld gewählt werden, desto besser ist auch der gemessene Wert der Metrik.

Ein weiterer möglicher Ansatz wäre, nochmals nicht veränderte Anfragen an die zu testende Schnittstelle eines Programms zu schicken und mit Hilfe eines Tools zu beobachten, welche Blöcke des Programmcodes wie oft aufgerufen werden. Basiernd auf den Ergebnissen dieser

Beobachtung können Blöcke, die öfters durch die Anfragen aufgerufen werden als andere Blöcke, zum späteren Fuzzing bevorzugt werden. Das Fuzzing der öfters aufgerufenen Blöcke kann die Wahrscheinlichkeit, ein Fehlverhalten auszulösen verbessern.

7.2.2 Mögliche Verbesserungen des SSH-Fuzzers

In der Ergebnisbewertung des SSH-Fuzzers sind offene zu verbessernde Punkte benannt worden. Zu diesen Punkten werden nun weitere Verbesserungsmöglicheiten vorgeschlagen. Dabei wird auf die folgenden Verbesserungen in den nächsten Unterkapiteln eingegangen:

- Implementierung von weiteren Testfällen
- Leistungsverbesserungen des SSH-Fuzzers
- Automatisierte Analyse der Fuzzingergebnisse
- Mitschnitt von gefuzzten Paketen

7.2.2.1 Implementierung von weiteren Testfällen

Die Entscheidung, welche zu generierenden Fuzzingdaten in welchen Testfall eingebaut werden, kann hierbei auf der Basis einer Datenflussanalyse geschehen.

Eine Datenflussanalyse beruht auf der statischen Code-Analyse eines Computerprogramms. Diese untersucht, zwischen welchen Teilen einer Implementierung Daten weitergegeben werden und welche Abhängigkeiten daraus resultieren. Die Datenflussanalyse kann dabei aus einem Kontrollflussgraphen enstehen. Diese Vorgehensweise wird dann Vorwärtsanalyse genannt. Die Datenflussanalyse liefert als Ergebnis eine Übersicht, welche Werte die Daten in den verschiedenen Blöcken enthält. Enthält ein Block beispielsweise den Code y=3, so verändern sich die Daten des Programms so, dass die Variable y nach dem Durchlaufen dieses Blocks den Wert 3 enthält, unabhängig davon, welchen bzw. ob die Variable einen Wert enthalten hat.

Eine manuelle Analyse des Datenflusses aller Variablen in einem komplexen Programm ist in einem überschaubaren zeitlichen Rahmen nicht durchführbar. Dafür kann eine automatisierte Analyse des Programmcodes durchgeführt werden.

Als Ergebnis der Datenflussanalyse ergeben sich zum Beispiel mehrfach aufgerufene Funktionen oder obsoleter Code, der niemals aufgerufen wird. Auf Basis der Ergebnisse bezüglich des Datenflusses der Variablen entstehen Fuzzingdaten für weitere Testfälle. Sobald

ein veränderter Datenfluss einer Variable ein anderes Verhalten der Implementierung bewirkt, wird ein neuer Testfall für den Fuzzer angelegt. Diese haben zur Folge, dass eine gleich bleibende Codeabdeckung mit weniger Testfällen erreicht wird.

Zur Erweiterung der Testfälle mit weiteren Fuzzingdaten, kann ein anderes Fuzzingtool zur Generierung von Fuzzingdaten dienen. Diese können zum Beispiel von einem vorhandenen Tool wie dem SPIKEfile[9] Fuzzingtool generiert werden.

Das Übernehmen der generierten Fuzzingdaten in eine Testfalldatenbank kann nur erfolgreich sein, wenn zusätzliche Parameter wie die SSH-MSG-ID, Postion der Fuzzingdaten, Länge der Fuzzingdaten und ob die Fuzzingdaten eingefügt oder ersetzt werden, sollen auch angegeben werden. Dies muss entweder manuell durchgeführt werden oder eine weitere Automatisierung eingeführt werden.

7.2.2.2 Leistungsverbesserungen des SSH-Fuzzers

Auf die genannten drei Sekunden pro durchzuführenden Testfall entfallen jeweils zwei Sekunden auf die Verwendung der sleep-Funktion.

Die erste Position, an der eine sleep-Funktion auftritt, ist bei dem Aufruf des SSH-Clients. Der SSH-Client wird keine Verbindung mit dem Interceptor aufbauen können, da der Interceptor den Port 5000 noch nicht geöffnet hat. Diese sleep-Funktion von einer Sekunde dient dem Interceptor, in dieser Zeit den Port zu öffnen.

Hierbei könnte auch durch den Interceptor signalisiert werden, dass die Steuerungslogik des Fuzzers nun eine SSH-Verbindung durch den SSH-Client aufbauen kann. Diese Methode wäre deutlich schneller, als die Verwendung der sleep-Funktion.

Die zweite Stelle, an der eine sleep-Funktion zum Einsatz kommt, befindet sich am Ende jedes Sendens einer gefuzzten Nachricht. Würde diese sleep-Funktion nicht zum Einsatz kommen, würde der Interceptor durch ein zu frühes Schließen des Sockets das Testergebnis verfälschen.

Wenn der SSH-Server nach einem Testfall mit einer SSH-Disconnect Nachricht und mit einem FIN,ACK auf der TCP-Ebene reagiert, dann kann dieses Signal vom Interceptor abgefangen werden und als Beenden der Verbindung interpretiert werden. Die Überprüfung, ob ein FIN,ACK durch den SSH-Server verschickt wurde, muss mehrfach stattfinden. Erst nach einer definierten Anzahl an negativen Prüfungen wird die Verbindung vom Interceptor getrennt. Dieses Szenario kommt zum Einsatz, wenn der Testfall bewirkt, dass der SSH-Server abstürzt.

Somit könnte eine Zeitersparnis von 66 Prozent erzielt werden, was im Falle von tausenden Testfällen einen enormen Unterschied ausmacht würde.

Desweiteren kann die Ausführung der Testfälle und die Überwachung der Reaktion des SSH-Servers parallelisiert werden. Bei der Parallelisierung der Testfälle ist zu beachten,

dass die Überwachung nun deutlich erschwert wird. Es muss jede Prozess-ID einer SSH-Verbindung dem Fuzzer mitgeteilt werden. Die bekannte Prozess-ID dient dazu, dass der Fuzzer die richtigen authlog Informationen filtert und eine korrekte Ressourcenauswertung durchführt. Im Falle von parallelisierten Testfällen gestaltet sich die Reproduzierbarkeit eines Fehlers, wenn eine Verkettung von Testfällen zu einem Fehlverhalten geführt hat, als schwierig. Dies liegt daran, dass es im Falle der Parallelisierung keine festgelegte Reihenfolge der Testfälle gibt.

7.2.2.3 Automatisierte Analyse der Fuzzingergebnisse

Die Untersuchung der acht ausgeführten Testfälle auf nicht Einhalten der Erwartungshaltung ist in diesem kleinen Maß nicht sehr zeitaufwendig. Doch kommt es zu dem Fall, dass die Testfälle in den Bereich von tausenden Testfällen liegen, dann wird sich die Untersuchung der Eregbnisse des Fuzzers als schwierig erweisen.

Die Idee wäre, dass die erstellte CSV-Datei auf enthaltene Werte untersucht wird. Die definierten Werte, die in einem Testfall durch das Analysetool nicht gefunden werden, sollen gefiltert werden und dem Benutzer des SSH-Fuzzers zur genaueren Untersuchung vorgelegt werden.

7.2.2.4 Mitschnitt von gefuzzten Paketen

Während der Ausführung der Testfälle sind zusätzlich manuelle Wiresharkmitschnitte der versendeten Nachrichten erstellt worden, damit eine genauere Analyse durchgeführt werden kann und darauf basierend auch eine bessere Bewertung der Testfälle statt findet.

Die manuelle Bedienung zum Starten und Stoppen eines Wiresharkmitschnittes ist sehr umständlich und sollte deshalb automatisiert werden. Dies sollte als eine Option dem Benutzer angeboten werden, da das Starten und Aufrufen eines Programms, das den Netzwerkverkehr mitschneiden kann, die Leistung des SSH-Fuzzers verringern würde.

Jede Datei, die mitgeschnitte Pakete enthält, sollte in dem Ordner abgelegt werden, an welchem auch die Textdatei eines jeden Testfalles abgelegt wird.

A Anhang

A.1 Ergebnisse der Testfälle

A.1.1 Testfall 1

A.1.1.1 Testfall 1: Wiresharkmitschnitte

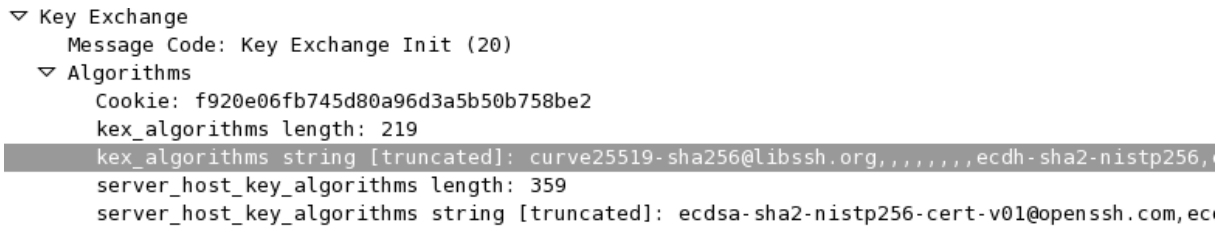

Abb. A.1: Testfall1: Inhalt der gefuzzten SSH-Nachricht

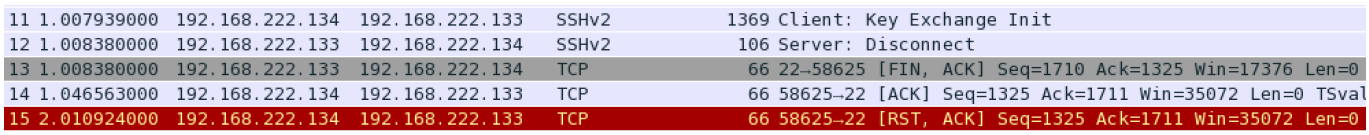

Abb. A.2: Testfall1: Verbindungsabbau der SSH-Verbindung

A.1.1.2 Testfall 1: Ressourcenüberwachung

List. A.1: Testfall1: Ressourcenüberwachung

```
1 Auszug  des Ressourcenverbrauchs des SSH-Daemon:
2
3 SSH-Daemon Prozessorauslastung: 0.05%
4 SSH-Daemon Speicherauslastung:  2348 KByte (Grenzwert nicht ueberschritten)
```

A.1.1.3 Testfall 1: Authlog-Auswertung

List. A.2: Testfall1: Ausschnitt der SSH-Loggingdaten

```
1 Auszug der Loggingdatei des SSH-Daemons:
2
3 May 28 14:55:55 SSH-Server sshd[32305]: Connection from 192.168.222.134 port 58625 on
      192.168.222.133 port 22
4 May 28 14:55:56 SSH-Server sshd[32305]: padding error: need 1295 block 8 mod 7 [preauth]
5 May 28 14:55:56 SSH-Server sshd[32305]: Disconnecting: Packet corrupt [preauth]
```

A.1.1.4 Testfall 1: Verbindungsüberprüfung

List. A.3: Testfall1: Ausschnitt der Verbindungsüberprüfung

```
1 Verbindungsueberpruefung Ergebnisse:
2
3 Ergebnis Portcheck:                    pass
4 Ergebnis SSH-Verbindungsueberpruefung:  pass
```

A.1.2 Testfall 2

A.1.2.1 Testfall 2: Wiresharkmitschnitte

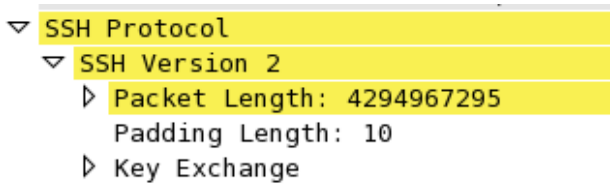

Abb. A.3: Testfall2: Inhalt der gefuzzten SSH-Nachricht

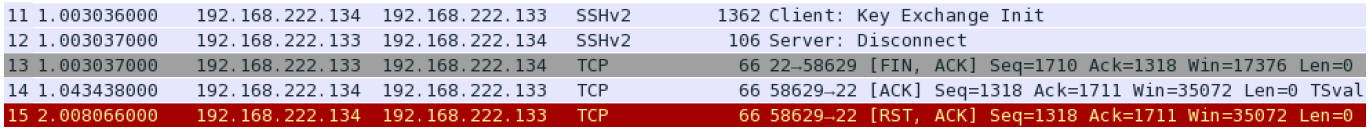

Abb. A.4: Testfall2: Verbindungsabbau der SSH-Verbindung

A.1.2.2 Testfall 2: Ressourcenüberwachung

List. A.4: Testfall2: Ressourcenüberwachung

```
1 Auszug  des Ressourcenverbrauchs des SSH-Daemon:
2
3 SSH-Daemon Prozessorauslastung: 0.00%
4 SSH-Daemon Speicherauslastung:  2360 KByte (Grenzwert nicht ueberschritten)
```

A.1.2.3 Testfall 2: Authlog-Auswertung

List. A.5: Testfall2: Ausschnitt der SSH-Loggingdaten

```
1 Auszug der Loggingdatei des SSH-Daemons:
2
3 May 28 14:56:54 SSH-Server sshd[16780]: Connection from 192.168.222.134 port 58629 on
      192.168.222.133 port 22
4 May 28 14:56:55 SSH-Server sshd[16780]: Bad packet length 4294967295. [preauth]
5 May 28 14:56:55 SSH-Server sshd[16780]: Disconnecting: Packet corrupt [preauth]
```

A.1.2.4 Testfall 2: Verbindungsüberprüfung

List. A.6: Testfall2: Ausschnitt der Verbindungsüberprüfung

```
1 Verbindungsueberpruefung Ergebnisse:
2
3 Ergebnis Portcheck:                    pass
4 Ergebnis SSH-Verbindungsueberpruefung:  pass
```

A.1.3 Testfall 3

A.1.3.1 Testfall 3: Wiresharkmitschnitte

```
                0d 0a 0d 0a 0d 0a  0a 14 00 c4 6e 11 e5 32
b9 87 f8 b3 0c f0 d7 cd  4d c4 00 00 00 d4 63 75
72 76 65 32 35 35 31 39  2d 73 68 61 32 35 36 40
```

Abb. A.5: Testfall3: Inhalt der gefuzzten SSH-Nachricht

11	1.000938000	192.168.222.134	192.168.222.133	TCP	1364 [TCP segment of a reassembled PDU]
12	1.001562000	192.168.222.133	192.168.222.134	SSHv2	106 Server: Disconnect
13	1.001563000	192.168.222.133	192.168.222.134	TCP	66 22→58637 [FIN, ACK] Seq=1710 Ack=1320 Win=17376 Len=0
14	1.041880000	192.168.222.134	192.168.222.133	TCP	66 58637→22 [ACK] Seq=1320 Ack=1711 Win=35072 Len=0 TSval
15	2.004073000	192.168.222.134	192.168.222.133	TCP	66 58637→22 [RST, ACK] Seq=1320 Ack=1711 Win=35072 Len=0

Abb. A.6: Testfall3: Verbindungsabbau der SSH-Verbindung

A.1.3.2 Testfall 3: Ressourcenüberwachung

List. A.7: Testfall3: Ressourcenüberwachung

```
1 Auszug  des Ressourcenverbrauchs des SSH-Daemon:
2
3 SSH-Daemon Prozessorauslastung: 0.00%
4 SSH-Daemon Speicherauslastung:  2368 KByte (Grenzwert nicht ueberschritten)
```

A.1.3.3 Testfall 3: Authlog-Auswertung

List. A.8: Testfall3: Ausschnitt der SSH-Loggingdaten

```
1 Auszug der Loggingdatei des SSH-Daemons:
2
3 May 28 14:57:30 SSH-Server sshd[1570]: Connection from 192.168.222.134 port 58637 on
      192.168.222.133 port 22
4 May 28 14:57:31 SSH-Server sshd[1570]: Bad packet length 218762506. [preauth]
5 May 28 14:57:31 SSH-Server sshd[1570]: Disconnecting: Packet corrupt [preauth]
```

A.1.3.4 Testfall 3: Verbindungsüberprüfung

List. A.9: Testfall3: Ausschnitt der Verbindungsüberprüfung

```
1 Verbindungsueberpruefung Ergebnisse:
2
3 Ergebnis Portcheck:                    pass
4 Ergebnis SSH-Verbindungsueberpruefung:  pass
```

A.1.4 Testfall 4

A.1.4.1 Testfall 4: Wiresharkmitschnitte

Abb. A.7: Testfall4: Inhalt der gefuzzten SSH-Nachricht

12	1.201258000	192.168.222.134	192.168.222.133	SSHv2	114	Client: Diffie-Hellman Key Exchange Init
13	1.201258000	192.168.222.133	192.168.222.134	TCP	66	22→58641 [ACK] Seq=1670 Ack=1366 Win=17328 Len=0
14	1.203246000	192.168.222.133	192.168.222.134	TCP	66	22→58641 [FIN, ACK] Seq=1670 Ack=1366 Win=17376
15	1.241438000	192.168.222.134	192.168.222.133	TCP	66	58641→22 [ACK] Seq=1366 Ack=1671 Win=35072 Len=0
16	2.008274000	192.168.222.134	192.168.222.133	TCP	66	58641→22 [FIN, ACK] Seq=1366 Ack=1671 Win=35072
17	2.008276000	192.168.222.133	192.168.222.134	TCP	66	22→58641 [ACK] Seq=1671 Ack=1367 Win=17376 Len=0

Abb. A.8: Testfall4: Verbindungsabbau der SSH-Verbindung

A.1.4.2 Testfall 4: Ressourcenüberwachung

List. A.10: Testfall4: Ressourcenüberwachung

```
1 Auszug  des Ressourcenverbrauchs des SSH-Daemon:
2
3 SSH-Daemon Prozessorauslastung: 0.00%
4 SSH-Daemon Speicherauslastung:  1948 KByte (Grenzwert nicht ueberschritten)
```

A.1.4.3 Testfall 4: Authlog-Auswertung

List. A.11: Testfall4: Ausschnitt der SSH-Loggingdaten

```
1 Auszug der Loggingdatei des SSH-Daemons:
2
3 May 28 15:13:09 SSH-Server sshd[4386]: Connection from 192.168.222.134 port 58697 on
      192.168.222.133 port 22
4 May 28 15:13:10 SSH-Server sshd[4386]: fatal: Incorrect size for server Curve25519
      pubkey: 0 [preauth]
```

A.1.4.4 Testfall 4: Verbindungsüberprüfung

List. A.12: Testfall4: Ausschnitt der Verbindungsüberprüfung

```
1 Verbindungsueberpruefung Ergebnisse:
2
3 Ergebnis Portcheck:                    pass
4 Ergebnis SSH-Verbindungsueberpruefung:  pass
```

A.1.5 Testfall 5

A.1.5.1 Testfall 5: Wiresharkmitschnitte

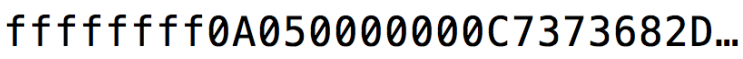

Abb. A.9: Testfall5: Entschlüsselte, gefuzzte SSH-Nachricht

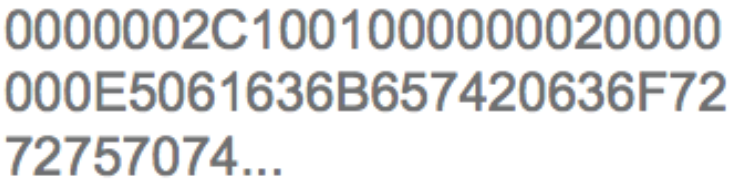

Abb. A.10: Testfall5: Entschlüsselte SSH-Disconnect Nachricht

A.1.5.2 Testfall 5: Ressourcenüberwachung

List. A.13: Testfall5: Ressourcenüberwachung

```
1 Auszug  des Ressourcenverbrauchs des SSH-Daemon:
2
3 SSH-Daemon Prozessorauslastung: 0.00%
4 SSH-Daemon Speicherauslastung:  1952 KByte (Grenzwert nicht ueberschritten)
```

A.1.5.3 Testfall 5: Authlog-Auswertung

List. A.14: Testfall5: Ausschnitt der SSH-Loggingdaten

```
1 Auszug der Loggingdatei des SSH-Daemons:
2
3 May 28 15:13:30 SSH-Server sshd[18606]: Connection from 192.168.222.134 port 58701 on
      192.168.222.133 port 22
4 May 28 15:13:31 SSH-Server sshd[18606]: Bad packet length 458859818. [preauth]
5 May 28 15:13:31 SSH-Server sshd[18606]: Disconnecting: Packet corrupt [preauth]
```

A.1.5.4 Testfall 5: Verbindungsüberprüfung

List. A.15: Testfall5: Ausschnitt der Verbindungsüberprüfung

```
Verbindungsueberpruefung Ergebnisse:

Ergebnis Portcheck:                     pass
Ergebnis SSH-Verbindungsueberpruefung:  pass
```

A.1.6 Testfall 6

A.1.6.1 Testfall 6: Wiresharkmitschnitte

```
0000007C39320000000D737368706173737764617574
80000000E7373682D636F6E6E656374696F6E00000008
70617373776F7264000000000D0d0a0d0a61737377…
```

Abb. A.11: Testfall6: Entschlüsselte, gefuzzte SSH-Nachricht

```
0000002C100100000020000
000E5061636B657420636F72
72757074...
```

Abb. A.12: Testfall6: Entschlüsselte SSH-Disconnect Nachricht

A.1.6.2 Testfall 6: Ressourcenüberwachung

List. A.16: Testfall6: Ressourcenüberwachung

```
Auszug  des Ressourcenverbrauchs des SSH-Daemon:

SSH-Daemon Prozessorauslastung: 0.00%
SSH-Daemon Speicherauslastung:  2024 KByte (Grenzwert nicht ueberschritten)
```

A.1.6.3 Testfall 6: Authlog-Auswertung

List. A.17: Testfall6: Ausschnitt der SSH-Loggingdaten

```
Auszug der Loggingdatei des SSH-Daemons:

May 28 15:07:11 SSH-Server sshd[3007]: Connection from 192.168.222.134 port 58657 on
    192.168.222.133 port 22
```

```
May 28 15:07:13 SSH-Server sshd[3007]: Failed publickey for sshpasswdauth from
    192.168.222.134 port 58657 ssh2: RSA fc:d5:77:95:ed:a8:09:b3:37:0f:b0:1e:27:30:6d:97
May 28 15:07:13 SSH-Server sshd[3007]: Bad packet length 1246211668. [preauth]
May 28 15:07:13 SSH-Server sshd[3007]: Disconnecting: Packet corrupt [preauth]
```

A.1.6.4 Testfall 6: Verbindungsüberprüfung

List. A.18: Testfall6: Ausschnitt der Verbindungsüberprüfung

```
Verbindungsueberpruefung Ergebnisse:

Ergebnis Portcheck:                     pass
Ergebnis SSH-Verbindungsueberpruefung:  pass
```

A.1.7 Testfall 7

A.1.7.1 Testfall 7: Wiresharkmitschnitte

```
0000027C10320000000D7373687075626B6579617574
680000000E7373682D636F6E6E656374696F6E000000
097075626C69636B6579010000000773736 82D727361
000001170000000773736 82D72736100000003ffffff
0000010…
```

Abb. A.13: Testfall7: Entschlüsselte, gefuzzte SSH-Nachricht

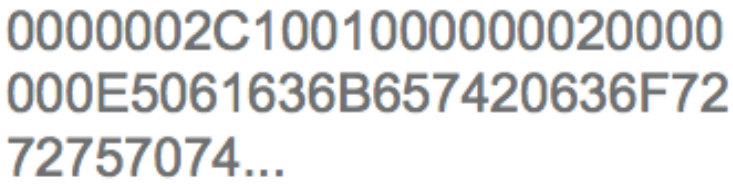

Abb. A.14: Testfall7: Entschlüsselte SSH-Disconnect Nachricht

A.1.7.2 Testfall 7: Ressourcenüberwachung

List. A.19: Testfall7: Ressourcenüberwachung

```
Auszug  des Ressourcenverbrauchs des SSH-Daemon:

SSH-Daemon Prozessorauslastung: 0.00%
SSH-Daemon Speicherauslastung:  2044 KByte (Grenzwert nicht ueberschritten)
```

A.1.7.3 Testfall 7: Authlog-Auswertung

List. A.20: Testfall7: Ausschnitt der SSH-Loggingdaten

```
1 Auszug der Loggingdatei des SSH-Daemons:
2
3 May 28 15:07:42 SSH-Server sshd[2012]: Connection from 192.168.222.134 port 58667 on
      192.168.222.133 port 22
4 May 28 15:07:44 SSH-Server sshd[2012]: Postponed publickey for sshpubkeyauth from
      192.168.222.134 port 58667 ssh2 [preauth]
5 May 28 15:07:44 SSH-Server sshd[2012]: Bad packet length 647672397. [preauth]
6 May 28 15:07:44 SSH-Server sshd[2012]: Disconnecting: Packet corrupt [preauth]
```

A.1.7.4 Testfall 7: Verbindungsüberprüfung

List. A.21: Testfall7: Ausschnitt der Verbindungsüberprüfung

```
1 Verbindungsueberpruefung Ergebnisse:
2
3 Ergebnis Portcheck:                    pass
4 Ergebnis SSH-Verbindungsueberpruefung:  pass
```

A.1.8 Testfall 8

A.1.8.1 Testfall 8: Wiresharkmitschnitte

```
0000027C10320000000D7373687075626B6579617574
680000000E7373682D636F6E6E656374696F6E000000
097075626C69636B6579010000000773736 82D727361
000001170000000773736 82D72736100000003010001
ffffff0100D0…
```

Abb. A.15: Testfall8: Entschlüsselte gefuzzte SSH-Nachricht

```
0000002C100100000020000
000E5061636B657420636F72
72757074...
```

Abb. A.16: Testfall6: Entschlüsselte SSH-Disconnect Nachricht

A.1.8.2 Testfall 8: Ressourcenüberwachung

List. A.22: Testfall8: Ressourcenüberwachung

```
1 Auszug  des Ressourcenverbrauchs des SSH-Daemon:
2
3 SSH-Daemon Prozessorauslastung: 0.00%
4 SSH-Daemon Speicherauslastung:  1972 KByte (Grenzwert nicht ueberschritten)
```

A.1.8.3 Testfall 8: Authlog-Auswertung

List. A.23: Testfall8: Ausschnitt der SSH-Loggingdaten

```
1 Auszug der Loggingdatei des SSH-Daemons:
2
3 May 28 15:08:08 SSH-Server sshd[11700]: Connection from 192.168.222.134 port 58676 on
      192.168.222.133 port 22
4 May 28 15:08:10 SSH-Server sshd[11700]: Postponed publickey for sshpubkeyauth from
      192.168.222.134 port 58676 ssh2 [preauth]
5 May 28 15:08:10 SSH-Server sshd[11700]: Bad packet length 1592995729. [preauth]
6 May 28 15:08:10 SSH-Server sshd[11700]: Disconnecting: Packet corrupt [preauth]
```

A.1.8.4 Testfall 8: Verbindungsüberprüfung

List. A.24: Testfall8: Ausschnitt der Verbindungsüberprüfung

```
1 Verbindungsueberpruefung Ergebnisse:
2
3 Ergebnis Portcheck:                    pass
4 Ergebnis SSH-Verbindungsueberpruefung:  pass
```

Literaturverzeichnis

[1] SSH, About SSH, http://www.ssh.com/about, 05.06.2015

[2] University Madison Wisconsin, Fuzz Testing of Application Reliability, http://pages.cs.wisc.edu/ bart/fuzz/, 04.05.2015

[3] University of Oulu, PROTOS - Security Testing of Protocol Implementations, https://www.ee.oulu.fi/research/ouspg/Protos, 04.05.2015

[4] Immunity, Free Software - SPIKE, http://www.immunitysec.com/resources/, 04.05.2015

[5] CL-FUZZ - Fuzz Testing in Common Lisp, http://ndantam.github.io/cl-fuzz/, 04.05.2015

[6] iFuzz, http://fuzzing.org/wp-content/ifuzz.tar, 04.05.2015

[7] Immunity, Free Software - Sharefuzz, http://www.immunitysec.com/resources/, 04.05.2015

[8] securiteam, FileFuzz - Windows Based File Format Fuzzing Tool, http://www.securiteam.com/tools/5PP051FGUE.html, 04.05.2015

[9] securiteam, SPIKEfile - Linux Based File Format Fuzzing Tool, http://www.securiteam.com/tools/5OP041FGUC.html, 04.05.2015

[10] securiteam, NotSPIKEfile - Linux Based File Format Fuzzing Tool, http://www.securiteam.com/tools/5NP031FGUI.html, 04.05.2015

[11] sourceforge, Protofuzz, http://sourceforge.net/projects/protofuzz/, 04.05.2015

[12] digitaldwarf, dhcpfuzz, http://www.digitaldwarf.be/products/dhcpfuzz.pl, 04.05.2015

[13] heise online, FTPStress Fuzzer, http://www.heise.de/security/artikel/FTPStress-Fuzzer-271468.html, 04.05.2015

[14] OWASP Project, OWASP WebScarab Project, https://www.owasp.org/index.php/Category:OWASP_WebScarab_Project, 04.05.2015

[15] Codenomicom, HTTP Server Suite, http://www.codenomicon.com/products /defensics/datasheets/http-server.html, 04.05.2015

[16] M. Sutton, Fuzzing: Brute Force Vulnerability Discovery, Addison Wesley, 2006

[17] T. Ylonen, The Secure Shell (SSH) Transport Layer Protocol, Cisco Systems Inc., 2006, https://tools.ietf.org/html/rfc4253

[18] T. Ylonen, RFC 4251 , IETF , www.ietf.org/rfc/rfc4251 ,2015

[19] T. Ylonen, RFC 4253 , IETF , www.ietf.org/rfc/rfc4253 ,2015

[20] T. Ylonen, RFC 4252 , IETF , www.ietf.org/rfc/rfc4252 ,2015

[21] T. Ylonen , RFC 4254 , IETF , www.ietf.org/rfc/rfc4254 ,2015

[22] S. Lehtinen, RFC 4250 , IETF , www.ietf.org/rfc/rfc4250 ,2015

[23] Julien Tinnes, SSH tools, https://www.cr0.org/progs/sshfun/, 18.06.2015

[24] Fuzzing States , M. Sutton, Fuzzing: Brute Force Vulnerability Discovery, Addison Wesley, 2006

[25] Microsoft Security and Research Blog , Using code coverage to improve fuzzing results, Lars Opstad http://blogs.technet.com/b/srd/archive/2010/02/24/using-code-coverage-to-improve-fuzzing-results.aspx, 21.05.2015