myliterature.bib

University of Applied Sciences
HOCHSCHULE EMDEN·LEER

Fachbereich Technik
Abteilung Elektrotechnik und Informatik

# Network Fuzzing

## Seminararbeit

Christopher Hübner
7022754

Oldenburg, November 14, 2025

Betreut von
Patrick Felke
Fredderik Gosewehr

# Rechtliche Erklärung

## Erklärung

[ ja|nein ] Die vorliegende Arbeit enthält vertrauliche / kommerziell nutzbare Informationen, deren Rechte außerhalb der Hochschule Emden/Leer liegen. Sie darf nur den am Prüfungsverfahren beteiligten Personen zugänglich gemacht werden, die hiermit auf ihre Pflicht zur Vertraulichkeit hingewiesen werden (Sperrvermerk).

[ ja|nein ] Soweit meine Rechte berührt sind, erkläre ich mich einverstanden, dass die vorliegende Arbeit Angehörigen der Hochschule Emden/Leer für Studium / Lehre / Forschung uneingeschränkt zugänglich gemacht werden kann.

Nicht Zutreffendes bitte streichen.

## Eidesstattliche Versicherung

Ich, der/die Unterzeichnende, erkläre hiermit an Eides statt, dass ich die vorliegende Arbeit selbständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Alle Quellenangaben und Zitate sind richtig und vollständig wiedergegeben und in den jeweiligen Kapiteln und im Literaturverzeichnis wiedergegeben. Die vorliegende Arbeit wurde nicht in dieser oder einer ähnlichen Form ganz oder in Teilen zur Erlangung eines akademischen Abschlussgrades oder einer anderen Prüfungsleistung eingereicht.

Mir ist bekannt, dass falsche Angaben im Zusammenhang mit dieser Erklärung strafrechtlich verfolgt werden können.

Ort, Datum, Unterschrift

# Vorwort

Diese Seminararbeit entstand im Rahmen des Moduls *„Spezielle Verfahren der IT-Sicherheit“*. Ziel des Moduls ist es, aktuelle Themen und Forschungsschwerpunkte der IT-Sicherheit zu behandeln und eigenständige Aufgabenstellungen zu erarbeiten, die von den Studierenden selbstständig bearbeitet und präsentiert werden.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Einleitung

Diese Arbeit führt systematisch in die grundlegenden Konzepte und Terminologie des Fuzzings ein. Zunächst wird eine präzise Definition des Begriffs „Fuzzing“ gegeben und dessen Zielsetzung erläutert: die automatisierte Erzeugung und Ausführung zahlreicher, bewusst fehlerhafter oder ungewöhnlicher Eingaben mit dem Zweck, Robustheits-, Verfügbarkeits- und Sicherheitslücken in Software- und Systemkomponenten aufzudecken. Darauf aufbauend wird eine strukturierte Taxonomie von Fuzzing-Ansätzen vorgestellt — unter anderem die Unterscheidung in Black-Box, Gray-Box und White-Box Verfahren sowie die Differenzierung nach generativen und mutativen Techniken und nach Coverage-Guided vs. nicht-coverage-basierten, sowie nach Feedback und nicht Feedback orientierten Methoden. Weiterhin werden die maßgeblichen Klassifikationskriterien beschrieben, die Vergleiche und Evaluierungen von Fuzzern ermöglichen (u. a. Testfallgenerierung, Orakel-Strategien, Instrumentierungsgrad, Automatisierungsgrad und Skalierbarkeit). Im Anschluss werden typische Implementierungen dieser Algorithmen, wie AFL, vorgestellt. Abschließend werden die Grundlagen von HTTPS und TLS wiederholt, um einen methodisch sauberen Übergang zum Hauptthema zu schaffen.

Im Kern der Arbeit werden um die Einordnung der vorgestellten Konzepte und eine Aktualisierung und Aufarbeitung von aktuellen Forschungsdaten und Szenarien in der Praxis gehen.

# Chapter 2

# Grundlagen des Fuzzings

In der vorliegenden Arbeit stützen wir uns auf die nachfolgende Definition des Fuzzings unter Bezugnahme auf die zitierten Quellen.

**Definition 2.0.1** (nach Chen et al., 2018)**.** Fuzzing ist eine effektive und weit verbreitete Methode zur Identifikation von Sicherheitslücken und Schwachstellen in Software. Dabei werden dem Zielprogramm gezielt unregelmäßige oder zufällige Testdaten zugeführt, um eine Ausführungssituation zu erzeugen, die potenziell eine verwundbare Programmstelle offenlegt chen2018systematic.

Ein Bug oder Fehler wird in diesem Zusammenhang wie folgt definiert:

**Definition 2.0.2** (nach Chen et al., 2018)**.** Ein Software-Bug ist ein Fehler, Mangel oder Defekt in einem Computerprogramm oder -system, der dazu führt, dass das Programm ein inkorrektes oder unerwartetes Ergebnis liefert oder sich auf eine Weise verhält, die nicht intendiert ist (Wikipedia, 2017a).

Bugs stellen demnach grundsätzlich Fehler dar, die ein vom Programmierer nicht intendiertes Verhalten hervorrufen. Dabei ist zu beachten, dass ein Algorithmus trotz scheinbar normaler Funktionalität durch gezielt gewählte Eingaben unerwartetes Verhalten erzeugen kann. Die Identifikation und Vermeidung solcher Fehler ist die zentrale Aufgabe des Fuzzings.

## 2.1 Blackbox, Whitebox und Greybox

Beim Blackbox-Fuzzing wird die interne Logik, oder Architektur der Software nicht berücksichtigt. Es wird ledigtlich das Ergebnis der Eingabe betrachtet. Whitebox hingegen berücksichtigt die interne Logik des Programms. Ziel kann hier beispielsweise eine maximale Pfadabdeckung beim Fuzzing sein. Es geht nicht nur um das Ergebnis, sondern um eine detaillierte Einsicht

in die interne Logik des Verhaltens. Greybox liegt zwischen diesen beiden Extremen. Es werden einige Informationen über das Softwaresystem herangezogen, aber nicht im Detail.

Whitebox kann somit als ein Teil des Softwareentwicklungs, bzw. Testprozesses verstanden werden. Während Black und Greybox vor allem für Pentester relevant ist, die nicht in den Entwicklungsprozess mit eingebunden sind.

## 2.2 Klassifikation nach Eingabegenerierung

Die Eingabestrategien beim Fuzzing lassen sich grundsätzlich in mutation-basierte und generation-basiertes, sowie Fuzzer-In-The-Middle Ansätze unterteilen.

Bei einer mutation-basierten Eingabestrategie werden bestehende Input-Seeds zufallsbasiert transformiert. Dies kann frühzeitig zur Erzeugung invalider Eingaben führen, die vom Zielprogramm verworfen werden. Gleichzeitig führt dies zu einer hohen Code-Coverage.[**?**]

Im Gegensatz dazu generiert eine generation-basierte Eingabestrategie neue Testdaten auf der Grundlage einer formalen Spezifikation oder eines Modells. Beispielsweise können bestimmte Grammatiken oder formale Regeln zur Erzeugung valider Eingaben herangezogen werden. Dieser Ansatz führt typischerweise zu einer geringeren Anzahl an Testfällen im Vergleich zur mutation-basierten Strategie, kann jedoch eine höhere zeitliche Komplexität aufweisen.[**?**]

Im Paper [**?**] wird Fuzzing-in-the-Middle (FitM) als ein Fuzzing-Ansatz, bei dem sich der Fuzzer zwischen zwei legitimen Kommunikationspartner setzt – typischerweise zwischen Client und Server – und laufende Netzwerkkommunikation in Echtzeit beobachtet, manipuliert und fuzzed.

## 2.3 Mit und ohne Feedbackmechanismus

Die Eingabestrategie kann die Ausgabe des Programms heranziehen, um neue Eingabedaten in der nächsten Schleife zu erstellen. Diese Techniken zielen in erster Linie auf eine höhere Pfadabdeckung und sind somit vor allem für das Whitebox Fuzzing relevant. Beim Feedback kann es sich um die Ausgabe des Programmes, oder um andere Laufzeitinformationen, beispielsweise Informationen von einem Debugger, handeln.

## 2.4 Anatomie der ersten Fuzzing Systeme

**Definition 2.4.1. Dynamic Taint Analysis** ist eine Technik, bei der Laufzeitdaten markiert („getaintet“) werden, um ihren Einfluss auf spätere Berechnungen zu verfolgen.

**Definition 2.4.2. Coverage** ist die Abdeckung in der Software gemeint, welche die EIngabeparameter des Fuzzings erreichen. Es wird in Pfad, Branch, Zielen, oder Funktionsabdeckung unterschieden.

**Definition 2.4.3. Dynamic-Symbolic-Execution** ist eine Methode um möglichen input für ein Programm zu berechnen.

Die ersten Fuzzing-Systeme waren streng iterativ aufgebaut. In jeder Iteration werden Testfälle unter Verwendung einer vordefinierten Strategie generiert. Der **Testcase Generator** generiert Testfälle unter Verwendung von Informationen die statisch und zur Laufzeit anfallen. Ein Monitor liefert Laufzeitinformationen, beispielsweise Informationen über **??Coverage**, **??Dynamic-Symbolic Execution** und Ergebnisse der **??dynamic taint analysis**. **??**Bugs im Zielprogramm werden mit Hilfe eines **Bug Filter** gefiltert, um aus ihnen Schwachstellen abzuleiten.

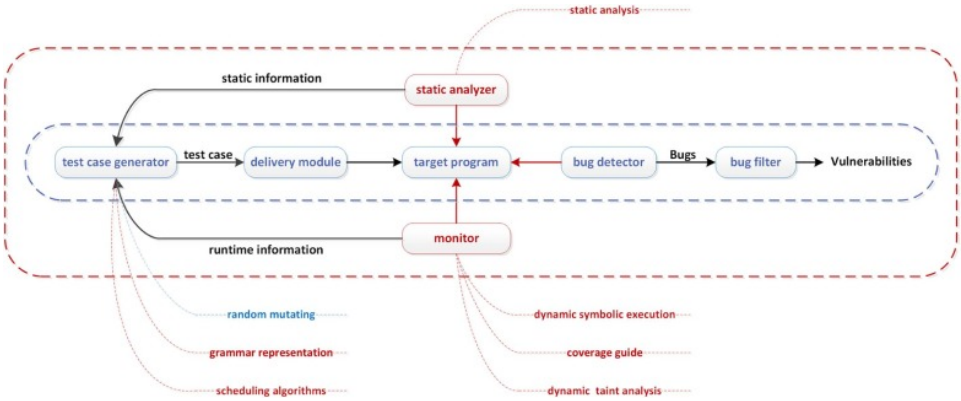


**Figure 2.1:** Darstellung des Fuzzing-Prozesses
*Quelle:[?]*

# Chapter 3

# Grundlage Netzwerke und Kryptographie

## 3.1 OSI Schichten-Modell

### 3.1.1 Anwendungsschicht

#### HTTP und Webserver

#### SSH

### 3.1.2 Transportschicht

#### TCP/UDP

#### TLS

### 3.1.3 Netzwerkschicht

#### VPN

## 3.2 Kryptografische Grundlagen

### 3.2.1 Asymmetrische Verfahren (RSA und Diffie Hellmann)

## 3.3 Zertifikate

# Chapter 4

# Netzworkfuzzing

## 4.1 Herausforderungen und Besonderheiten

Die Autoren des Papers [**?**]**Systematic Fuzzing and Testing of TLS Libraries** identifizieren vier zentrale Charakteristika von Netzwerkprotokollen, die Fuzzing vor besondere Hürden stellen:

- **??** Abhängigkeit von Netzwerk-Verbindungen.
- **??** Zustandsbehaftetheit.
- **??** Hoch strukturierte Eingaben.
- **??** Nicht-Uniformität.

### 4.1.1 Abhängigkeit von Netzwerk-Verbindungen

Anders als beim Datei-Input Fuzzing, müssen hier Netzwerkpackete gefuzzt werden, die je nach Protokoll, in einer bestimmten zeitlichen, sequentiellen Abfolge bearbeitet werden müssen. Der Fuzzer muss also ein vollwertiges PAcket für das jeweilige Netzwerkprotokoll über eine Schnittstelle übertragen. Damit muss er nicht nur das Protokoll, sondern auch die Schnittstelle verstehen. Wenn beispielsweise ein HTTPs Server, oder ein VPN-Gateway gefuzzt werden soll, kann dies zu einer erhöhten Auslastung der Infrastruktur der Firma, oder ggf. zum Ausfall führen. Für den Whitebox Hacker ist diese Art des Fuzzings somit mit ggf. höheren Kosten als beim klassischem Desktop-CLI Fuzzing verbunden. Der Blackhat-Hacker wird unter umständen schneller erkannt und muss seinen Ursprung gut tarnen.

Es lassen sich ggf. zeitliche und wirtschaftliche Kosten einsparen, wenn ein Testsystem eingerichtet wird, welches das Produktivsystem hinreichend emuliert. Beispielsweise kann auch einem Testrechner mittels OpenSSL ein TLS-Server mit derselben Version und Funktionalität eingerichtet werden.

### 4.1.2 Zustandsbehaftetheit

Netzwerkrptokolle sind üblicherweise Zustandsbehaftet. Dies kann dazu führen, dass derselbe Input, zu einem anderen Verhalten führt. Denn Zustandsbehaftet heißt, dass das Verhalten des Servers von seinem internen Zustand abhängig ist. Dies beeinflusst direkt das Design des Fuzzers. Der Input muss demzufolge so generiert werden, dass er Kette von Eingaben zum Protokoll darstellt.

Die Grafik zeigt einen von den Autor*innen vorgeschlagenen Aufbau eines Fuzzers für zustandsbehaftete Protokolle. $\mathbf{S_0}$, $\mathbf{S_1}$ und $\mathbf{S_2}$ repräsentieren Zustände des Protokolls. Sie werden aus den Eingaben $\mathbf{C_{01}}$, $\mathbf{C_{12}}$ und $\mathbf{P}$ sequentiell generiert. Der Zustand $\mathbf{S_2}$ ist der Endzustand des Protokolls. Ist dieser ein Fehlzustand, so wird dieser in eine Liste von Bugs übernommen.

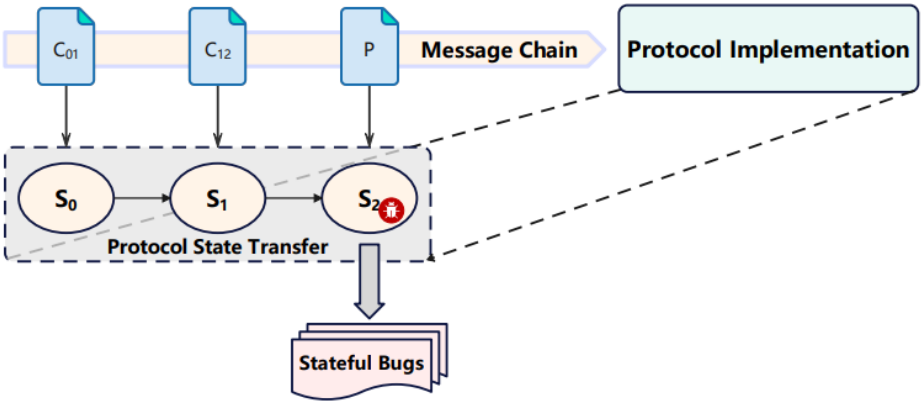


**Figure 4.1:** *Quelle:[?]*

### 4.1.3 Hoch strukturierte Eingaben

Netzwerkprotokolle sind hoch strukturiert. NAchrichten können in Big, oder Byte Felder mit strikter Grammatik partitioniert werden. Jedes dieser Partitionen hat eine klar definierte Range und Bedeutung. Diese Strukturen dürfen vom Fuzzer nicht verletzt werden. Randomisierte bitweise Eingabengenerierung, wie von AFL, kann daher zu einer vielzahl von Fehleingaben führen.

### 4.1.4 Nicht-Uniformität

Es gibt eine vielzahl von Netzwerkprotokollen, welche sich meist in den zugrundeliegenden Zustandsübergängen und Grammatiken der Nachrichten unterscheiden. Daher ist eine Vielzahl an Netzwerkfuzzern auf ein Protokoll spezialisiert.

## 4.2 Allgemeines Vorgehensmodell

Die Forscher des Artikels [**?**]**A Survey of Network Protocol Fuzzing: Model, Techniques and Directions** schlagen ein vierstufiges Vorgehensmodell zur Lösung der genannten Herausforderungen vor:

- Protokollsyntax-Erfassung und Modelling
- Testfall-Erzeugung
- Testausführung und Monitoring
- Feedback-Informationserfassung und Nutzung

### 4.2.1 Erste Stufe: Protokollsyntax-Erfassung

In der ersten Stufe wird die Syntax der Nachrichten und der Zustandsübergänge (State-Machine) des Protokolls modelliert.

Die Syntax der Nachrichten definiert hierbei eine über das Protokoll übertragene, valide Nachricht (Daten und Metadaten), ohne Kontext oder Zustand mit einzubeziehen. Es werden die Einteilung der Felder einer Nachricht, sowie deren Organisation und Abhängigkeiten betrachtet. Eine Nachricht kann Felder mit bestimmter Bedeutung und festgelegtem Datentyp haben, die in einer festgelgten Art miteinander in Beziehung stehen.

Die State-Machine ist definiert die Zuständsübergange eines Netzwerkprotokolls. Die Zustandsübergange werden repräsentiert durch einen Automaten (Ref:Theoretische Informatik).

Dies bildet die Basis für alle weiteren Stufen im Vorgehensmodell. Wichtig ist, dass diese Stuffe zum Hauptfuzzing-Prozess parralelisiert werden kann, da die Ergebnisse dieses Prozesses dynamisch in den Hauptprozess übertragen werden können.

### 4.2.2 Zweite Stufe: Testfall-Erzeugung

# Chapter 5

# Beispiele aus der Praxis

## 5.1 Spezielle Vorgehensmodelle

### 5.1.1 Zweistufiger Fuzzingansatz von TLS-Attacker

### 5.1.2 TLS-Fuzzer Libary

## 5.2 Implementierungen und Anwendung

### 5.2.1 SSH-Fuzzer

### 5.2.2 TLS-Attacker

### 5.2.3 TLS-Fuzzer

### 5.2.4 AFLnet

# Chapter 6

# Zusammenfassung und Ausblick

In diesem Kapitel wird ein Resümee zu den Ergebnissen der Projekt- oder Abschlussarbeit gegeben. Dieses besteht im Wesentlichen aus einer kurzen Zusammenfassung der Aufgabenstellung und der in den Kapiteln **??** und **??** gewonnen Erkenntnisse. Hierbei ist eine selbstkritische Darstellung angebracht und folgende Fragen sollten in einer Kurzfassung beantwortet werden:

- Mit welchem Ansatz (Kapitel **??**) wurde die Aufgabenstellung gelöst?
- Wie gut (Kapitel **??**) funktioniert die Umsetzung?
- Konnte die Aufgabenstellung (Kapitel **??**) vollständig umgesetzt werden?

Sowohl Optimierungsvorschläge als auch die Abgrenzung zu Themen, die explizit nicht behandelt wurden, dienen als Vorlage für den Ausblick auf Folgearbeiten.

**Beispiel 6.1: Webcam**

Das realisierte Kamerasystem ist in der Lage bis zu 60 Farbbilder in der Sekunde in VGA-Auflösung aufzunehmen. Die nachgelagerte Bildverarbeitungseinheit zur Schaferkennung benötigt derzeit etwa 24 $ms$ pro Bild, ist also in der Lage max. 42 Eingangsbilder pro Sekunde zu prozessieren, siehe Abschnitt **??**. Dies geht signifikant über die in der Aufgabenstellung in Abschnitt **??** geforderten 30 Bilder pro Sekunde hinaus. Aufgrund der limitierten Bandbreite der Ethernet-Schnittstelle mit 10BASE-T, ermöglicht das Kamerasystem jedoch ohne Bildkompression lediglich 22 Live-Bilder pro Sekunde an einen PC übertragen.

Sowohl die Realisierung einer Bildkompression als auch eine deutliche Reduzierung des Energieverbrauchs ist daher Gegenstand weiterführender Arbeiten.

[heading=bibintoc, title=Literatur]

# Appendix A

# Bonusmaterial

Inhalte, die nicht im direkten Fokus der Aufgabenstellung stehen, jedoch zur Ausarbeitung indirekt beigetragen haben oder zum besseren Verständnis der dargestellten Aussagen beitragen, finden im Anhang der Arbeit eine passende Position.

## A.1 Messdaten

Im begrenzten Umfang ist es hilfreich, im Anhang weiteres Datenmaterial der Arbeit hinzuzufügen, beispielsweise Tabellen, Messreihen oder kleinere Skripte, siehe Abschnitt **??**.

Bei größeren Mengen an Daten oder Quellcode ist es jedoch sinnvoller, diese in elektronischer Form als Datei im Originalformat beizulegen, beispielsweise auf CD-ROM, USB-Stick, Multimedia Card (MMC) oder Download im Repository einer Versionsverwaltung wie SVN oder Git.

## A.2 Formalien

Der Inhalt ist wichtiger als die Verpackung. Dieser Grundsatz gilt insbesondere für eine Projekt- oder Abschlussarbeit. Dennoch gilt es einen gewissen Standard bei der Gestaltung der Ausarbeitung einzuhalten. Dieses Dokument kann beim Aufbau und der Gestaltung als Vorlage dienen. Um eine ingenieurwissenschaftliche Arbeit zu verfassen stehen die Standard Office-Produkte wie beispielsweise MS Word zur Verfügung. Word wurde jedoch nicht zum Verfassen wissenschaftlicher Arbeiten mit Formeln, Abbildungen und Referenzen konzipiert und dies macht sich im Laufe der Arbeit durch offensichtliche Unzulänglichkeiten schnell bemerkbar, wie beispielsweise die unterschiedliche Darstellung eines Word-Dokuments auf verschiedenen Rechnern mit abweichenden Word-Versionen. Weiterhin bedarf es sehr viel Aufwand und Zeit, bis ein Dokument annähernd

so professionell gestaltet ist wie beispielsweise mit dem Textsetzprogramm LaTeX, das zum Verfassen wissenschaftlicher Texte[1] geschaffen wurde.

## A.3 Häufige Fehler

Grobe Rechtschreibfehler sind durch eine oftmals verwendete Autokorrektur seltener geworden - im Bereich Zeichensetzung weisen Studierende jedoch erfahrungsgemäß oftmals Wissenslücken auf. Wenn Fragen zur Grammatik und Rechtschreibung bestehen, so sollte nicht gezögert werden, den Blick in ein Fachbuch zu werfen. Es muss nicht immer der Duden sein, es existieren auch flüssig geschriebene, kompakte Nachschlagewerke auf dem Markt, beispielsweise von Balcik und Röhe [**?**]. Die hierbei investierte Zeit wird sich im Laufe des Berufslebens schnell amortisieren!

Ebenso verhält es sich mit mathematischen Definitionen, die Klarheit schaffen können. Auch hier gilt: Für Projekt- oder Abschlussarbeit sollte das Wissen in diesem Bereich aufgefrischt werden - beispielsweise mit Fachbüchern aus den ersten Semestern [**?**].

Noch ein letzter praktischer Hinweis: Das Inhaltsverzeichnis ist nicht Bestandteil des Inhaltsverzeichnisses!

[1] `https://de.wikipedia.org/wiki/LaTeX`