University of Applied Sciences
HOCHSCHULE EMDEN·LEER

Fachbereich Technik
Abteilung Elektrotechnik und Informatik

Network Fuzzing

Seminararbeit

Christopher Hübner
7022754

Oldenburg, 16. November 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.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

Einleitung

Diese Arbeit führt systematisch in die grundlegenden Konzepte und Terminologie des Netzwerkfuzzings ein. Zunächst werden die wichtigsten Begriffe und Definitionen erläutert, um ein gemeinsames Verständnis zu schaffen. Anschließend werden verschiedene Ansätze und Techniken des Netzwerkfuzzings vorgestellt und deren Vor- und Nachteile diskutiert. Dabei werden sowohl klassische, als auch intelligente Fuzzing-Methoden betrachtet. Abschließend wird ein Überblick über Tools und Frameworks gegeben, die im Bereich des Netzwerkfuzzings eingesetzt werden können. Dies ermöglicht dem Leser eine eigene Anwendung auf dem lokalen Gerät oder in einer Testumgebung.

Das Ziel dieser Arbeit ist es, dem Leser ein fundiertes Wissen über Netzwerkfuzzing zu vermitteln, um die Sicherheit von Netzwerkanwendungen und -protokollen effektiv testen und verbessern zu können. Dafür werden aktuelle Forschungsergebnisse forgestellt.

Kapitel 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 (Chen et al., 2018).

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 mutationbasierte 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.Chen et al., 2018

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.Chen et al., 2018

Im Paper Jiang et al., 2024 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 2.4.2**Coverage**, 2.4.3**Dynamic-Symbolic Execution** und Ergebnisse der 2.4.1**dynamic taint analysis**. 2.0.2Bugs im Zielprogramm werden mit Hilfe eines **Bug Filter** gefiltert, um aus ihnen Schwachstellen abzuleiten.

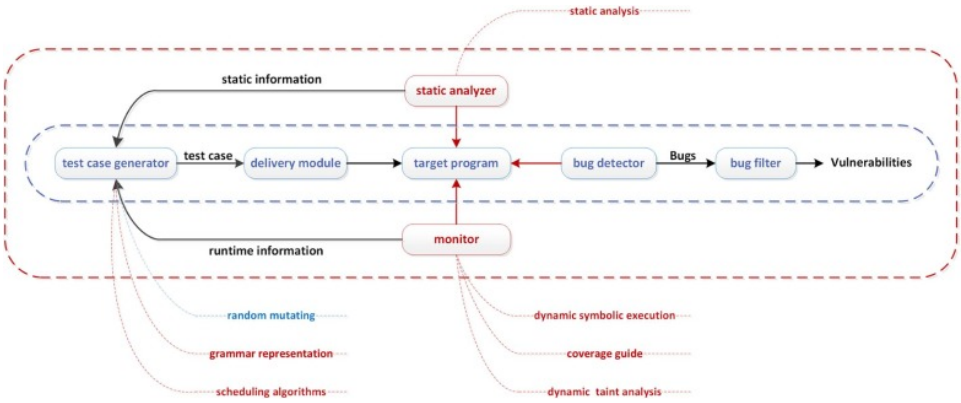

Abbildung 2.1: Darstellung des Fuzzing-Prozesses
Quelle:Chen et al., 2018

Kapitel 3

Netzworkfuzzing

Netzwerkfuzzing ist eine spezialisierte Form des Fuzzings, die sich auf die Anslyse und das Testen von Netzwerkprotokollen und -diensten konzentriert. Das Fuzzing kann ab Schicht 2 auf allen Schichten des OSI-Modells angewendet werden.

Das **OSI-Modell** (Open Systems Interconnection Model) ist ein Referenzmodell, das die Kommunikation zwischen Computersystemen in sieben hierarchische Schichten unterteilt[1]. Diese Schichten reichen von der physischen Übertragung (Schicht 1) bis zur Anwendungsschicht (Schicht 7), die direkt mit Softwareanwendungen interagiert. Jede Schicht erfüllt spezifische Aufgaben und kommuniziert nur mit den direkt angrenzenden Schichten, wodurch eine modulare und standardisierte Netzwerkkommunikation ermöglicht wird.

Im folgenden betrachten wir die Grundkonzepte und Herausforderungen, ohne auf die spezifischen Herausforderungen einzelner Protokolle oder Schichten einzugehen.

3.1 Herausforderungen im Netzwerkprotokoll-Fuzzing

Die Autoren des Papers Jiang et al., 2024**Systematic Fuzzing and Testing of TLS Libraries** identifizieren vier zentrale Charakteristika von Netzwerkprotokollen, die Fuzzing vor besondere Hürden stellen:

- 3.1.1 Abhängigkeit von Netzwerk-Verbindungen.
- 3.1.2 Zustandsbehaftetheit.
- 3.1.3 Hoch strukturierte Eingaben.
- 3.1.4 Nicht-Uniformität.

[1] Siehe https://de.wikipedia.org/wiki/OSI-Modell

3.1.1 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.

3.1.2 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.

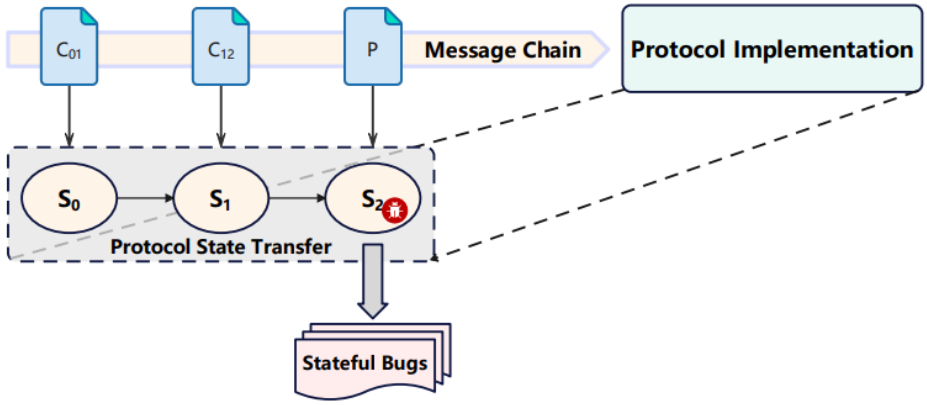

Abbildung 3.1: *Quelle:Jiang et al., 2024*

3.1.3 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.

3.1.4 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.

3.2 Allgemeines Vorgehensmodell

Die Forscher des Artikels Jiang et al., 2024**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

3.2.1 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.

3.2.2 Zweite Stufe: Testfall-Erzeugung

Diese Stufe fokussiert sich auf die Erzeugung von Testfällen. Basierend auf dem gewählten Modell und der gewählten Input-Generation-Strategie, werden in dieser Stufe die Testfälle generiert.

Die zu unterscheidenden Strategien wurden bereits in der Section 2 eingeleitet.

Generationbasierte Strategien definieren im Fall des Protokoll-Fuzzings zwei Arten von Modellen: Zustandsübergänge und Protokoll-Syntax. Frühere Fuzzer, sowie Spike Jiang et al., 2024 stellten ein Interface in Form einer Konfigurationsdatei zur Verfügung, um diese Modelle zu spezifizieren. Diese Dateien enthalten eine Art Datenstruktur der Protokoll-Syntax und der Übergänge und bilden damit die erste Epoche der hier vorgestellten Strategie.

Dies setzt natürlich ein umfangreiches Wissen über das zu testende Protokoll voraus. Kleine Fehler des Modells führen zu einer gravierend kleineren Qualität des Testprozesses. Darüber hinaus fehlt diesen Modellen die Fähigkeit den Zustandsraum autonom, oder heuristisch zu erforschen, um eine optimale Testabdeckung zu erzielen. Stattdessen führen sie einen festen Zustandsautomaten aus, der auf kleine Veränderungen der Implementierung des Protokolls schlecht reagiert.Jiang et al., 2024

Eine Strategie aktueller Fuzzer, wie Peach oder PAVFuzz, gibt bestimmten Feldern eines Paketes eine unterschiedlich hohe, oder niedrige, Gewichtung in der Wichtigkeit. Sie kalkulieren dynamische mutationen auf diesen

Gewichtungen für die Daten des Elements und speichern sie in relationalen Tabellen. Dadurch werden sogenannte **state sensitive Mutations** auf diesen Daten durchgeführt, um damit mit höherer Warscheinlichkeit Verwundbarkeiten zu entdecken.

Mutation-Basierte Strategien benötigen keine Spezifikation einer Syntax. Sie operieren auf Bit-Ebende mit Bitflip, Arithmetic, Havoc oder Splice Operationen, auf vordefinierten Seeds, um Testfälle zu generieren.

Die Autoren des Artikels Jiang et al., 2024 kritisieren, dass eine Manipulation auf Bit-Ebene ohne Syntax-Unterstützung häufig Testfälle generieren, die nicht der geforderten Grammatik des Protokolls entsprechen. Somit sind Coverage-Based Greybox Fuzzer, wie AFL und deren smarten derivate, wie AFLsmart, nicht auf Netzwerkprotokolle anzuwenden. Die Unterschiede zwischen klassischer Software und Protokollen sind zu groß. Diese Studie Li et al., 2021 hat untersucht, dass bis zu 90% der generierten Testfälle invalide waren, wenn AFL auf Netzwerkprotokolle angewendet wurde.

AFLnet wendet daher einen spezifischen Mechanismus an, um Testfälle für Netzwerkprotokolle zu generieren: Es kombiniert sequentiell alle Nachrichten vom Client während einer Interaktion in ein einziges seed. ASCII wird für die Trennung der Nachrichten im Seed verwendet. GLeichzeitig pflegt und updated es eine endliche state machine durch Extraktion der Response Codes.Jiang et al., 2024

Dadurch wird es möglich, kontinuierlich neue Zustände in die State-Machine aufzunehmen, die im Fuzzing Prozess entdeckt werden.

Dennoch hat ein mutationsbasierter Ansatz eine entscheidente Achillisverse: Die Qualität des Fuzzinghängt maßgebend vom Seed ab. Eine hochqualitatives Seed, welches auf die Protokoll-Syntax angepasst wurde, ist unerlässig für ein hoch-qualitatives Ergebnis.

Fuzzer in the Middle Fuzzer in the Middle ist eine von dem Man in the Middle abgeleitete Strategie. Der Fuzzer steht hier zwischen Server und Client. Der Fuzzer fängt die Kommunikation ab und manipuliert diese beliebig. Dadurch kann der Fuzzer sowohl den Client, als auch den Server gleichzeitig fuzzen. Der Fuzzer muss hierbei nicht manipilieren. Vielmehr greift er gezielt den Zustand zwischen Client und Server an, für den Exploits gefunden werden sollen.

Fuzzer in the Middle wird beispielsweise durch ProxyFuzz utoFuzz Gorbunov, 2010, SEC-FuzzTsankov et al., 2012, oder FITMMaier et al., 2022 implementiert.

Nachteilig ist, dass diese Strategie die Komplexität der Implementierung und Ausführung des Fuzzings maßgeblich erhöht. Sofern die Kommunikation einem kryptographischem Verfahren zugrundeliegt, muss der Fuzzer über ein fundiertes Wissen über dieses verfügen.

Anforderungen an den Testfall

Die Erzeugung von Testfällen ist in Netzwerprotokollen vor zwei besonderen Herausforderungen gestellt: Zum einen muss ein Testfall der Protokollsyntax entsprechen. Zum anderen muss der generierte Testfall möglichst gut die komplexen Zustandsübergänge des Protokolls und seiner Implementierung abdecken.

3.2.3 Dritte Stufe: Testausführung und Monitor

Die Testausführung bezeichnet den Prozess, bei dem das zu testende Programm (Program Under Test, PUT) die generierten Testfälle empfängt und verarbeitet. Der Monitor überwacht kontinuierlich das Verhalten des PUT während der Testausführung, um Anomalien, Abstürze oder andere unerwartete Verhaltensweisen zu identifizieren. Die meisten Protokoll-Fuzzer implementieren hierfür Mechanismen zur Detektion von Programmabstürzen in Kombination mit speicherbasierten Fehlererkennungsverfahren (z. B. AddressSanitizer, Valgrind).

Im Unterschied zu klassischen Fuzzern, nimmt die Testausführung eine große Rolle bei den Systemressourcen ein. Netzwerkprotokolle sind häufig zeitkritisch und erfordern eine präzise Steuerung der Netzwerkkommunikation. Daher muss die Testausführung effizient gestaltet sein, um eine hohe Testfall-Durchsatzrate zu gewährleisten.

Netzwerkprotkolle haben ebenso im Vergleich zum Lesen eines Files, oder der CLI, einen zeitlichen Overhead. Dieser ist darauf zurückzuführen, dass die Protokollsoftware zunächst Daten über das Netzwerkinterface senden und empfangen muss. Dies kann zu Latenzen führen, die den Fuzzing-Prozess verlangsamen. Im idealfall sind Fuzzer und PUT auf demselben System implementiert, um Netzwerk-Latenzen zu minimieren. Aber selbst dann muss der Fuzzer über die Loopback-Schnittstelle kommunizieren, was immer noch langsamer ist als der direkte Dateizugriff.

Darüber hinaus sind Multi-Threaded Server prinzipiell mit höheren Kosten, in Bezug auf Laufzeit, Speicher und Stromkosten, verbunden.

In den folgenden Subsektionen werden Techniken vorgestellt, um die Effizienz der Testausführung und des Monitors erheblich zu verbessern.

Snapshots für den Monitor

Snapshots sind statische Kopien des internen Zustandes eines Betriebssystems, oder Prozesses im physischem Speicher und verschiedener Devices zu einem bestimmten Zeitpunkt.Jiang et al., 2024

Im Kontext des Netzwerkprotokoll-Fuzzings können Snapshots verwendet werden, um den Zustand des PUT zu einem bestimmten Zeitpunkt während der Testausführung zu erfassen. Dies ermöglicht es dem Monitor, den

Zustand des PUT zu analysieren und potenzielle Fehlerquellen zu identifizieren und ggf. den Zustand wiederherzustellen, um den Fuzzing-Prozess fortzusetzen.

Diese Sektion könnte deutlich mehr Aufmerksamkeit erhalten. Um aber den Fokus nicht zu verlieren, wird gern auf die Originalquelle verwiesen. In dieser werden diverse weitere Quellen genannt, die einen tieferen Einblick in die Materie ermöglicht.

Emulation von Netzwerken

Einige Forschungen und Tools haben sich darauf konzentriert, kostenintensive Netzwerkinterfaces durch simulierte Netzwerke zu ersetzen. Die Forscher des Artikels Jiang et al., 2024[A Survey of Network Protocol Fuzzing: Model, Techniques and Directions] nennen hier beispielsweise **Nyx-net**, welches ein emuliertes Netzwerkinterface bereitstellt, das den Netzwerkverkehr zwischen dem Fuzzer und dem PUT simuliert. Dadurch können Netzwerkprotokolle getestet werden, ohne dass physische Netzwerkhardware erforderlich ist.

Dieses Thema ist allgemein sehr umfangreich und geht weit über den Rahmen in dieser Arbeit hinaus. Dennoch ist es wichtig zu wissen, dass es solche Ansätze gibt, um die Kosten und Komplexität des Netzwerkprotokoll-Fuzzings zu reduzieren.

IO-Synchronisation

SFuzz implementiert Synchronisationspunkte zur deterministischen Steuerung der Nachrichtenübertragung. Netzwerkprotokollsoftware operiert nach der Initialisierungsphase typischerweise in einer iterativen Event-Loop-Struktur. Jede Client-Eingabe induziert einen Verarbeitungszyklus bestehend aus: Nachrichtenempfang → Verarbeitung → Rücksprung in die Event-Loop. **SFuzz** platziert einen Synchronisationspunkt am Einstiegspunkt der Event-Loop, welcher deterministisch signalisiert, dass das PUT (Program Under Test) bereit zur Entgegennahme einer neuen Nachricht ist. Dadurch entfällt die Notwendigkeit heuristischer Verfahren (wie Timeout-basierte Mechanismen) zur Bestimmung des optimalen Sendezeitpunkts. Dies resultiert in einer signifikanten Steigerung der Fuzzing-Effizienz.

3.2.4 Stufie vier: Feedback Informationserfassung und Nutzung

3.3 Zwei-Stufenmodell von TLS-Attacker

Kapitel 4

Anwendungsbeispiele

4.1 Lokale SSH und TLS Server

4.2 SSH-Fuzzer

4.3 TLS-Attacker

4.4 TLS-Fuzzer

4.5 AFLnet

Kapitel 5

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 1) 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 5.1: 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.

Literatur

Chen, C., Cui, B., Ma, J., Wu, R., Guo, J., & Liu, W. (2018). A Systematic Review of Fuzzing Techniques. *Computers & Security*, *75*, 118–137. https://doi.org/10.1016/j.cose.2018.02.002

Gorbunov, S. (2010). Autofuzz: Automated network protocol fuzzing framework. *International Journal of Computer Science and Network Security (IJCSNS)*, *10*, 239.

Jiang, S., Zhang, Y., Li, J., Yu, H., Luo, L., & Sun, G. (2024). A Survey of Network Protocol Fuzzing: Model, Techniques and Directions. *arXiv preprint arXiv:2402.17394*. https://doi.org/10.48550/arXiv.2402.17394

Li, S., Li, J., Fu, J., Xue, M., Yu, H., & Sun, G. (2021). Protocol fuzzing with specification guided message generation. *Proceedings of the 2021 International Conference*.

Maier, D., Bittner, O., Munier, M., & Beier, J. (2022). FITM: Binary-only coverage-guided fuzzing for stateful network protocols [Part of the Network and Distributed System Security Symposium (NDSS) Workshops]. *Workshop on Binary Analysis Research (BAR)*.

Tsankov, P., Dashti, M. T., & Basin, D. (2012). Secfuzz: Fuzz-testing security protocols. *2012 7th International Workshop on Automation of Software Test (AST)*, 1–7.

Anhang 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 **balcik2010**. 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 **teschl2013**.

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

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