Effizientes Fuzzing von IoT-Geräten

Zur Erlangung des akademischen Grades eines

Doktors der Ingenieurwissenschaften

von der KIT-Fakultät für Informatik
des Karlsruher Instituts für Technologie (KIT)

genehmigte

Dissertation

von

Matthias Börsig

Tag der mündlichen Prüfung: 18. Dezember 2025

1. Referent: PD Dr.-Ing. Ingmar Baumgart
2. Referent: Prof. Dr. Andreas Zeller

Kurzzusammenfassung

Fuzzing ist eine bewährte Methode zur Identifizierung von Sicherheitslücken in Software. In dieser Dissertation wird untersucht, wie sich die Effizienz des Fuzzings für IoT-Geräte, am Beispiel des ESP32-Mikrocontrollers, steigern lässt. Während Fuzzing in klassischen Softwareumgebungen etabliert ist, fehlen speziell angepasste Verfahren für ressourcenarme IoT-Hardware. Das Ziel besteht in der Entwicklung eines konzeptionellen Frameworks, das eine umfassende und effiziente Testung trotz begrenzter IoT-Ressourcen ermöglicht. Zu diesem Zweck werden vier Ansätze zur Effizienzsteigerung sowie ein Konzept zur flexiblen Kombination dieser Ansätze vorgestellt.

Beim Binary Rewriting wird Binärcode so modifiziert, dass die Funktionalität erhalten bleibt. Für viele gängige Architekturen existieren bereits entsprechende Verfahren. Für die Xtensa-Architektur des ESP32 gab es jedoch bislang keine Lösung. In dieser Dissertation wird gezeigt, wie sich Binary Rewriting auf dem ESP32 umsetzen lässt, um Fuzzing-Instrumentierungen direkt in die Firmware zu integrieren und Laufzeitinformationen an den Fuzzer zurückzumelden.

Zudem wurde die Emulationsumgebung des ESP32 erheblich erweitert. Dadurch ist nun Fuzzing von beliebiger Firmware möglich, auch von Firmware mit zuvor nicht unterstützten Hardwarekomponenten. Im Vergleich zur realen Hardware arbeitet die Emulation deutlich effizienter. Während hardwarebasiertes Fuzzing vier bis 40 Anfragen pro Sekunde verarbeitet, sind es in der Emulation bis zu 320 Anfragen pro Sekunde.

Zur Generierung valider Eingaben wurde ein Verfahren zum automatisierten Protocol Reverse Engineering (PRE) entwickelt, das Künstliche Neuronale Netze (KNNs) verwendet. Während PRE bislang manuell erfolgen musste, können nun Protokollstrukturen automatisch abgeleitet und damit syntaktisch korrekte Netzwerkpakete erzeugt werden. In den Tests waren 67,6 % der erzeugten HTTP-Pakete und 100 % der FTP-Pakete gültig.

Für das grammatikbasierte Fuzzing wird ein Ansatz mittels Large Language Models (LLMs) vorgestellt. Die zentrale Herausforderung bestand in der effizienten Integration des LLM in den Fuzzing-Prozess. Mithilfe der entwickelten Methode lassen sich syntaktisch und semantisch korrekte XML-Dateien generieren. Dies steigert die Programmflussabdeckung um den Faktor sechs gegenüber einer Ausführung ohne LLM und erreicht eine um 50 % höhere Abdeckung als klassische grammatikbasierte Fuzzer.

Abschließend wird ein Integrationskonzept präsentiert, das eine flexible Kombination der Ansätze ermöglicht und deren Verbesserungen additiv nutzbar macht. Dadurch trägt das Framework zur Effizienzsteigerung des Fuzzings von IoT-Geräten bei. Die Dissertation leistet somit einen wichtigen Beitrag zur praxisnahen Absicherung von IoT-Geräten.

Abstract

Fuzzing is a well-established method of identifying security vulnerabilities in software. This dissertation explores ways to improve the efficiency of fuzzing for Internet of Things (IoT) devices, using the ESP32 microcontroller as a case study. While fuzzing is widely used in traditional software environments, few methods have been developed to address the limited resources available on IoT hardware.

The aim is to develop a conceptual framework that enables thorough and effective testing despite the limited resources available on IoT devices. To this end, four approaches to increasing efficiency are presented, along with a concept for combining these approaches flexibly.

In binary rewriting, the functionality of the machine code is preserved, but the code itself is modified. Although techniques already exist for many common architectures, there has been no solution available for the Xtensa architecture of the ESP32. This dissertation demonstrates how binary rewriting can be implemented on the ESP32 to integrate fuzzing instrumentation directly into the firmware, feeding runtime information back to the fuzzer.

The ESP32's emulation environment has also been significantly expanded. Consequently, fuzzing is now feasible for arbitrary firmware, including that which previously relied on unsupported hardware components. Compared to physical hardware, the emulation runs far more efficiently: while hardware-based fuzzing processes four to 40 requests per second, the emulation achieves up to 320.

To generate valid inputs, an automated Protocol Reverse Engineering (PRE) approach using Neural Networks (NN) was developed. While protocol reverse engineering had previously required manual effort, protocol structures can now be derived automatically, enabling the generation of syntactically correct network packets. During testing, 67.6 % of the generated HTTP packets and 100 % of the FTP packets were valid.

A new approach leveraging a Large Language Model (LLM) for grammar-based fuzzing has been introduced. The main challenge was integrating the LLM efficiently into the fuzzing process. The resulting method can generate XML files that are both syntactically and semantically correct. Compared to execution without an LLM, program flow coverage increases sixfold, achieving 50 % higher coverage than classical grammar-based fuzzers.

Finally, an approach is presented that integrates these concepts, enabling improvements to be applied additively. This enhances the efficiency of fuzzing for IoT devices. Consequently, this dissertation makes a significant contribution to the practical security of IoT systems.

Inhaltsverzeichnis

Abkürzungsverzeichnis

AE	Autoencoder
AFL	American Fuzzy Lop
AiFF	Ansatz für ein integriertes Fuzzing-Framework
API	Application Programming Interface
BCE	Binary Cross-Entropy
BLE	Bluetooth Low Energy
BMU	Best Matching Unit
CCE	Categorical Cross-Entropy
CE	Cross-Entropy
CISC	Complex Instruction Set Computer
CNN	Convolutional Neural Network
CoAP	Constrained Application Protocol
CSV	Comma-Separated Values
DBSCAN	Density-Based Spatial Clustering of Applications with Noise
DDoS	Distributed Denial of Service
DL	Deep Learning
DoS	Denial of Service
EBR	ESP32 Binary Rewriting
ELF	Executable and Linkable Format
EOP	End of Package
EQF	ESP32-QEMU-FUZZ
FTP	File Transfer Protocol
GAN	Generative Adversarial Network
GDB	GNU Debugger
GPU	Graphics Processing Unit
HDBSCAN	Hierarchical Density-Based Spatial Clustering of Applications with Noise
HTTP	Hypertext Transfer Protocol
HTTYL	How to Train Your Llama
IDF	IoT Development Framework
IoT	Internet of Things
IR	Intermediate Representation
ISA	Instruction Set Architecture
ISO	International Organization for Standardization
KI	Künstliche Intelligenz
KNN	Künstliches Neuronales Netz
LLM	Large Language Model
LoRA	Low-Rank Adaptation

LSTM	Long Short-Term Memory
ML	Machine Learning
MQTT	Message Queueing Telemetry Transport
MSB	Most Significant Bit
MSE	Mean Squared Error
NLL	Negative Log-Likelihood
NVS	Non-Volatile Storage
PEFT	Parameter-Efficient Fine-Tuning
PoC	Proof of Concept
PREUNN	Protocol Reverse Engineering using Neural Networks
PRE	Protocol Reverse Engineering
PUT	Program Under Test
ReLU	Rectified Linear Unit
RISC	Reduced Instruction Set Computer
RNN	Recurrent Neural Network
Seq2Seq	Sequence to Sequence
SOM	Self-Organizing Map
SOP	Start of Package
TCP	Transmission Control Protocol
TPU	Tensor Processing Unit
TTM	Time-to-Market
UDP	User Datagram Protocol
XML	Extensible Markup Language
XMPP	Extensible Messaging and Presence Protocol

1 Einleitung

Dieses Kapitel führt in die Dissertation ein und erläutert ihre wissenschaftliche Relevanz. Im Fokus stehen die Sicherheit im Internet der Dinge (engl. Internet of Things, kurz IoT) und das Fuzzing von IoT-Geräten als zentrale Forschungsfelder. Anschließend werden die Ziele und Beiträge der Dissertation vorgestellt sowie ihr Aufbau erläutert.

1.1 Motivation

In den letzten Jahren ist die Anzahl von IoT-Geräten signifikant gewachsen. Intelligente Sensoren und vernetzte Geräte finden zunehmend Anwendung in der Industrie, in Smart-Home-Systemen und in der kritischen Infrastruktur. Laut Fortune Business Insights betrug die weltweite Marktgröße für IoT im Jahr 2019 rund $251 Milliarden und wird bis 2027 voraussichtlich auf über $1,463 Milliarden anwachsen [For20]. Trotz dieser Dynamik wurden Sicherheitsaspekte in der IoT-Entwicklung lange vernachlässigt. Der IoT Threat Report 2020 zeigt, dass 57 % der untersuchten Geräte Schwachstellen mittlerer oder hoher Schwere aufwiesen [Uni20]. Dies macht IoT-Systeme zu attraktiven Zielen für Angriffe, beispielsweise können sie übernommen und in sogenannte Botnetze vereint werden, die dann ferngesteuert für Distributed Denial of Service (DDoS)-Angriffe genutzt werden, um Webseiten lahmzulegen [BSI25].

Eine wichtige Rolle in der IoT-Landschaft spielt der ESP32-Mikrocontroller, der 2016 veröffentlicht und seitdem über 100 Millionen Mal verkauft wurde [Esp18; Eli22]. Aufgrund seiner hohen Integrationsdichte, des günstigen Preises und der Unterstützung von WLAN- und Bluetooth-Kommunikation ist der ESP32 in zahlreichen kommerziellen Produkten weit verbreitet [Mil21]. Seine Popularität macht ihn zugleich zu einem lohnenden Ziel für Angreifer.

Ein zentrales Problem in der IoT-Entwicklung ist das sogenannte Time-to-Market (TTM)-Dilemma: Um möglichst schnell marktreife Produkte zu liefern, wird IT-Sicherheit oft nachrangig behandelt oder erst nachträglich berücksichtigt [Bau+19]. Dies führt regelmäßig zu gravierenden Sicherheitslücken, die Angreifern einen einfachen Zugang zu sensiblen Daten oder vollständige Kontrolle über Geräte ermöglichen [Kre16].

Eine etablierte Methode zur Identifikation von Sicherheitslücken ist neben der statischen Codeanalyse das *Fuzzing* [Lia+18; GPS17]. Dabei wird Software automatisiert mit zufälligen oder gezielt generierten Eingaben getestet, um fehlerhafte Zustände wie Abstürze, Speicherlecks oder unbehandelte Ausnahmen zu provozieren. Ein vollständiges Durchtesten aller möglichen Eingaben wäre ideal, ist in der Praxis jedoch aufgrund der enormen

Größe des Eingaberaums nicht realisierbar [AIB11]. Stattdessen werden gezielte Strategien eingesetzt, um den Suchraum effizient einzugrenzen.

Typische Ziele des Fuzzings sind Software-Komponenten, Netzwerkprotokolle, Schnittstellen oder Anwendungen, wie etwa XML-Parser. Durch die systematische Eingabe fehlerhafter, unerwarteter oder zufälliger Daten lassen sich Schwachstellen und Stabilitätsprobleme aufdecken. Mithilfe von Fuzzing lassen sich unter anderem Pufferüberläufe, Speicherlecks, unbehandelte Ausnahmen und Logikfehler identifizieren [TDM08].

Im Kontext des IoT stößt Fuzzing jedoch auf besondere Herausforderungen. Zum einen sind viele Geräte durch knappe Hardware-Ressourcen wie begrenzten Speicher, geringe Rechenleistung und fehlende Debugging-Schnittstellen eingeschränkt, was die direkte Analyse auf den Geräten erheblich erschwert [Yun+22; Tou+24]. Zum anderen führt die große Vielfalt an Mikrocontroller-Architekturen und Betriebssystemen dazu, dass etablierte Fuzzing-Tools nur eingeschränkt wiederverwendet werden können [Mue+18]. Darüber hinaus setzen IoT-Geräte häufig auf komplexe Kommunikationsschnittstellen wie WLAN, Bluetooth Low Energy (BLE) oder ZigBee, deren vielfältige Zustandsräume das systematische Testen zusätzlich erschweren [Tou+24].

Um diese Probleme gezielt anzugehen, bieten sich zwei technische Ansätze an. Binary Rewriting adressiert insbesondere die Einschränkungen durch fehlende Debugging-Schnittstellen und begrenzte Ressourcen: Durch das gezielte Injektieren von Analysecode in Firmware-Binaries kann die Codeabdeckung verfolgt und der Kontrollfluss direkt an den Fuzzer zurückgemeldet werden, ohne das ursprüngliche Verhalten zu verändern [DGR20]. Dadurch wird das Testen selbst auf ressourcenarmen Geräten ermöglicht. Emulation ergänzt diesen Ansatz, indem die Firmware in einer kontrollierten und reproduzierbaren Umgebung ausgeführt wird. Dadurch lassen sich interne Zustände, Speicherzugriffe und Kommunikationsabläufe beobachten, während sich Fuzzing-Kampagnen parallelisieren und beschleunigen lassen. Gleichzeitig reduziert die Emulation die Abhängigkeit von der konkreten Hardwarearchitektur und erleichtert so die Wiederverwendbarkeit der Tests über verschiedene Geräte hinweg [Cle+20; Wri+21; Yun+22].

Darüber hinaus eröffnen Künstliche Intelligenz (KI)-gestützte Methoden neue Perspektiven für das Fuzzing. Verfahren aus den Bereichen Machine Learning (ML) und Deep Learning (DL) werden bereits erfolgreich in der statischen Codeanalyse, Malware-Erkennung und Spam-Filterung eingesetzt [DB25; Kna+25; KY24; GAJ24]. Im Fuzzing ermöglichen diese Methoden eine gezielte Optimierung der Eingabegenerierung: Mithilfe von Mustererkennung können Protokollstrukturen approximiert und synthetische, wohldefinierte Testdaten erzeugt werden. Dadurch steigt die Wahrscheinlichkeit, dass Eingaben vom Program Under Test (PUT) akzeptiert werden, was die Testabdeckung verbessert und die Effizienz der Schwachstellensuche erheblich steigert [Zha+24a].

1.2 Zielsetzung und wissenschaftlicher Beitrag

Die zentrale Forschungsfrage dieser Dissertation lautet:

„Wie kann das Fuzzing von IoT-Geräten effizienter gestaltet werden?“

Zu diesem Zweck wird ein konzeptionelles Framework für ESP32-Anwendungen entwickelt. Dieses Framework integriert verschiedene Ansätze zur Effizienzsteigerung systematisch und macht deren Synergien nutzbar (siehe Definition von effizientem Fuzzing in Abschnitt 2.4.7). Der ESP32 dient dabei als exemplarische Plattform, da seine Architektur die typischen Einschränkungen ressourcenarmer IoT-Geräte, wie begrenzte Rechenleistung, Speicher- und Energie-Ressourcen, widerspiegelt. Als methodische Grundlage werden vier Techniken realisiert: ESP32 Code-Injektion bei unverändertem Kontrollfluss mittels Binary Rewriting, Fuzzing von ESP32-Mikrocontrollern mittels QEMU-Emulation, Protocol Reverse Engineering (PRE) mittels neuronaler Netze und effizientes grammatikbasiertes Fuzzing mittels Large Language Models (LLMs). Die praktischen Implementierungen demonstrieren den Nutzen dieser Ansätze, während das Framework ihre flexible Kombination für unterschiedliche Anwendungsszenarien aufzeigt.

Die wesentlichen Beiträge dieser Dissertation lassen sich wie folgt zusammenfassen:

- **ESP32 Code-Injektion bei unverändertem Kontrollfluss mittels Binary Rewriting:** Beim Binary Rewriting kann beliebiger Code nachträglich in ein Programm eingefügt werden, ohne dessen ursprüngliches Verhalten zu verändern. Dazu zählt auch die Instrumentierung, also Code, der während der Ausführung zusätzliche Informationen erfasst – beispielsweise über Speicherzugriffe, Funktionsaufrufe oder den Kontrollfluss. Bislang existierten Ansätze hierfür nur für andere Architekturen wie x86 und ARM. Eine zentrale Herausforderung bestand darin, diese Technik für die Xtensa-Architektur des ESP32 technisch umsetzbar zu machen. Es wurde ein Verfahren entwickelt, das den Binärcode erfolgreich modifiziert und erlaubt relevante Laufzeitinformationen direkt an den Fuzzer zu übertragen. Dadurch wird die Analyse beschleunigt und die Datenmenge reduziert. Praktische Tests zeigen, dass sämtliche Instrumentierungen korrekt ausgeführt werden und die Integrität der ursprünglichen Firmware erhalten bleibt.
- **Fuzzing von ESP32-Mikrocontrollern mittels QEMU-Emulation:** Auf Basis von QEMU wurde eine vollständige Emulationsumgebung mit einem integrierten Fuzzer für den ESP32 realisiert. Die zentrale Herausforderung bestand einerseits darin, die Emulationssoftware inklusive der fehlenden Hardwarekomponenten (z. B. WLAN-Unterstützung) durch gezielte Anpassungen so zu verändern, dass beliebige ESP32-Software lauffähig wird. Andererseits musste ein Fuzzer nahtlos in die Emulation integriert werden, um automatisierte Tests innerhalb derselben Umgebung zu ermöglichen. Das resultierende System erlaubt direkten Zugriff auf interne Systeminformationen und erleichtert die effiziente Erkennung von Speicherfehlern und Schwachstellen. Zudem ermöglicht es parallele Ausführung und schnellere Fuzzing-Durchläufe auf der leistungsstarken PC-Hardware, was die Effizienz erheblich

steigert. Für den Vergleich wurde auf Parallelisierung verzichtet, um gleiche Bedingungen zur Hardwareausführung sicherzustellen: Während hardwarebasiertes Fuzzing vier bis 40 Eingaben pro Sekunde verarbeitet, erreicht die Emulation bis zu 320 Eingaben pro Sekunde.

- **Protocol Reverse Engineering mittels neuronaler Netze:** Es wurde ein KI-basierter Ansatz für das automatisierte PRE von Netzwerkprotokollen entwickelt. Die zentrale Herausforderung bestand darin, geeignete ML- und DL-Modelle auszuwählen und zu kombinieren, um Protokollstrukturen ohne vorhandene Spezifikationen zuverlässig abzuleiten. Während das PRE bisher manuell durchgeführt werden musste, lassen sich mit diesem Ansatz nun Protokollstrukturen automatisiert ableiten und syntaktisch sowie semantisch korrekte Netzwerkpakete erzeugen. In den Tests waren 67,6 % der generierten Hypertext Transfer Protocol (HTTP)-Pakete und 100 % der File Transfer Protocol (FTP)-Pakete gültig. Dadurch erhöht sich die Testabdeckung signifikant, da im Vergleich zu zufällig generierten Daten ein größerer Anteil der Eingaben vom PUT akzeptiert wird und somit verwertbare Rückmeldungen erzeugt werden, statt sofort als ungültig verworfen zu werden.

- **Effizientes grammatikbasiertes Fuzzing mittels Large Language Models:** Für den Fuzzing-Prozess wurde ein Ansatz entwickelt, der mithilfe von LLMs syntaktisch und semantisch korrekte Dateien (z. B. Extensible Markup Language (XML)-Dateien) als Eingaben erzeugt. Die zentrale Herausforderung bestand darin, die LLMs effizient in bestehende Testabläufe zu integrieren. Im Rahmen dieser Untersuchung konnte gezeigt werden, dass diese Integration erfolgreich umgesetzt werden kann. Die Programmflussabdeckung steigt im Vergleich zu Ausführungen ohne LLM um den Faktor 6 und erreicht zudem eine um 50 % höhere Abdeckung, als klassische grammatikbasierte Fuzzer.

- **Ansatz für ein integriertes Fuzzing-Framework:** Es wird ein Ansatz vorgestellt, bei dem die vier vorgestellten Techniken flexibel kombiniert werden, um Synergien gezielt zu nutzen. Binary Rewriting ermöglicht eine effiziente Gewinnung von Laufzeitinformationen und unterstützt die Schwachstellenerkennung. Die Emulation erhöht die Anzahl der pro Sekunde verarbeiteten Eingaben und trägt ebenfalls zur Verbesserung der Schwachstellenerkennung bei. Je nach Anwendungsfall können mit dem PRE-Ansatz entweder gültige Netzwerkpakete automatisiert generiert oder mit dem LLM-Ansatz strukturierte Eingabedateien (z. B. im XML-Format) erstellt werden. Das Konzept zeigt, dass sich die Effizienzsteigerungen der einzelnen Methoden kombinieren lassen, um die Gesamteffektivität der Testgenerierung zu erhöhen.

Die Dissertation beantwortet somit die Frage, wie das Fuzzing von IoT-Geräten effizienter gestaltet werden kann. Zu diesem Zweck werden vier Ansätze vorgestellt und ein konzeptionelles Framework zu deren Kombination entwickelt. Damit schließt sie zentrale Forschungslücken im Bereich des Fuzzings von IoT-Geräten, insbesondere bei ressourcenschwachen Plattformen und heterogenen Protokollen. Gleichzeitig demonstriert sie die Machbarkeit und Effektivität der Methoden und schafft eine fundierte Grundlage für die praxisnahe Absicherung von IoT-Geräten.

1.3 Aufbau der Dissertation

Die Dissertation ist so strukturiert, dass zunächst die theoretischen Grundlagen vermittelt und anschließend die entwickelten Methoden, ihre Umsetzung und ihre Anwendung im IoT-Fuzzing systematisch vorgestellt werden.

Kapitel 2 (Grundlagen) führt in die grundlegenden Konzepte ein, die für das Verständnis der Dissertation erforderlich sind. Neben den theoretischen Grundlagen des Fuzzings werden auch die Besonderheiten der Sicherheitsanalyse von IoT-Geräten erläutert.

Kapitel 3 (ESP32 Code-Injektion bei unverändertem Kontrollfluss mittels Binary Rewriting) stellt die Methode des Binary Rewriting vor und erläutert die Umsetzung für den ESP32-Mikrocontroller. Es wird gezeigt, wie sich Firmware gezielt instrumentieren lässt, um relevante Laufzeitinformationen ohne vollständiges Speicherauslesen an den Fuzzer zu übertragen. Dies schafft die Grundlage für effizientere Testabläufe.

In Kapitel 4 (Fuzzing von ESP32-Mikrocontrollern mittels QEMU-Emulation) wird die Umsetzung praxisnaher Fuzzing-Methoden am Beispiel des ESP32-Mikrocontrollers beschrieben. Es wird erläutert, wie durch Emulation die Einschränkungen realer Hardware umgangen und parallele Fuzzing-Instanzen realisiert werden können.

Kapitel 5 (Protocol Reverse Engineering mittels neuronaler Netze) erläutert die KI-gestützte Protokollanalyse. Es wird beschrieben, wie mittels PRE unbekannte textbasierte Kommunikationsprotokolle aus dem Datenverkehr rekonstruiert und durch den Einsatz von neuronalen Netzwerken gezielt valide Netzwerkpakete als Eingaben für das Fuzzing erzeugt werden.

Kapitel 6 (Effizientes grammatikbasiertes Fuzzing mittels Large Language Models) behandelt den Einsatz von LLMs zur Generierung von Eingaben. Es wird gezeigt, wie diese Modelle variantenreiche, syntaktisch und semantisch korrekte Eingaben für komplexe Programme erzeugen.

In Kapitel 7 (Ansatz für ein integriertes Fuzzing-Framework) wird ein konzeptioneller Ansatz zur Kombination der vier vorgestellten Techniken beschrieben. Es wird erläutert, welche Synergieeffekte möglich sind und wie unterschiedliche Anwendungsfälle von einer Integration profitieren könnten.

In Kapitel 8 (Verwandte Arbeiten) werden die vorgestellten Verfahren in den aktuellen Stand der Forschung eingeordnet und bestehende Ansätze sowie deren Limitationen aufgezeigt.

Kapitel 9 (Zusammenfassung und Ausblick) fasst die wichtigsten Ergebnisse der Dissertation zusammen und gibt einen Ausblick auf zukünftige Forschungsrichtungen im Bereich IoT-Fuzzing.

2 Grundlagen

Dieses Kapitel legt die theoretischen und technischen Grundlagen, die für diese Dissertation erforderlich sind. Es werden zentrale Konzepte aus den Bereichen IoT-Geräte, Mikrocontroller-Architekturen, Binary Rewriting, Fuzzing, Netzwerkprotokolle und neuronale Netzwerke systematisch vorgestellt. Ziel ist es, ein einheitliches Verständnis zu schaffen, das die spätere Analyse und praktische Umsetzung erleichtert.

Zu Beginn widmet sich das Kapitel den Besonderheiten von IoT-Geräten, die sich durch spezifische Architekturen, Kommunikationsprotokolle und Fuzzing-Herausforderungen auszeichnen. Anschließend wird der ESP32-Mikrocontroller als exemplarische Plattform mit seiner Architektur, Firmware-Struktur und der Xtensa-Instruction Set Architecture (ISA) vorgestellt. Darauf aufbauend werden die wesentlichen Konzepte des Binary Rewritings sowie die Grundlagen des Fuzzings einschließlich grammatikbasiertem Fuzzing und der Nutzung von Emulatoren präsentiert. Abschließend werden relevante Netzwerkprotokolle wie Protocol Reverse Engineering erläutert. Zentrale neuronale Netzwerkarchitekturen und weitere grundlegende Definitionen, die die methodische und theoretische Basis dieser Dissertation bilden, werden abschließend präsentiert.

2.1 Besonderheiten von IoT-Geräten

IoT-Geräte weisen spezifische Eigenschaften auf, die sie von klassischen Computersystemen unterscheiden. Dazu zählen ihre starke Heterogenität, die Beschränkung der Hardware-Ressourcen, spezielle Kommunikationsprotokolle sowie vielfältige Einsatzumgebungen. Diese Faktoren haben unmittelbare Konsequenzen für die Entwicklung, die Sicherheit und die Testverfahren. Dies ist besonders relevant für den Einsatz von Fuzzing, das sich an die besonderen Randbedingungen dieser Geräte anpassen muss.

2.1.1 Architekturen und Protokolle von IoT-Geräten

IoT-Geräte bestehen typischerweise aus eingebetteten Hardwareplattformen, die mit Sensoren, Aktoren und Kommunikationsmodulen ausgestattet sind. Typische Architekturen umfassen mikrocontrollerbasierte Systeme wie den ESP32 oder STM32, Einplatinencomputer wie den Raspberry Pi sowie FPGA-basierte Lösungen für spezialisierte Anwendungen [SBV23]. Zur Kommunikation nutzen IoT-Geräte standardisierte Protokolle auf Basis von TCP/IP [Pos81a; Pos81b], wie Message Queueing Telemetry Transport (MQTT) [Ban+19], Constrained Application Protocol (CoAP) [SHB14], Extensible

Tabelle 2.1: Übersicht über verschiedene 32-Bit-Mikrocontroller. Quelle: [SBV23]

Mikrocontroller	Taktfrequenz	RAM	Schnittstellen
STM32F4 [STM25]	180 MHz	96 KB	UART, SPI, I^2C
Raspberry Pi Pico [Ras24]	133 MHz	264 KB	UART, SPI, I^2C
ESP8266 [Esp23]	80 MHz	96 KB	UART, SPI, I^2C (Software)
ESP32 [Esp25a]	240 MHz	520 KB	UART, SPI, I^2C, WLAN, BLE

Messaging and Presence Protocol (XMPP) [Sai11] und HTTP [Nie+99], beispielsweise für REST-Application Programming Interfaces (APIs) [Fie00]. Außerdem nutzen sie verschiedene Technologien wie Bluetooth Low Energy (BLE) [Blu23], Z-Wave [Z-W21] und ZigBee [Con23] für die Kurzstreckenkommunikation. Diese sind speziell auf die ressourcenbeschränkten Anforderungen von IoT-Geräten zugeschnitten und ermöglichen eine effiziente Datenübertragung in verschiedenen Anwendungsbereichen [Has+19].

2.1.2 Herausforderungen beim Fuzzing von IoT-Geräten

Das Fuzzing von IoT-Geräten stellt aufgrund der begrenzten Rechenressourcen und spezifischen Hardwarearchitekturen besondere Anforderungen an Fuzzing-Methoden dar. Repräsentative IoT-Plattformen auf Basis von 32-Bit-Mikrocontrollern wie dem STM32F4 [STM25], dem Raspberry Pi Pico [Ras24], dem ESP8266 [Esp23] oder dem ESP32 [Esp25a] verfügen typischerweise über folgende Ressourcen [SBV23]:

- **Rechenleistung:** Prozessoren mit Taktfrequenzen von 80 MHz bis zu 240 MHz
- **Speicher:** RAM-Kapazitäten zwischen 32 KB und 520 KB
- **Kommunikationsschnittstellen** [Gup19]**:** UART, SPI, I^2C, WLAN, BLE

Tabelle 2.1 gibt eine exemplarische Übersicht über gängige 32-Bit-Mikrocontroller und verdeutlicht die Vielfalt verfügbarer Plattformen. Die begrenzten Ressourcen von IoT-Geräten wirken sich direkt auf die Durchführung von Fuzzing (siehe Abschnitt 2.4) aus. Die eingeschränkte Speicherkapazität limitiert die Verwaltung umfangreicher Testdaten im Arbeitsspeicher, während die begrenzte Rechenleistung die Anzahl gleichzeitig verarbeitbarer Eingaben limitiert. Zusätzlich können langsame Kommunikationsschnittstellen die Geschwindigkeit der Fehlerdetektion erheblich reduzieren. Darüber hinaus erfordern unterschiedliche Hardware- und Softwarekonfigurationen spezifische Anpassungen der Fuzzing-Methoden, was die Entwicklung universell einsetzbarer Fuzzer zusätzlich erschwert [SBV23].

2.2 ESP32 Mikrocontroller

Für diese Dissertation wurde der ESP32 (siehe Abbildung 2.1) als repräsentatives IoT-Gerät ausgewählt, da er zu den am weitesten verbreiteten Mikrocontrollern zählt und breit in

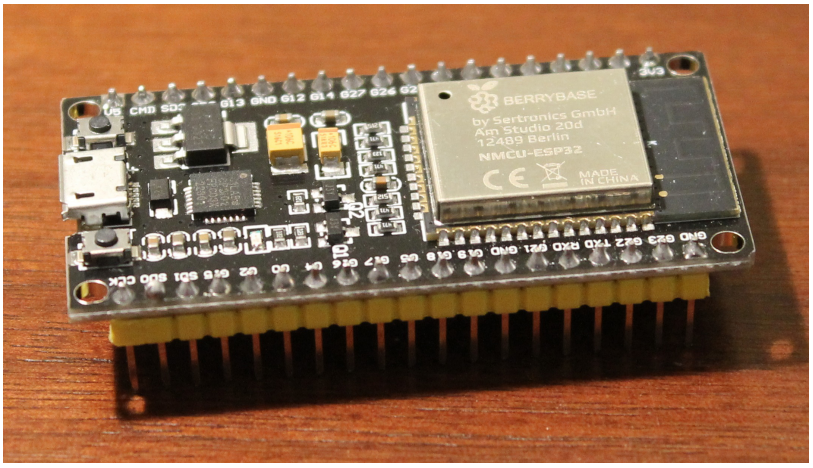

Abbildung 2.1: NMCU-ESP32: Ein ESP32 auf einem NodeMCU-Entwicklungsboard.

Forschung sowie Industrie eingesetzt wird. Der ESP32 bietet eine moderne Architektur mit integrierter WLAN- und Bluetooth-Konnektivität und unterstützt zahlreiche Schnittstellen. Diese Eigenschaften machen ihn zu einer vielseitigen Plattform, die typische Merkmale moderner IoT-Geräte repräsentiert. Darüber hinaus zeichnet sich der ESP32 durch seine offene Dokumentation, eine aktive Community und gute Verfügbarkeit aus, was ihn besonders geeignet macht, um experimentelle Ansätze systematisch zu untersuchen und zu evaluieren [Esp18; Eli22].

2.2.1 Architektur

Die ESP32-Familie von Espressif Systems umfasst energieeffiziente und kostengünstige Mikrocontroller, die speziell für IoT-Anwendungen entwickelt wurden. Im Vergleich zum Vorgängermodell ESP8266 verfügt der ESP32 über erweiterte Fähigkeiten hinsichtlich Rechenleistung, Speicherausstattung und Schnittstellenvielfalt. Abhängig von der Variante kommen entweder einzelne oder duale Xtensa LX6-Kerne [Cad25] bzw. LX7-Kerne [Cad24] oder ein Reduced Instruction Set Computer (RISC)-V-Kern [Wat+16; Esp25b] zum Einsatz. Während der LX7 gegenüber dem LX6 vor allem durch höhere Taktraten und optimierte Pipeline-Architektur mehr Leistung bietet, zeichnet sich RISC-V durch seine Offenheit, Modularität und breite Unterstützung in der Embedded-Community aus. Dabei implementieren alle Versionen die Xtensa-ISA (siehe Abschnitt 2.2.3) – eine anpassbare und erweiterbare RISC-basierte Befehlssatzarchitektur [Esp25a].

In dieser Dissertation wird der `ESP32-WROOM-32` [Esp25c] als Referenzplattform verwendet. Die wesentlichen technischen Eigenschaften sind in Tabelle 2.2 aufgeführt. Dank seiner integrierten WLAN- und BLE-Funktionen, seiner vergleichsweise hohen Rechenleistung und seines geringen Energieverbrauchs eignet sich der ESP32 besonders für ressourcenbeschränkte IoT-Szenarien. Gleichzeitig ist er aufgrund seiner in der Regel niedrigen Anschaffungskosten und seiner Vielseitigkeit auch in industriellen Anwendungen weit

Tabelle 2.2: Technische Spezifikationen des ESP32-WROOM-32 [Esp25c]

Eigenschaft	Wert
Mikroarchitektur	Xtensa 32-bit LX6
CPU-Kerne	2
Maximale Taktfrequenz	240 MHz
Flash-Speicher	4 MB
ROM	448 KB
SRAM	520 KB
WLAN	IEEE 802.11 b/g/n
Bluetooth	Classic und BLE

verbreitet. Dies unterstreichen nicht nur die hohen Verkaufszahlen, die bereits 2018 die Marke von 100 Millionen ausgelieferten Einheiten überschritten, sondern auch Berichte, die den ESP32 als wichtigen Treiber für vernetzte Anwendungen im Kontext von Industrie 4.0 und Industrie 5.0 hervorheben [Esp18; AAZ25].

2.2.2 Firmware

Eine Übersicht über die ESP32-Firmware ist in Abbildung 2.2 zu finden. Die Firmware enthält drei Hauptkomponenten: den Bootloader, die Partitionstabelle und mindestens eine App-Partition, die den Anwendungscode enthält. Der Bootloader und der Anwendungscode werden in Executable and Linkable Format (ELF)-Binärdateien kompiliert und anschließend in ein Binärformat umgewandelt, das für den ESP32 geeignet ist [Cir25].

Diese Binärdateien werden zusammen auf den Mikrocontroller geflasht. Obwohl sie nicht zu einer Datei zusammengefasst werden, werden der Bootloader, die Partitionstabelle und das Anwendungsabbild in dieser Dissertation gemeinsam als Firmware-Abbild bezeichnet. Das Modifizieren oder Austauschen einer dieser Komponenten führt in der Regel dazu, dass die Firmware nicht mehr korrekt funktioniert.

2.2.2.1 Bootloader

Der Bootloader wird zuerst ausgeführt, wenn der ESP32 eingeschaltet oder zurückgesetzt wird. Er initialisiert die Hardware, stellt die Systemuhr ein, konfiguriert den Speicher und überprüft die Integrität des Anwendungscodes, bevor er die Kontrolle an die Anwendung übergibt [Cir25].

2.2.2.2 Partitionstabelle

Die Partitionstabelle definiert die Anordnung des Flash-Speichers des ESP32, einschließlich der Position und Größe des Bootloaders, der Anwendung und anderer Partitionen.

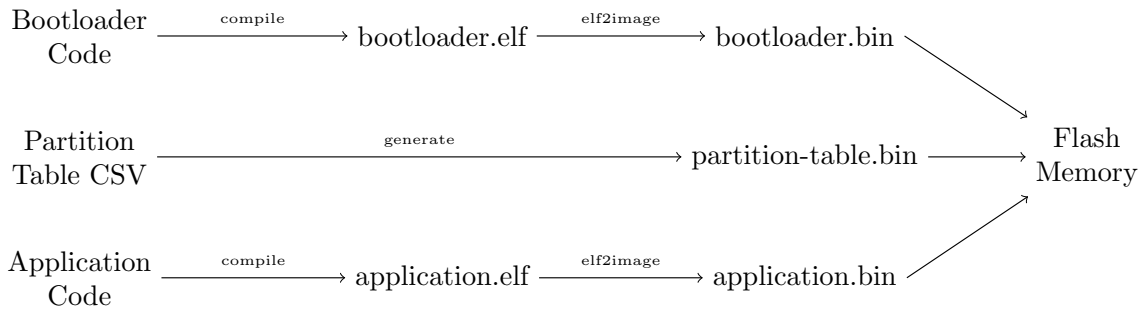

Abbildung 2.2: Der Build- und Flash-Prozess der ESP32-Firmware. Quelle: [Pla+25]

Partitionstabellen werden dabei in einfachen Comma-Separated Values (CSV)-Dateiformat [Sha05] definiert und während des Build-Prozesses in ein Binärformat kompiliert [Cir25].

2.2.2.3 Anwendungsabbild

Das Anwendungsabbild ist die Haupt-Firmware, die den Code für die Funktionalität des ESP32 enthält. Nachdem der Bootloader das System initialisiert hat, übergibt er die Kontrolle an das Anwendungsabbild, das dann die beabsichtigten Operationen des Geräts ausführt [Cir25].

2.2.3 Xtensa ISA

Die Xtensa-ISA [Cad22] ist eine RISC-basierte Architektur, die von Cadence Tensilica entwickelt wurde und sich durch ihre hohe Konfigurierbarkeit auszeichnet. Sie erlaubt es Entwicklern, den Befehlssatz gezielt an Anforderungen bezüglich Leistung, Speicherbedarf und Energieeffizienz anzupassen, unter anderem durch benutzerdefinierte Instruktionen und Register [Cad22].

Obwohl die Xtensa-ISA den meisten klassischen RISC-Prinzipien folgt, wie etwa einer Load/Store-Architektur und dass Instruktionen in einem „Clock Cycle“ abgearbeitet werden, bietet sie mit der *Code Density Option* eine Besonderheit: Ist diese aktiviert (Standardfall), können viele reguläre 24-Bit-Instruktionen durch kompaktere 16-Bit-Varianten ersetzt werden. Laut offizieller Dokumentation lassen sich in typischem Code etwa die Hälfte aller Befehle in diesem komprimierten Format darstellen, was den Speicherbedarf signifikant reduziert [Cad22].

Die „Windowed Register Option“ führt zusätzliche allgemeine Register ein, wodurch die Gesamtzahl von 16 auf 64 steigt. Um zu vermeiden, dass neue Anweisungen mit größeren Registerkodierungen (von 4 auf 6 Bit pro Register, das in der Anweisung verwendet wird) eingeführt werden müssen, bleibt die Anzahl der sichtbaren Register bei 16. Während Funktionsaufrufen kann dieses Fenster sichtbarer Register um ein Vielfaches von 8 verschoben

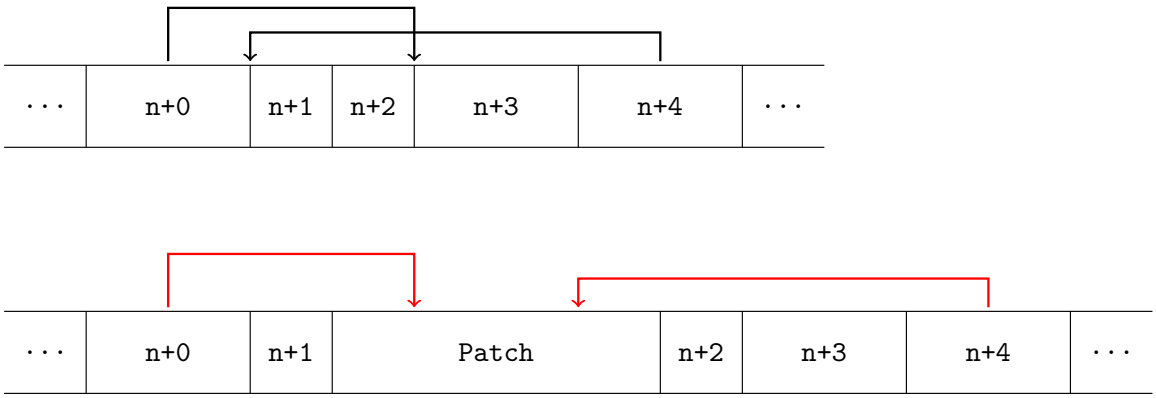

Abbildung 2.3: Das Code Location Problem wird beim Einfügungen von Code deutlich: Verschobene Instruktionen führen zu fehlerhaften relativen Sprüngen und Abstürzen. Der Sprung von Instruktion $n + 0$ zeigt nach dem Patch nicht mehr auf den Beginn von Instruktion $n + 3$, sondern auf den neu hinzugefügten Code. Es ist nicht sichergestellt, dass an dieser Stelle eine Instruktion beginnt, was zu einem Absturz führen kann. Quelle: [Pla+25]

werden. Diese Option reduziert die Anzahl der Register, die vor einem Funktionsaufruf auf dem Stack gespeichert werden müssen [Cad22].

Diese komplexen architektonischen Besonderheiten haben direkte Auswirkungen auf die Struktur der Firmware und die nachträgliche Modifikation des Binärcodes. Im folgenden Abschnitt wird daher Rewriting als Konzept vorgestellt. Dieses befasst sich mit den Herausforderungen und Verfahren zur Modifikation von Firmware-Abbildern.

2.3 Rewriting

Dieser Abschnitt gibt einen Überblick über ESP32-Firmware-Abbilder und das *Code Location Problem*, das beim Binary Rewriting auftritt, sowie verschiedene Kategorien des Binary Rewritings, d. h. statisches und dynamisches Rewriting.

2.3.1 Code Location Problem

Das *Code Location Problem* tritt auf, wenn der Binärcode eines Programms verändert wird. Es betrifft insbesondere Systeme, die relative Adressierung verwenden, beispielsweise bei Sprung- oder Verzweigungsanweisungen, deren Zieladressen relativ zu ihrer Speicherposition berechnet werden. Durch das Einfügen neuer oder die Modifikation bestehender Instruktionen verschieben sich die Speicheradressen nachfolgender Instruktionen (siehe Abbildung 2.3). Für Debugger stellt dies ein zentrales Problem dar, da sie auf eine präzise Zuordnung von Quellcodezeilen zu Instruktionsadressen angewiesen sind, um Breakpoints, das Schritthalten und das gezielte Betreten und Verlassen von Subroutinen korrekt zu

realisieren. Nach einer Codeverschiebung verweisen viele Sprünge nicht mehr auf ihre vorgesehenen Ziele, was fehlerhafte Kontrollflüsse, Programmabstürze oder unvorhersehbares Verhalten zur Folge haben kann [TG00].

Die nachträgliche Korrektur solcher Sprungadressen ist insbesondere beim ESP32 anspruchsvoll. Die Architektur unterstützt sowohl 16-Bit- als auch 24-Bit-Instruktionen, wodurch die exakte Position einzelner Befehle schwer vorhersagbar ist. Schon kleine Änderungen können Verschiebungen im gesamten nachfolgenden Code verursachen. Hinzu kommt, dass Code und Daten im Flash-Speicher häufig eng miteinander vermischt sind, was eine lineare Disassemblierung erschwert [Cad22; Esp25a].

2.3.2 Binary Rewriting

Unter *Binary Rewriting* versteht man das nachträgliche Verändern von bereits kompiliertem Maschinencode, ohne dass der ursprüngliche Quellcode benötigt wird. Typische Ziele sind das Einfügen von Instrumentierungscode (z. B. für Logging oder Fuzzing), das Anpassen von Programmlogik, das Einfügen von Sicherheitsmechanismen oder das Beheben von Fehlern direkt im Binärcode.

Das Binary Rewriting lässt sich grob in statische und dynamische Methoden unterteilen, die jeweils unterschiedliche Ansätze und Kompromisse aufweisen.

Beim dynamischen Rewriting erfolgen Anpassungen basierend auf dem tatsächlichen Verhalten des Programms zur Laufzeit. Diese Methode eignet sich besser für dynamischen oder selbstmodifizierenden Code, verursacht jedoch einen spürbaren Laufzeit-Overhead, da bei jeder Ausführung zusätzliche Kontroll- und Umleitungslogik berücksichtigt werden muss. Während dieser Mehraufwand bei Desktop- oder Serversystemen oft nur wenige Prozent Leistung kostet und durch hohe Rechenressourcen kompensiert werden kann, wirkt er sich in IoT-Szenarien deutlich stärker aus. Begrenzte Taktfrequenzen und kleiner Arbeitsspeicher machen dynamisches Rewriting für den ESP32 und vergleichbare Mikrocontroller meist unpraktikabel [SBF22].

Beim statischen Rewriting wird der Binärcode verändert, ohne ihn auszuführen, und es wird eine neue, modifizierte Binärdatei erzeugt. Diese Methode erlaubt eine gründliche Analyse und Optimierung vor der Ausführung. Allerdings kann das Neuberechnen von Sprungzielen innerhalb des Codes schwierig sein, insbesondere bei selbstmodifizierendem Code. Änderungen können dabei den ursprünglichen Kontrollfluss stören und neue Fehler einführen. Statische Rewriter arbeiten häufig auf einer sogenannten Intermediate Representation (IR) [Cho13], also einer Zwischendarstellung eines Programms zwischen Quellcode und Maschinencode, beispielsweise auf Assembler-Ebene. Auf dieser Ebene werden Anpassungen vorgenommen, bevor die Binärdatei wieder zusammengesetzt wird [SBF22].

Eine spezielle Technik des statischen Rewritings sind die sogenannten Trampolin-Rewriter. Dabei werden neue Instruktionen in ungenutzten Speicherbereichen abgelegt und der Kontrollfluss durch Sprünge dorthin umgeleitet (siehe Abbildung 2.4). So bleiben die ursprünglichen Instruktionen unverändert, während zusätzliche Funktionalitäten außerhalb

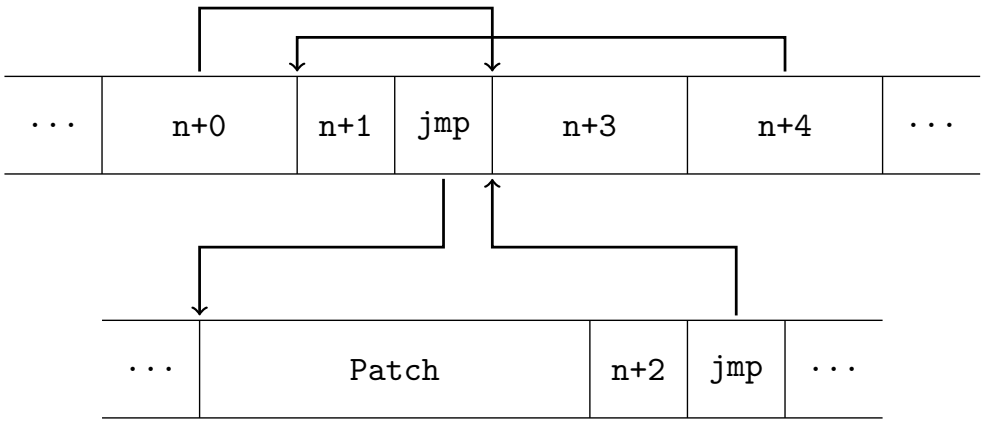

Abbildung 2.4: Trampolin-Rewriter bieten eine Lösung für das *Code Location Problem*. Dabei wird die ursprüngliche Instruktion durch einen Sprungbefehl ersetzt und sowohl die Instruktion als auch der Patch werden in ungenutzten Speicher verschoben. Dadurch bleiben alle nachfolgenden Adressen unverändert. Quelle: [Pla+25]

des Originalcodes integriert werden. Dadurch lässt sich die Kontrollflussintegrität wahren und das *Code Location Problem* effektiv umgehen [DGR20].

Die hier vorgestellten Techniken des Binary Rewritings bilden eine wichtige Grundlage für verschiedene Test- und Sicherheitsverfahren auf Mikrocontroller-Ebene. Im nächsten Abschnitt wird das Fuzzing behandelt, eine weit verbreitete Methode zur automatisierten Erkennung von Softwarefehlern. Dabei werden durch gezielte Eingabegenerierung Schwachstellen in Firmware und Software aufgedeckt.

2.4 Fuzzing

In diesem Unterkapitel wird der Begriff des Fuzzings als Testmethode zur Erkennung von Softwarefehlern durch die systematische Generierung und Einspeisung ungewöhnlicher oder ungültiger Eingaben definiert. Es werden zentrale Grundlagen vorgestellt, darunter typische Einsatzszenarien, Verfahren zur Eingabegenerierung und Ansätze des feedbackgesteuerten Fuzzings. Darüber hinaus werden Kriterien wie Codeabdeckung, Effizienz sowie syntaktische und semantische Korrektheit der Eingaben behandelt, um die methodische Einordnung und Messbarkeit des Fuzzings zu verdeutlichen.

2.4.1 Definition von Fuzzing

Fuzzing (kurz für *fuzz testing*) ist eine automatisierte Testtechnik zur Identifikation von Fehlern und Sicherheitslücken in Software. Dabei wird die Software wiederholt mit zufälligen oder systematisch generierten Eingaben versorgt, während das Laufzeitverhalten

überwacht wird, um unerwartete Reaktionen wie Abstürze, unbehandelte Ausnahmen oder Speicherlecks zu erkennen und potenzielle Schwachstellen aufzudecken [SGA07; Mhi+25; Lia+18].

Die Idee des Fuzzing entstand 1988 durch Barton P. Miller von der University of Wisconsin–Madison. Während eines Gewitters führten Störungen auf einer langsamen Modem-Leitung dazu, dass durch „Rauschen“ getippte Zeichen zufällig Unix-Utilities abstürzen ließen. Dieses scheinbar banale Phänomen inspirierte Miller dazu, das Konzept systematisch zu untersuchen. Im Rahmen eines Kurses sollten Studierende Programme gezielt mit zufälligen Eingaben testen, wodurch Fuzzing entstand. Bereits in der initialen Studie konnten durch dieses Verfahren 25–33 % der getesteten Programme zum Absturz gebracht werden. Dies war besonders bemerkenswert, da es die Anfälligkeit weit verbreiteter und scheinbar stabiler Software durch einfache zufällige Eingaben aufzeigte [TDM08].

2.4.2 Ungültige Eingaben

Eingaben im Rahmen von Fuzzing lassen sich typischerweise in drei Kategorien einteilen:

- *Gültige Eingaben:* Diese entsprechen vollständig der Spezifikation und werden vom Programm ohne Fehlermeldungen verarbeitet.
- *Semi-valide Eingaben:* Sie sind ein Sonderfall der gültigen Eingaben und erfüllen gerade noch die Spezifikation, um vom Parser akzeptiert zu werden, enthalten jedoch gezielt Variationen oder Extremwerte, um seltene Fehlerzustände zu provozieren.
- *Ungültige Eingaben:* Diese verletzen die grundlegende Struktur der erwarteten Daten und werden in der Regel unmittelbar vom Programm verworfen. Sie treten insbesondere bei stark strukturierten Formaten wie XML oder bei Netzwerkprotokollen auf und liefern häufig keine verwertbaren Rückmeldungen für die Laufzeitanalyse.

Da eine vollständige Abdeckung aller möglichen Eingaben praktisch unmöglich ist, liegt der Schwerpunkt neben den gültigen Eingaben vorwiegend auf den semi-validen Randfällen. Diese Strategie erhöht die Wahrscheinlichkeit, seltene oder schwer reproduzierbare Fehlerzustände zu identifizieren [Bha22].

2.4.3 Fuzzing-Szenarien

Fuzzing-Methoden unterscheiden sich vor allem durch den Grad des Wissens über das zu testende System. Üblich ist die Einteilung in Whitebox-, Blackbox- und Greybox-Fuzzing.

- **Whitebox-Fuzzing:** Der Fuzzer hat vollständigen Zugriff auf den Quellcode und die interne Programmlogik. Mittels statischer und dynamischer Analyse werden gezielt Eingaben generiert, um schwer erreichbare Codepfade abzudecken. Oft wird dynamische symbolische Ausführung (engl. „concolic execution“) eingesetzt. Während der Programmausführung werden symbolische Bedingungen gesammelt und gezielt negiert. Mithilfe eines Constraint-Solvers werden anschließend neue Eingaben

berechnet, die unerforschte Pfade ansteuern. Damit können theoretisch Testfälle generiert werden, die alle möglichen Ausführungspfade abdecken. In der Praxis ist dies jedoch oft nicht vollständig umsetzbar, da reale Software sehr viele potenzielle Pfade enthält und die Lösung der symbolischen Bedingungen komplex sein kann. Dennoch ermöglicht Whitebox-Fuzzing eine systematische und effiziente Fehlerentdeckung [GLM08; Lia+18].

- **Blackbox-Fuzzing:** Das Testen erfolgt ohne Kenntnis des Quellcodes. Die Eingaben werden zufällig oder heuristisch erzeugt, häufig aus bestehenden Beispielen, die durch Mutationen wie Bitflips, Byte-Kopien oder Löschungen verändert werden. Moderne Ansätze nutzen zusätzlich Wissen über das Grammatik- oder Eingabeformat, um teilweise gültige Eingaben zu erzeugen. Blackbox-Fuzzing ist einfach implementierbar und breit anwendbar, erreicht jedoch aufgrund fehlender Rückmeldungen zur Programmausführung meist nur eine geringe Codeabdeckung [Lia+18].
- **Greybox-Fuzzing:** Dieser Ansatz liegt zwischen White- und Blackbox-Fuzzing. Der Fuzzer erhält begrenzte Einblicke in das Programm, beispielsweise durch leichte Instrumentierung oder Feedback zur Codeabdeckung. Komplexe Techniken wie symbolische Ausführung werden nicht verwendet. Eingaben, die neue Pfade erreichen, werden gezielt wiederverwendet und als Basis für Mutationen genutzt. Dadurch steigt die Testeffektivität im Vergleich zu reinem Blackbox-Fuzzing [Lia+18; Zel+24].

In dieser Dissertation wird, soweit nicht anders angegeben, stets ein Greybox-Fuzzing-Szenario zugrunde gelegt.

2.4.4 Eingabegenerierung

Die Generierung von Testeingaben ist ein zentraler Bestandteil des Fuzzings. Eingaben können zufällig, durch Mutationen bestehender Beispiele oder systematisch auf Basis formaler Regeln erzeugt werden. Entsprechend werden Fuzzer häufig nach ihrer Methode der Eingabegenerierung unterschieden:

- **Dumb Fuzzer:** Diese Fuzzer erzeugen Eingaben vollständig zufällig, ohne Kenntnis des zugrunde liegenden Formats oder der erwarteten Datenstruktur. Sie arbeiten meist auf Byte- oder String-Ebene und führen einfache Manipulationen durch, wie das Einfügen, Löschen oder Verändern von Zeichen. Sie nutzen allenfalls grundlegende Heuristiken, erkennen aber keine komplexen Zusammenhänge. Aufgrund der vielen ungültigen Eingaben sind sie in komplexen Systemen nur begrenzt effektiv und eignen sich eher für einfache oder robuste Parser [LZZ18].
- **Mutationsbasierte Fuzzer:** Sie starten mit gültigen Eingaben (Seeds) und verändern diese durch zufällige oder heuristische Operationen wie Ersetzen, Einfügen oder Löschen von Bytes. Durch die Verwendung funktionierender Ausgangsdaten wird eine höhere Wahrscheinlichkeit erreicht, dass die Eingaben vom Programm verarbeitet werden. Diese Methode ist besonders effektiv, wenn hochwertige Seed-Dateien verfügbar sind [LZZ18].

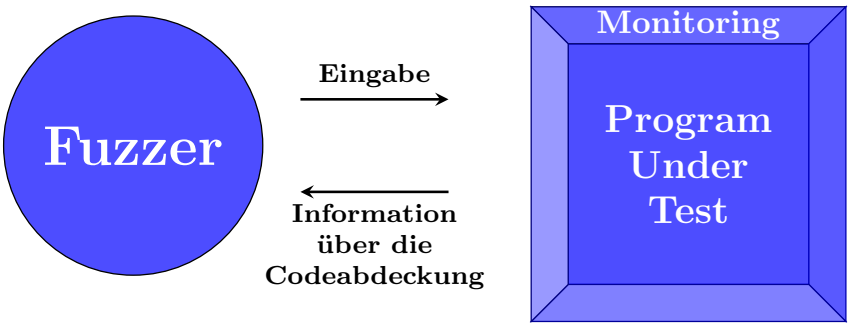

Abbildung 2.5: Schematische Darstellung von feedbackgesteuertem Fuzzing

- **Generationsbasierte Fuzzer:** Hier werden Eingaben vollständig neu erzeugt, basierend auf einer formalen Beschreibung des Eingabeformats, wie Datenmodellen, Protokollspezifikationen, Zustandsautomaten oder Ablaufdiagrammen. Diese Methode erzeugt syntaktisch gültige und semantisch anspruchsvolle Testfälle, die sich besonders für strukturierte Eingaben wie Netzwerkprotokolle, APIs oder komplexe Dateiformate eignen [LZZ18].
- **Grammatikbasierte Fuzzer:** Eine spezielle Untergruppe der generationsbasierten Fuzzer. Sie verwenden kontextfreie Grammatiken, um komplexe Eingaben zu erzeugen. Diese Technik eignet sich besonders für textbasierte Formate mit klar definierter Syntax, strukturierte Daten wie JSON oder XML mit eindeutiger Grammatik (siehe Abschnitt 2.5) [SS19].

2.4.5 Feedbackgesteuertes Fuzzing

Die Effektivität eines Fuzzers lässt sich deutlich erhöhen, wenn Rückmeldungen aus der Programmausführung zur gezielten Steuerung der Eingabegenerierung genutzt werden. Beim feedbackgesteuerten Fuzzing (engl. *Feedback-driven Fuzzing*, auch *Coverage-guided Fuzzing* genannt) überwacht der Fuzzer während der Ausführung, welche Codebereiche durch eine bestimmte Eingabe erreicht werden. Mithilfe dieser Informationen werden anschließend neue Eingaben so verändert, dass bisher unerreichte oder selten ausgeführte Pfade gezielt abgedeckt werden. Eine schematische Darstellung ist in Abbildung 2.5 zu finden. Vor allem mutationsbasierte Fuzzer profitieren stark von diesem Ansatz, da das Feedback es ihnen ermöglicht, Eingaben adaptiv zu optimieren und die Codeabdeckung systematisch zu steigern [Zal19].

2.4.6 Codeabdeckung und Messbarkeit von Fuzzing

Die einfachste Möglichkeit, die Wirksamkeit eines Fuzzers zu bewerten, wäre, die Anzahl der gefundenen Abstürze oder Schwachstellen innerhalb eines begrenzten Zeitraums

(z. B. 24 Stunden) zu messen. Allerdings ist oft unklar, ob das getestete Programm überhaupt Schwachstellen enthält und ob diese innerhalb des vorgegebenen Zeitraums gefunden werden können. Daher wird die Wirksamkeit von Fuzzing häufig anhand der Codeabdeckung (engl. *Code Coverage*) beurteilt. Diese gibt an, wie tief ein Fuzzer in die Programmlogik vordringt, ist aber kein direkter Indikator für die Anzahl gefundener Schwachstellen [Wan+20b].

Ein gängiges Maß für die Testabdeckung ist die sogenannte Zeilenabdeckung (engl. *Line Coverage*). Sie gibt an, welcher Anteil der Quellcodezeilen während eines Fuzzing-Laufs mindestens einmal ausgeführt wurde. Eine vollständige Zeilenabdeckung garantiert jedoch nicht, dass die zugrunde liegende Logik vollständig getestet wurde, denn unterschiedliche Eingaben innerhalb einer Zeile können zu abweichendem Verhalten führen. Eine hohe Zeilenabdeckung signalisiert hingegen, dass ein Großteil des Codes aktiviert und potenziell auf Fehler überprüft wurde. Beim feedbackgesteuerten, mutationsbasierten Fuzzing dient die Zeilenabdeckung als Leitwert für die Auswahl von Seeds. Eingaben, die eine hohe Zeilenabdeckung erzeugen, werden bevorzugt mutiert, um neue Testfälle zu erzeugen. Auf diese Weise lassen sich auch tief im Programm liegende Funktionen systematisch testen [WHJ15; BSM22].

Die Zweigabdeckung (engl. *Branch Coverage*) betrachtet zusätzlich alle logischen Verzweigungen, etwa beide Pfade eines `if`-Statements oder sämtliche `switch`-Fälle. Somit liefert sie ein detaillierteres Bild des Programmflusses und erleichtert die gezielte Optimierung von Testeingaben. Nicht abgedeckte Verzweigungen deuten auf unzureichende Eingaben hin [BSM22].

Da weder die Zeilen- noch die Zweigabdeckung direkt mit der Anzahl entdeckter Schwachstellen korreliert, werden zur objektiven Bewertung oft synthetische Bugs (engl. *Synthetic Bugs*) eingesetzt. Dabei werden bekannte Fehler gezielt in das PUT eingebaut, sodass überprüft werden kann, welche davon ein Fuzzer tatsächlich findet. Dieses Verfahren ermöglicht einen vergleichbaren und reproduzierbaren Test der Effektivität verschiedener Fuzzing-Strategien [Bun+21].

Der folgende Abschnitt befasst sich deshalb mit dem Thema „Effizientes Fuzzing“ und beleuchtet diese Aspekte näher. Dabei werden auch wichtige Kennzahlen zur Bewertung vorgestellt.

2.4.7 Effizientes Fuzzing

Im Kontext des Fuzzings bezeichnet Effizienz das Verhältnis zwischen eingesetztem Ressourcenaufwand, beispielsweise Zeit, Rechenleistung oder Energie, und dem erzielten Testfortschritt. Im Unterschied dazu bezeichnet Effektivität die Fähigkeit, tatsächlich sicherheitsrelevante Schwachstellen oder Bugs aufzudecken [GGG22]. Ein Fuzzer kann also sehr effizient arbeiten (z. B. durch hohe Eingabeverarbeitungsraten), ohne notwendigerweise effektiv zu sein, wenn er dabei keine relevanten Fehler findet.

Welche Kennzahl zur Bewertung der Effizienz herangezogen wird, hängt davon ab, welcher Aspekt des Testfortschritts im jeweiligen Szenario erreichbar und sinnvoll messbar ist:

- **Fehlerentdeckungsrate:** Ist der Quellcode zugänglich, lassen sich gezielt synthetische Bugs einbauen und deren Anzahl exakt bestimmen. In diesem Fall kann die Effizienz unmittelbar als Geschwindigkeit der Fehlerentdeckung definiert werden, da gemessen werden kann, wie viele dieser bekannten Schwachstellen pro Zeiteinheit aufgedeckt werden. Je schneller diese Bugs gefunden werden, desto direkter werden die eingesetzten Ressourcen in sicherheitsrelevanten Fortschritt umgesetzt [GGG22].
- **Codeabdeckung:** Ist der Code nicht veränderbar oder die Anzahl der Bugs unbekannt, bleibt die Codeabdeckung die praktikabelste Messgröße. Vor allem Greybox-Fuzzer wie AFL instrumentieren den Binärcode so, dass die Abdeckung die tatsächlich durchlaufenen Pfade widerspiegelt. Auch wenn dies durch Compiler-Optimierungen nicht exakt den Quellcodezeilen entspricht, erlaubt es dennoch einen direkten Vergleich verschiedener Ansätze hinsichtlich der erreichten Programmpfad-Abdeckung. Eine hohe Abdeckung weist darauf hin, dass mit den gleichen Ressourcen ein größerer Programmteil getestet wurde, was aus Effizienzsicht vorteilhaft ist [LZZ18; Zal19].
- **Verarbeitungsrate:** Selbst ohne vollständigen Fuzzer kann die Effizienz einzelner Komponenten anhand ihrer Eingabeverarbeitungsrate bewertet werden. Besonders auf ressourcenbeschränkten IoT-Geräten ist die Zahl der pro Sekunde verarbeiteten Anfragen begrenzt, während virtuelle Testumgebungen höhere Raten erlauben. Je mehr Eingaben in derselben Zeit verarbeitet werden, desto effizienter werden die verfügbaren Ressourcen genutzt [Bör+20].
- **Qualität der Eingaben:** Nicht nur die Menge, sondern auch die Vielfalt und Relevanz der generierten Eingaben beeinflussen die Effizienz. Ungültige Eingaben aktivieren oft nur triviale Programmteile, wodurch die Testtiefe sinkt. Werden hingegen gezielt Eingaben generiert, die möglichst viele verschiedene Codepfade auslösen, steigt die Wahrscheinlichkeit, kritische Teile des Programms mit begrenztem Ressourceneinsatz zu testen [Mhi+25].

Zusammenfassend lässt sich die Effizienz beim Fuzzing also als das Verhältnis von eingesetzten Ressourcen zu erzieltem Testfortschritt verstehen. Je nach Kontext kann der Fortschritt über Fehlerfunde, Codeabdeckung, Eingabedurchsatz oder Eingabequalität gemessen werden. Diese unterschiedlichen Perspektiven sind komplementär und verdeutlichen, dass Effizienz stets relativ zu den Testzielen interpretiert werden muss.

Um den Erfolg eines Fuzzers jedoch noch gezielter zu verbessern, ist es wichtig, neben der Menge auch die Qualität der generierten Eingaben zu betrachten. Dabei spielen insbesondere die syntaktische und semantische Korrektheit der Testeingaben eine entscheidende Rolle. Auf diesen Aspekt wird im Folgenden näher eingegangen.

2.4.8 Syntaktische und semantische Korrektheit

Eingaben, Datenpakete oder Dokumente können hinsichtlich ihrer *syntaktischen* und *semantischen* Korrektheit unterschieden werden [FS00; Ais25]:

- **Syntaktische Korrektheit:** Die Eingabe erfüllt die formalen Strukturregeln eines Datenformats oder Protokolls. Bei XML bedeutet dies korrekt geschlossene Tags, gültige Attribute und konsistente Verschachtelungen. Bei Netzwerkpaketen müssen alle erforderlichen Headerfelder vorhanden sein, die Feldlängen müssen stimmen und die Checksummen müssen korrekt sein. Syntaktisch korrekte Eingaben werden vom Parser akzeptiert und können vom System verarbeitet werden.
- **Semantische Korrektheit:** Die Eingabe ist inhaltlich konsistent und entspricht den logischen Erwartungen des Systems. Ein XML-Dokument ist semantisch korrekt, wenn die Feldinhalte den erwarteten Datentypen und zulässigen Wertebereichen entsprechen. Ein Netzwerkpaket ist semantisch korrekt, wenn Sequenznummern, Befehle oder Parameter konsistent mit dem Protokoll sind und sinnvoll interpretiert werden können.

Für das Fuzzing bildet die syntaktische Korrektheit die notwendige Grundlage, da nur korrekt geparste Eingaben verarbeitet werden können. Darüber hinaus steigert semantische Korrektheit die Wahrscheinlichkeit, seltene Fehlerzustände und Sicherheitslücken zu identifizieren, insbesondere bei stark strukturierten Formaten wie XML und komplexen Netzwerkprotokollen [Bra+08; Vis+11].

Um diese Aspekte der Korrektheit bei der Eingabegenerierung gezielt zu berücksichtigen, wird im folgenden Abschnitt das Konzept des grammatikbasierten Fuzzings erläutert. Es nutzt formale Grammatiken, um strukturierte und gültige Testfälle zu erzeugen.

2.5 Grammatiken für Fuzzing

Grammatikbasiertes Fuzzing ist eine spezialisierte Form des generationsbasierten Fuzzings. Es nutzt formale Grammatiken zur präzisen Beschreibung syntaktischer Regeln und ermöglicht so die Erzeugung strukturell gültiger Eingaben. Dies ist besonders bei komplexen oder stark regulierten Formaten von Vorteil [GLM08; Zel+24].

2.5.1 Aufbau der Grammatik

Eine Grammatik beschreibt eine formale Struktur, die definiert, wie gültige Zeichenketten einer Sprache gebildet werden können. Sie besteht im Wesentlichen aus einer endlichen Menge von Terminalsymbolen, die die konkreten Zeichen der Sprache darstellen, einer Menge von Variablen (auch Nicht-Terminalsymbole genannt), die als Platzhalter für syntaktische Kategorien dienen, sowie einer Menge von Produktionsregeln, die angeben, wie Variablen durch andere Variablen oder Terminale ersetzt werden können. Eine dieser Variablen ist als *Startsymbol* ausgezeichnet und dient als Ausgangspunkt für die Erzeugung von Zeichenketten [HMU11].

2.5.2 Formale Beschreibung von Grammatiken

Formal wird eine Grammatik G als ein Tupel $G = (V, T, P, S)$ definiert, bestehend aus:

- V: der endlichen Menge der *Variablen* (Nicht-Terminalsymbole), die Platzhalter für syntaktische Strukturen darstellen,
- T: der endlichen Menge der *Terminalsymbole*, die die konkreten Zeichen der Sprache bilden,
- P: der endlichen Menge der *Produktionsregeln*, die definieren, wie Variablen durch Terminale oder andere Variablen ersetzt werden,
- $S \in V$: dem *Startsymbol*, von dem aus die Erzeugung gültiger Zeichenketten beginnt.

Eine Produktionsregel hat die Form

$$A \rightarrow \alpha$$

wobei $A \in V$ eine Variable und $\alpha \in (V \cup T)^*$ eine Zeichenkette aus Terminalen und/oder Variablen ist. Durch wiederholte Anwendung dieser Regeln lassen sich syntaktisch gültige Zeichenketten konstruieren [HMU11].

2.5.3 Kontextfreie Grammatiken

Eine *kontextfreie Grammatik* ist eine spezielle Klasse formaler Grammatiken, bei der jede Produktionsregel genau eine Variable auf der linken Seite besitzt:

$$A \rightarrow \alpha \quad \text{mit} \quad A \in V,\ \alpha \in (V \cup T)^*$$

Die Anwendbarkeit einer Regel hängt dabei nicht vom Kontext der umgebenden Symbole ab.

Kontextfreie Grammatiken sind besonders relevant für das Fuzzing, da viele Datenformate, Kommunikationsprotokolle und Programmiersprachen mit ihnen vollständig beschrieben werden können. Grammatik-basierte Fuzzer können daher gezielt syntaktisch gültige Testeingaben generieren, was die Wahrscheinlichkeit erhöht, dass diese vom Zielprogramm akzeptiert und verarbeitet werden [GLM08; HMU11].

2.5.4 Grammatik von XML

Als Beispiel für eine weit verbreitete Grammatik wurde XML gewählt. Es handelt sich um ein standardisiertes, hierarchisch strukturiertes Datenformat, dessen Grammatik in der *XML 1.0 Specification* definiert ist [Bra+08].

Ein Fuzzer, der diese Grammatik berücksichtigt, kann gültige XML-Dokumente erzeugen, die von Parsern verarbeitet werden können. Aufgrund der breiten Nutzung von XML in Webservices, IoT-Protokollen und Konfigurationsformaten ist grammatikbasiertes Fuzzing in diesem Bereich besonders relevant. Fehler bei der Verarbeitung können schwerwiegende Sicherheitslücken verursachen, beispielsweise durch *XML External Entity Attacks* (XXE) oder *Billion Laughs Attacks* [Spä+16].

2.5.5 Vorteile des grammatikbasierten Fuzzings

Grammatikbasiertes Fuzzing bietet mehrere entscheidende Vorteile:

- **Höhere Eingabevalidität:** Generierte Testfälle sind syntaktisch gültig und werden daher vom PUT häufiger akzeptiert.
- **Höhere Codeabdeckung:** Durch gültige Eingaben lassen sich komplexe Verarbeitungspfade testen, die zufällige Eingaben nicht erreichen. Dadurch wird tiefer in das System vorgedrungen, wodurch auch versteckte Logik und selten genutzte Funktionen getestet werden können.
- **Effiziente Fehlerfindung:** Da weniger ungültige Eingaben erzeugt werden, wird weniger Rechenzeit auf unproduktive Testfälle verschwendet.
- **Gezielte Testgenerierung:** Grammatiken lassen sich erweitern oder einschränken, um bestimmte Strukturen oder Features des PUT gezielt zu testen.

Diese Eigenschaften machen grammatikbasiertes Fuzzing besonders geeignet für Szenarien, in denen stark strukturierte Eingaben verarbeitet werden, wie etwa bei Dateiformaten, Netzprotokollen oder Skriptsprachen [GKL08].

Um die Effektivität und Analysefähigkeit beim Fuzzing weiter zu steigern, bietet die Emulation eine flexible und kontrollierte Umgebung, in der Programme ausgeführt und gezielt manipuliert werden können. Im folgenden Abschnitt wird dargestellt, wie die Emulation als Ergänzung zum klassischen Fuzzing eingesetzt wird und welche Vorteile sie insbesondere bei der Instrumentierung von Firmware und Software bietet.

2.6 Fuzzing im Emulator

Die Emulation ermöglicht die Ausführung von Programmen in einer kontrollierten Umgebung und bietet mehrere zentrale Vorteile für den Fuzzing-Prozess:

- **Zielausführung mit Fuzzing-Hooks:** Fuzzing-Hooks ermöglichen eine gezielte Eingabesteuerung und die Analyse der Reaktionen des emulierten Systems auf unterschiedliche Eingaben. Dadurch lassen sich potenzielle Schwachstellen und deren Auswirkungen auf die Programmausführung untersuchen [HN17].
- **Codeabdeckung durch Emulation:** Die Emulation ermöglicht eine präzise Erfassung von Codeabdeckungsmetriken, beispielsweise in Form von Anweisungs- oder Zweigabdeckung. Diese Messungen liefern wertvolle Informationen darüber, welche Programmteile während des Fuzzing-Prozesses ausgeführt wurden. Auf dieser Grundlage kann die Testfallgenerierung gezielt optimiert werden, um die Abdeckung schrittweise zu erhöhen und ungetestete Programmregionen zu erreichen [HN17].
- **Registerzugriff in der Emulation:** Ein wesentlicher Vorteil der Emulation ist der direkte Zugriff auf Register und Speicherbereiche. Dadurch können die internen

Zustände des Programms überwacht, modifiziert und für Debugging-Zwecke analysiert werden. Diese Einblicke erleichtern die Identifikation von Speicherfehlern, unerwarteten Zustandsübergängen oder sicherheitskritischen Abweichungen im Kontrollfluss [Wri+21].

- **Fork-Join-Mechanismen:** Viele Emulatoren unterstützen Fork-Join oder Snapshotting, um Ausführungszustände einzufrieren und schnell wiederherzustellen. Dies ermöglicht parallele Testausführungen und steigert die Fuzzing-Geschwindigkeit erheblich, da wiederholte Initialisierungen vermieden werden [MRR12].
- **Determinismus und Reproduzierbarkeit:** Das deterministische Verhalten vieler Emulatoren sorgt dafür, dass identische Eingaben unter gleichen Bedingungen stets dieselben Zustände und Ergebnisse erzeugen. Diese Eigenschaft erleichtert das Debuggen und die reproduzierbare Analyse gefundener Schwachstellen [HN17].

Während die Emulation somit eine präzise Analyseumgebung für Firmware und Software bietet, ist für die umfassende Absicherung von vernetzten Systemen auch das Verständnis der zugrunde liegenden Kommunikationsprotokolle von zentraler Bedeutung. Das folgende Kapitel widmet sich daher den Grundlagen und Typen von Netzwerkprotokollen, die für die strukturierte Datenübertragung und Kommunikation in IoT-Systemen essenziell sind.

2.7 Netzwerkprotokolle

Netzwerkprotokolle bilden die Grundlage für die strukturierte Kommunikation in vernetzten Systemen. Sie definieren Regeln und Konventionen, die festlegen, wie Daten zwischen verschiedenen Endpunkten übertragen und interpretiert werden [TW11]. So legt ein Protokoll beispielsweise fest, in welchem Format und in welcher Abfolge Daten an ein Zielsystem übermittelt werden müssen, um bestimmte Aktionen auszulösen. Ebenso regelt es die Bedeutung und Verarbeitung der empfangenen Antworten. Dabei unterscheidet man zwischen *zustandsbehafteten Protokollen*, die Verbindungsinformationen speichern und so eine kontrollierte und zuverlässige Kommunikation ermöglichen (z. B. Transmission Control Protocol (TCP) [Pos81b]), und *zustandslosen Protokollen*, die ohne solche Kontextinformationen arbeiten (z. B. User Datagram Protocol (UDP) [Pos80]) [TC984].

Um die Vielzahl an Netzwerkprotokollen und deren Zusammenhänge zu strukturieren, wird häufig das von der International Organization for Standardization (ISO) standardisierte *ISO/OSI-Referenzmodell* herangezogen. Es unterteilt den Kommunikationsprozess in sieben klar definierte Schichten (siehe Abbildung 2.6), die jeweils spezifische Aufgaben erfüllen und über wohldefinierte Schnittstellen miteinander interagieren. Von unten nach oben werden folgende Schichten unterschieden [Pos81b; Aun10; MS12]:

1. **Bitübertragungsschicht (Physical Layer):** verantwortlich für die physische Übertragung von Rohbits über das Übertragungsmedium. Hier werden elektrische, optische und Funk-Schnittstellen definiert. Beispiele für physische Medien sind Kupferkabel, Glasfaser und Funkfrequenzen.

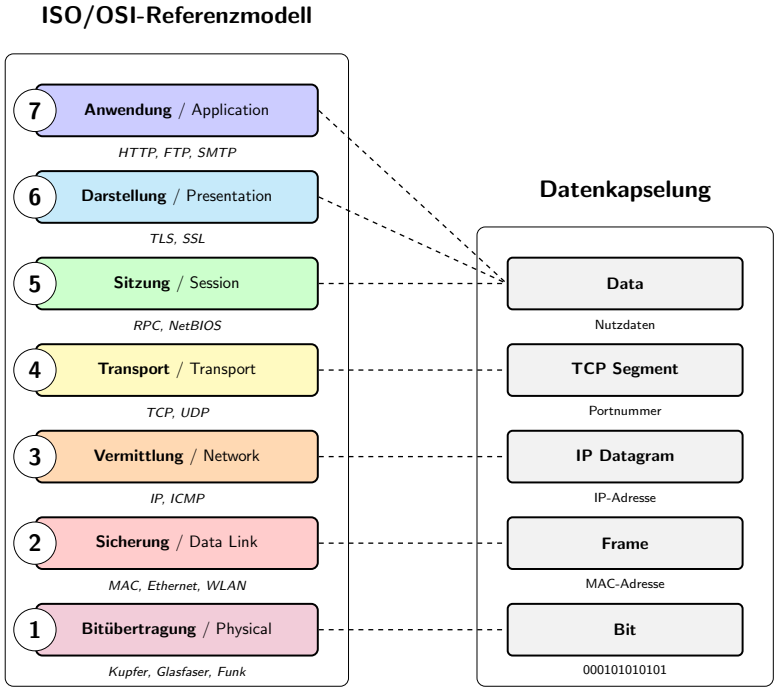

Abbildung 2.6: Das ISO/OSI-7-Schichtenmodell. Quelle: frei nach [Aun10].

2. **Sicherungsschicht (Data Link Layer):** gewährleistet eine fehlerfreie Übertragung zwischen zwei direkt verbundenen Netzwerkknoten. Daten werden in Frames verpackt, mit Prüfsummen versehen und der Zugriff auf das Übertragungsmedium wird geregelt. Hier kommen MAC-Adressen zur eindeutigen Identifizierung von Geräten zum Einsatz. Typische Protokolle sind Ethernet, WLAN und PPP.
3. **Vermittlungsschicht (Network Layer):** zuständig für die logische Adressierung, das Routing und die Weiterleitung von Datenpaketen über Netzgrenzen hinweg. Hier entstehen IP-Datagramme, die eine IP-Adresse enthalten, um Sender und Empfänger eindeutig zu identifizieren. Typische Protokolle: IP, ICMP, IPSec.
4. **Transportschicht (Transport Layer):** stellt die Ende-zu-Ende-Kommunikation zwischen Anwendungen sicher. Hier erfolgt die Segmentierung der Daten, ihre Nummerierung und Zuordnung zu Portnummern. TCP sorgt für zuverlässige, verbindungsorientierte Übertragungen, während UDP für schnelle, verbindungslose Kommunikation genutzt wird, z. B. bei Streaming oder VoIP.
5. **Sitzungsschicht (Session Layer):** verwaltet Sitzungen zwischen Anwendungen, steuert den Auf- und Abbau von Verbindungen und synchronisiert Datenströme. Sie ermöglicht das Fortsetzen von Dialogen nach Unterbrechungen. Beispiele: NetBIOS, RPC.
6. **Darstellungsschicht (Presentation Layer):** übersetzt Daten in ein einheitliches Format und kümmert sich um Kodierung, Kompression und Verschlüsselung. Hier arbeiten beispielsweise SSL/TLS sowie Datenformate wie JPEG, JSON oder ASCII.

7. **Anwendungsschicht (Application Layer):** bietet Schnittstellen für benutzernahe Anwendungen und definiert die Bedeutung der übertragenen Daten. Hier laufen die eigentlichen Dienste und Protokolle, z. B. HTTP, FTP, SMTP, DNS oder SNMP.

2.7.1 Struktur eines TCP-Headers

TCP operiert auf der Transportschicht (Layer 4). Ein TCP-Header enthält zentrale Steuerinformationen, darunter:

- **Quell- und Zielport:** zur Identifikation der kommunizierenden Anwendungen,
- **Sequenz- und Bestätigungsnummern:** zur Sicherstellung der korrekten Datenreihenfolge und Empfangsbestätigung,
- **Steuerflags:** (z. B. SYN, ACK, FIN) für Aufbau, Abbau und Kontrolle der Verbindung,
- **Fenstergröße:** zur Regulierung der Flusskontrolle.

Diese Felder bilden die Grundlage für die Zuverlässigkeit von Protokollen. In den folgenden Abschnitten werden die anwendungsnahen Protokolle **FTP** und **HTTP** exemplarisch betrachtet. Diese eignen sich besonders gut, da sie eine klare, textbasierte Struktur besitzen [Pos81b].

2.7.2 File Transfer Protocol (FTP)

In dieser Dissertation wird das FTP als beispielhaftes Anwendungsprotokoll herangezogen, da seine einfache, textbasierte Struktur die Analyse unverschlüsselter Nachrichten im Rahmen von PRE (siehe Abschnitt 2.7.5) ermöglicht. Im Gegensatz zu verschlüsselten Alternativen wie SFTP oder FTPS können hier die Inhalte direkt untersucht werden.

FTP ist zustandsbehaftet und verwendet zwei separate Verbindungen: eine Kontrollverbindung für Kommandos und Statusmeldungen sowie eine Datenverbindung für Dateiübertragungen. Für diese Dissertation wird ausschließlich die Kontrollverbindung betrachtet. Kommandos bestehen aus einem Schlüsselwort und optionalen Argumenten, Serverantworten aus Statuscodes und Text. Der Standard lässt sich durch implementierungsspezifische Kommandos erweitern.

Ein typischer Ablauf beginnt mit der Serverbegrüßung und der Authentifizierung mittels `USER` und `PASS`. Ein häufiger Anwendungsfall ist der anonyme Login, bei dem der Benutzername `anonymous` und ein beliebiges Passwort verwendet werden kann, üblicherweise `guest` oder eine E-Mail-Adresse. Im Anonymous-Modus ist der Zugriff auf das Serverdateisystem in der Regel eingeschränkt, etwa auf das Auflisten und Herunterladen von Dateien. Nach erfolgreicher Anmeldung kann der Client beispielsweise das aktuelle Verzeichnis abfragen (`PWD`) oder eine Datenverbindung aufbauen. Eine exemplarische FTP-Sitzung ist in Listing 2.1 dargestellt [PR85; DEM94].

```
< 230-Welcome user to FTP server
< 230-ProFTPD Server (FTPD) [XXX.XXX.XXX.XXX]
< 230 Please log in.
> USER anonymous
< 331 Anonymous login ok, send email address as your password
> PASS john@doe.com
< 230 Anonymous access granted, restrictions apply
> PWD
< 257 "/" is the current directory
```

Listing 2.1: Eine typische FTP-Kommunikation. Client-Pakete sind mit „>“ markiert, Server-Pakete mit „<“. Quelle: [Kie+22]

2.7.3 Hypertext Transfer Protocol (HTTP)

HTTP ist seit 1991 ein zentrales Protokoll des Internets, primär zur Übertragung von Webseiten. Als zustandsloses Protokoll enthält jede Anfrage alle notwendigen Informationen zur Verarbeitung. Ein Request umfasst die Methode (z. B. `GET`, `POST`), den Pfad und die HTTP-Version. Die Antwort enthält mindestens die Version und einen Statuscode [Nie+99].

Die Statuscodes werden dabei grob in fünf Kategorien unterteilt:

- `1xx`: Information
- `2xx`: Erfolg
- `3xx`: Weiterleitung
- `4xx`: Clientfehler
- `5xx`: Serverfehler

Einige Beispiele für Statuscodes sind `200` „OK“, `301` „Moved Permanently“ und `404` „Not Found“ [Nie+99].

HTTP-Nachrichten können zusätzliche Header-Felder transportieren, z. B. `Content-Type`, `User-Agent` oder `Cookie`, und optional einen Body enthalten (z. B. HTML oder JSON). Für diese Dissertation wird ausschließlich HTTP/1.1 betrachtet; neuere Versionen wie HTTP/2 oder HTTP/3 werden aufgrund der erhöhten Komplexität nicht analysiert. Laut Cloudflare entfallen aktuell etwa 50 % der API-Zugriffe auf HTTP/1.1 und nur rund 12 % auf HTTP/3 [BP23].

2.7.4 Angriffsvektoren auf Netzwerkprotokolle

Fehlerhafte Implementierungen von Netzwerkprotokollen entstehen häufig durch unzureichende Tests, missverstandene Spezifikationen oder unbeachtete Randfälle. Solche

Schwachstellen bieten Angreifern Möglichkeiten, um Systeme zu destabilisieren, unerwartetes Verhalten auszulösen oder unberechtigten Zugriff auf vertrauliche Daten zu erlangen. Typische Angriffsvektoren lassen sich wie folgt kategorisieren [SWS07]:

- **Denial of Service (DoS):** Ein System wird durch Überlastung (z. B. hohe Anfragefrequenz) oder durch Ausnutzung logischer oder implementierungsbedingter Fehler in seiner Verfügbarkeit eingeschränkt. Beispiele sind Ressourcenerschöpfung, Endlosschleifen oder speziell gestaltete Nachrichten, die intensive Verarbeitungspfade auslösen [IHR06].
- **Injections (z. B. SQL-Injection):** Werden Eingaben ohne sichere Behandlung in nachfolgende Verarbeitungsschichten eingebracht (z. B. in Datenbankabfragen), können Angreifer Codeausführung, unautorisierten Datenzugriff oder Datenmanipulationen erreichen. Solche Injections betreffen nicht nur SQL, sondern prinzipiell jede Schicht, die Eingaben interpretiert (Command-, XPath-, OS-Injection u. ä.) [Cla09].
- **Implementierungsfehler:** Fehlende Längenprüfungen, inkorrekte Grenzfallbehandlung oder Pointer-Fehler führen zu Schwachstellen (z. B. Pufferüberlauf). Historische Beispiele wie der Heartbleed-Bug [Bla25] in OpenSSL zeigen, dass ungenügende Prüfungen zu Auslese sensibler Speicherbereiche führen können und damit Vertraulichkeit und Integrität gefährden [SWS07].
- **Ausnutzung falscher Zustände:** Das Senden von Nachrichten in falscher Reihenfolge oder das Erzeugen inkonsistenter Zustände kann in zustandsbehafteten Protokollen zu unerwartetem Verhalten führen (z. B. Verarbeitungsfehler bei `DELETE` vor `CREATE`). Fehlt eine robuste Zustandsvalidierung, können dadurch Berechtigungsfehler oder inkonsistente Datenzustände entstehen [SWS07].

2.7.5 Protocol Reverse Engineering (PRE)

Beim PRE wird die Spezifikation eines bislang unbekannten Kommunikationsprotokolls systematisch aus beobachtbarem Netzwerk- oder Schnittstellenverkehr rekonstruiert. Das Ziel besteht darin, strukturelle Informationen wie Nachrichtenformate, Feldgrenzen, Konstanten, Nachrichtentypen und gegebenenfalls ein implizites Zustandsmodell zu extrahieren. Typische Ergebnisse sind eine formale Nachrichtensyntax (z. B. als Grammatik oder Feldlayout) und ein Modell der Kontrollflusslogik (z. B. als endlicher Automat) [Won+08; ANV11].

Der übliche PRE-Workflow umfasst das Erfassen und Vorverarbeiten von Traffic, die Segmentierung von Nachrichten, die Inferenz von Feldgrenzen und Datentypen, die Induktion von Syntaxregeln sowie das Lernen eines Zustandsmodells. Zur Umsetzung werden diverse Techniken kombiniert: Hierzu zählen Clusteranalyse und Feature-Engineering zur Gruppierung ähnlicher Nachrichten, heuristische Verfahren zur Felddetektion (Delimiter-Erkennung, Entropieanalyse) sowie Algorithmen zur Grammatikinduktion und formale Lernverfahren für Automaten. Praktische Tools verknüpfen häufig passive Beobachtung mit aktiven Maßnahmen (gezielte Anfragen), um nicht beobachtete oder verschleierte Protokollpfade zu erschließen [CKW07; Com+09; ANV11].

Protokolle lassen sich für PRE entlang zweier Dimensionen klassifizieren: textbasiert versus binär sowie zustandslos versus zustandsbehaftet. Textbasierte Formate verwenden druckbare Zeichen und Delimiter, während binäre Formate mit kompakten Bytefolgen, Längenfeldern und Bitfeldern arbeiten. Mischformen sind ebenfalls möglich. Zustandsbehaftete Protokolle besitzen implizite oder explizite Zustandsmaschinen, deren Rekonstruktion für sinnvolle, zustandsbewusste Testvektoren erforderlich ist. Bei zustandslosen Protokollen hingegen reichen oft unabhängige Nachrichtenmutationen aus [Sch08; Bha25].

Ein erfolgreiches PRE ermöglicht eine weitergehende Analyse der Kommunikation, wie tiefgehende Paketinspektion [Bro18] und Fuzzing [Bha22]. Beide Verfahren können effektiver arbeiten, wenn sie über die detaillierte Spezifikation des Protokolls verfügen [SM07; Com+09].

Das Vorgehen beim PRE lässt vermuten, dass KI hier ein besonders großes Potenzial bietet. KI kann Protokolle automatisiert analysieren, verborgene Zusammenhänge erkennen und somit die Effizienz und Tiefe der Analyseverfahren deutlich steigern. Dadurch wird PRE zu einem wichtigen Anwendungsfeld moderner KI-Technologien.

Aufbauend darauf bietet sich der Einsatz neuronaler Netzwerke an, die komplexe Muster in großen Datenmengen erkennen und verarbeiten können. Im folgenden Kapitel werden daher die grundlegenden Konzepte ihrer Architekturen vorgestellt, die für KI-gestützte Analyseverfahren von zentraler Bedeutung sind.

2.8 Neuronale Netzwerkarchitekturen

Neuronale Netze gehören zu den zentralen Methoden des maschinellen Lernens. Ihr Grundprinzip besteht darin, Eingaben durch eine Abfolge miteinander verbundener Verarbeitungseinheiten in Ausgaben zu überführen. Dabei werden die Verbindungen zwischen den Einheiten mit Gewichtungen versehen, deren Anpassung es ermöglicht, aus Daten zu lernen und Muster zu erkennen. Dieses Vorgehen erlaubt es, auch komplexe Abhängigkeiten in den Eingabedaten abzubilden und für unterschiedliche Aufgaben nutzbar zu machen [HSW+89].

Um die Funktionsweise und die Stärken dieser Modelle besser zu verstehen, ist es notwendig, ihre grundlegenden Architekturen zu betrachten. Im Folgenden werden daher verschiedene Ausprägungen neuronaler Netze vorgestellt.

2.8.1 Künstliches neuronales Netz (KNN)

Künstliche Neuronale Netze (KNNs) sind rechnergestützte Modelle, die von der Funktionsweise des menschlichen Gehirns inspiriert sind. Sie bestehen aus einer Vielzahl künstlicher Neuronen, die in mehreren Schichten organisiert sind und über gewichtete Verbindungen miteinander kommunizieren. Typischerweise setzt sich ein KNN, wie in Abbildung 2.7 dargestellt, aus einer Eingabeschicht, einer oder mehreren verborgenen Schichten (Hidden Layers) und einer Ausgabeschicht zusammen [Wut24].

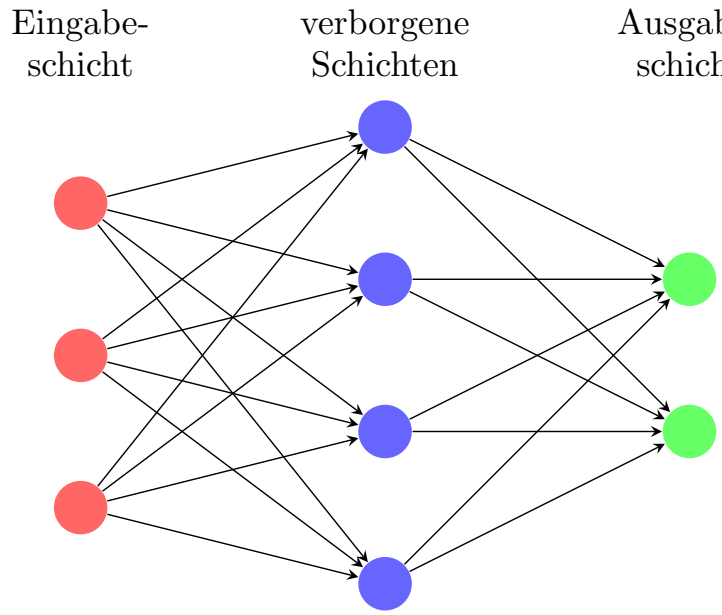

Abbildung 2.7: Schematische Darstellung eines künstlichen neuronalen Netzes. Quelle: [Wut24]

Die Verarbeitung von Informationen erfolgt schichtweise. Die Eingabedaten werden zunächst der Eingabeschicht zugeführt und dann schrittweise in den nachfolgenden Schichten transformiert. Dabei berechnet jedes Neuron eine gewichtete Summe seiner Eingaben und überführt diese mithilfe einer Aktivierungsfunktion in einen Ausgabewert. Dieser wird an die nächste Schicht weitergegeben [Wut24].

Beim Training eines neuronalen Netzes werden die Verbindungen zwischen den Neuronen schrittweise angepasst, um die Übereinstimmung zwischen den vorhergesagten und den tatsächlichen Ergebnissen zu optimieren. Zu diesem Zweck kommt das sogenannte Backpropagation-Verfahren in Kombination mit dem Gradientenabstieg zum Einsatz. Beim Backpropagation-Verfahren wird der am Ende des Netzes entstehende Fehler rückwärts durch alle Schichten weitergegeben. So lässt sich ermitteln, welche Verbindungen den Fehler am stärksten beeinflussen. Diese Verbindungen werden dann gezielt angepasst. Der Gradientenabstieg legt fest, in welche Richtung und wie stark die Gewichte verändert werden müssen, um den Fehler zu verringern. Dabei spielt die Lernrate eine wichtige Rolle: Ist sie zu groß, wird das Training instabil, ist sie zu klein, dauert es sehr lange, bis das Netz lernt. Durch diesen wiederholten Prozess aus Vorwärts- und Rückwärtsrechnung lernt das Netzwerk Schritt für Schritt, auch komplizierte Zusammenhänge in den Daten zu erkennen, um beispielsweise Dinge zu klassifizieren, Werte vorherzusagen oder Muster zu erkennen [Wut24].

Der Schichtaufbau ermöglicht zudem die Verarbeitung unterschiedlicher Datentypen. Eingabeschichten nehmen Rohdaten wie Text, Bilder oder Zeitreihen auf, während tiefere Schichten zunehmend abstraktere Merkmale extrahieren. Die anfänglich zufällig gesetzten Gewichtungen werden über viele Iterationen hinweg optimiert, wobei Validierungsda-

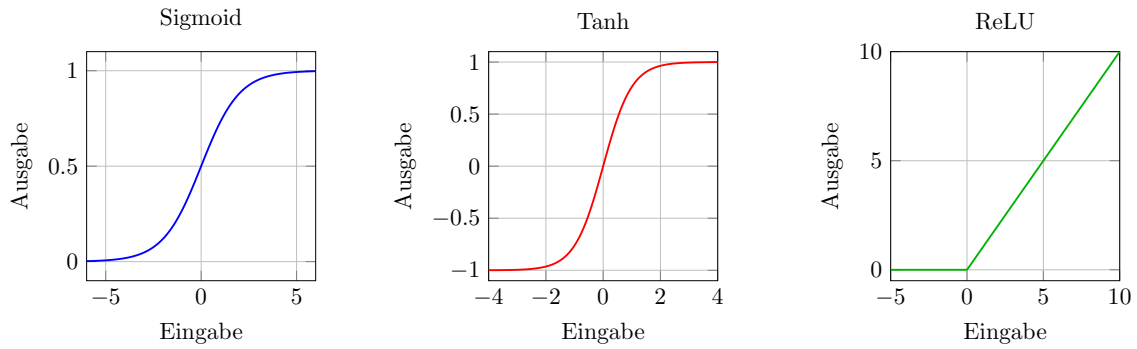

Abbildung 2.8: Funktionsdiagramm der Aktivierungsfunktionen von Sigmoid, Tanh und ReLU. Quelle: frei nach [Sha25]

ten helfen, eine Überanpassung zu vermeiden. Dabei bestimmen Struktur und Größe des Netzwerks sowohl die Fähigkeit zur Merkmalsextraktion als auch den Trainingsaufwand [Wut24].

Im Folgenden werden zentrale Bausteine neuronaler Netze näher erläutert.

2.8.1.1 Aktivierungsfunktion

Aktivierungsfunktionen entscheiden darüber, ob ein Neuron „aktiviert“ wird, und beeinflussen maßgeblich die Fähigkeit des Netzes, nichtlineare Zusammenhänge zu modellieren. Sie bestimmen, wie stark ein Neuron auf bestimmte Eingangssignale reagiert, und sind somit von entscheidender Bedeutung für die Lernfähigkeit des Netzes. Zu den gebräuchlichsten Funktionen zählen die in Abbildung 2.8 dargestellte Sigmoid-Funktion, die Tanh-Funktion und die Rectified Linear Unit (ReLU) [Sha25]:

- **Sigmoid-Funktion:** Die Sigmoid-Funktion hat die Form $\sigma(x) = \frac{1}{1+e^{-x}}$ und bildet jeden reellen Wert auf den Bereich $(0, 1)$ ab. Dadurch eignet sie sich gut für Ausgaben, die als Wahrscheinlichkeiten interpretiert werden sollen, insbesondere in binären Klassifikationsaufgaben. Ein Nachteil der Sigmoid-Funktion ist jedoch, dass sie bei großen oder kleinen Eingangswerten zu sehr flachen Gradienten führt (Gradientenproblem), wodurch sich das Lernen in tiefen Netzwerken verlangsamen kann [Sha25].
- **Tanh-Funktion:** Die Tanh-Funktion (hyperbolischer Tangens) ist definiert als $tanh(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$ und transformiert Eingaben in den Bereich $(-1, 1)$. Im Vergleich zur Sigmoid-Funktion ist sie zentriert, was oft zu schnellerer und stabilerer Konvergenz führt, da die Mittelwerte der Aktivierungen näher bei null liegen. Dennoch leidet auch die Tanh-Funktion unter dem Vanishing-Gradient-Problem bei extremen Eingabewerten [Sha25].
- **ReLU-Funktion** Die Rectified Linear Unit (ReLU)-Funktion ist definiert als $f(x) = \max(0, x)$ und hat sich in vielen modernen neuronalen Netzen als Standard durchgesetzt. Sie ist einfach, effizient und reduziert das Problem verschwindender Gradienten,

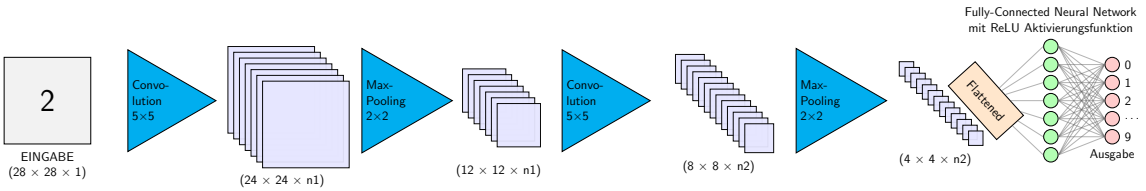

Abbildung 2.9: Schematische Darstellung eines Convolutional Neural Networks. Quelle: frei nach [Red+21]

da sie bei positiven Werten eine konstante Ableitung von 1 liefert. Ein Nachteil der ReLU ist jedoch das sogenannte „Dying ReLU"-Problem: Neuronen, die dauerhaft negative Eingabewerte erhalten, geben keine Aktivierung mehr weiter und „sterben" während des Trainings ab [Sha25].

2.8.1.2 Dropout

Dropout ist eine regulierende Technik, die während des Trainings zufällig ausgewählte Neuronen deaktiviert. Dadurch wird verhindert, dass sich das Netzwerk zu stark an die Trainingsdaten anpasst (Overfitting), und die Generalisierungsfähigkeit auf neue, unbekannte Daten wird verbessert. Dropout wirkt, indem es während des Trainings gewissermaßen viele leicht unterschiedliche Versionen des Netzwerks erzeugt, die gemeinsam trainiert werden, was die Stabilität und Robustheit des Modells erhöht [Sri+14].

2.8.1.3 Optimizer

Optimierer sind Algorithmen, die die Gewichte des neuronalen Netzes während des Trainings so anpassen, dass der Fehler minimiert wird. Neben dem klassischen Gradientenabstieg (Gradient Descent) existieren zahlreiche Varianten wie Adam [KB17], RMSprop [HSS12] oder Adagrad [DHS11], die durch adaptive Lernraten und zusätzliche Informationen wie Bewegungsrichtung (Momentum) die Konvergenz beschleunigen und verbessern [Rud17].

2.8.2 Convolutional Neural Network (CNN)

Das Convolutional Neural Network (CNN) wurde wegen seiner Leistung bei Aufgaben der Bilderkennung populär. Ein möglicher Aufbau eines CNNs ist in Abbildung 2.9 dargestellt. Im Gegensatz zu herkömmlichen neuronalen Netzen werden die Eingabedaten in einem CNN abschnittsweise analysiert. Dabei wird ein kleiner Bereich, das sogenannte Schiebefenster oder Filter, über das gesamte Bild bewegt. In jedem Schritt untersucht dieser Filter nur einen kleinen Teil des Bildes und erkennt darin einfache Muster, zum Beispiel Kanten oder Farbverläufe.

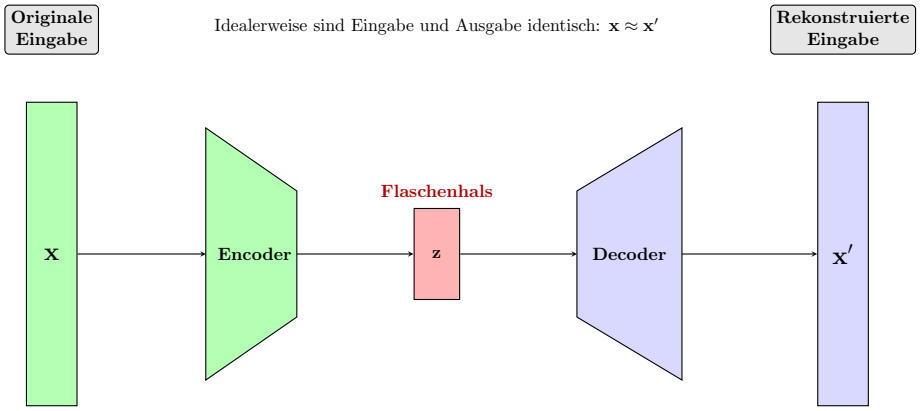

Abbildung 2.10: Schematische Darstellung eines Autoencoders. Quelle: frei nach [Wen18]

Die Gewichtungen des Filters bleiben bei dieser Bewegung gleich. Dadurch kann das Netz dieselben Merkmale überall im Bild erkennen, unabhängig davon, wo sie auftreten. Viele solcher Filter arbeiten gleichzeitig in verschiedenen Kanälen, um unterschiedliche Merkmale zu erfassen.

Diese Vorgehensweise ermöglicht es dem Netzwerk, selbstständig aussagekräftige Merkmale aus den Rohdaten zu lernen. Der früher notwendige Schritt, Merkmale manuell zu definieren, entfällt somit.

Obwohl CNNs hauptsächlich in der Bildverarbeitung eingesetzt werden, lässt sich das Prinzip auch auf andere strukturierte Daten anwenden, beispielsweise auf Texte, Musik oder Binärdateien [KSH12].

2.8.3 Autoencoder (AE)

Autoencoder (AE) sind künstliche neuronale Netzwerke, die darauf ausgelegt sind, eine Reduktion der Dimensionalität für ein gegebenes Eingabedatum zu erreichen, während so viel Information wie möglich erhalten bleibt. Dies ist in Abbildung 2.10 dargestellt. Während des Trainings versucht der Autoencoder, die Eingabedaten am Ausgang möglichst genau zu rekonstruieren. Die Verlustfunktion misst den Unterschied zwischen den ursprünglichen Eingabedaten und der vom Netzwerk erzeugten Rekonstruktion. Je kleiner dieser Unterschied ausfällt, desto besser hat das Netzwerk gelernt, die wichtigsten Merkmale der Daten zu erfassen und unwichtige Details zu verwerfen. Die Architektur selbst enthält einen Flaschenhals in einer mittleren Schicht, um das Netzwerk dazu zu zwingen, Daten zu komprimieren, aber rekonstruierbare Informationen zu bewahren. Die Größe der mittleren Schicht muss die komprimierte Größe balancieren und relevante Informationen in einer unbekannten Kodierung bewahren. Dies teilt das neuronale Netzwerk in zwei Komponenten, nämlich den Encoder- und den Decoder-Teil. Nachdem der AE trainiert wurde, wird das Decoderelement entfernt, sodass alle Eingabedaten nur noch in ihrer kodierten Form zurückgegeben werden [HS06].

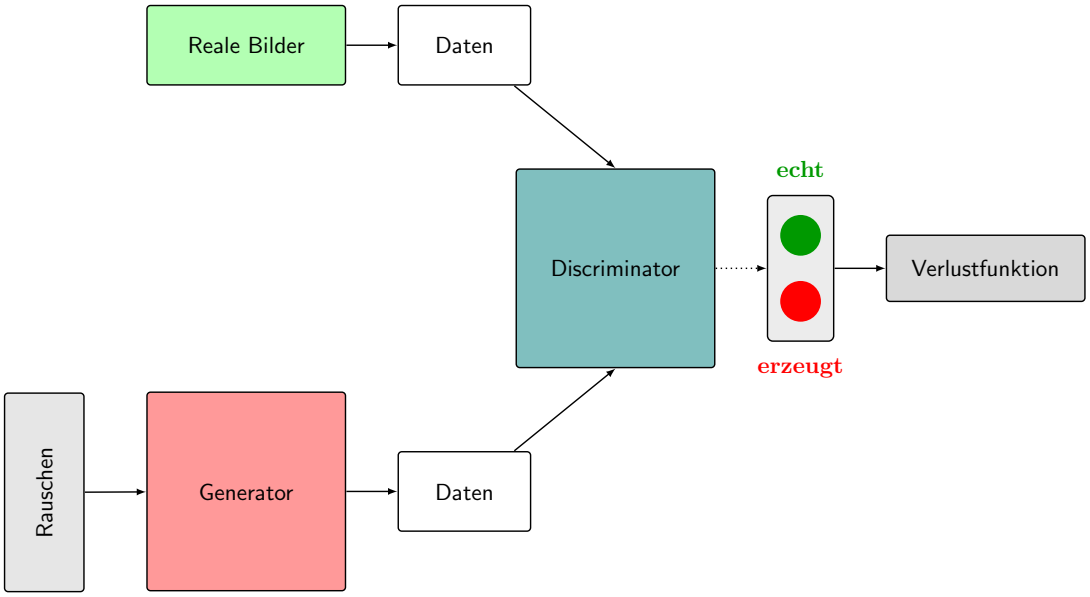

Abbildung 2.11: Schematische Darstellung eines Generative Adversarial Networks. Quelle: frei nach [Kin21]

2.8.4 Generative Adversarial Network (GAN)

Das Generative Adversarial Network (GAN) wurde entwickelt, um ein generatives Modell zu erzeugen, das die zugrunde liegende Verteilung der Trainingsdaten nachbildet. Wie in Abbildung 2.11 dargestellt, besteht die Architektur aus zwei miteinander konkurrierenden neuronalen Netzwerken.

Das erste Netzwerk, der Generator, erhält Zufallsrauschen als Eingabe und versucht, daraus ein Bild zu erzeugen, das den echten Beispielen aus dem Trainingsdatensatz möglichst ähnlich ist. Das zweite Netzwerk, der Diskriminator, bekommt sowohl ein echtes als auch ein vom Generator erzeugtes Bild und soll entscheiden, welches Bild echt und welches künstlich ist.

Die Rückmeldung des Diskriminators dient als Grundlage für die Fehlerkorrektur des Generators: Erkennt der Diskriminator ein Bild als „künstlich“, werden die Parameter des Generators so angepasst, dass dessen Ausgaben realistischer werden.

Durch dieses Zusammenspiel entsteht ein wettbewerbsorientierter Lernprozess, bei dem sich beide Netzwerke gegenseitig verbessern. Der Generator erzeugt immer überzeugendere Bilder und der Diskriminator erkennt immer feinere Unterschiede [Goo+14].

2.8.5 Long Short-Term Memory (LSTM)

Ein Recurrent Neural Network (RNN) ist eine Netzwerkarchitektur, bei der ein Teil des verborgenen Zustands an den nächsten Zeitschritt zurückgeführt wird. Dadurch kann das Netzwerk zeitliche Abhängigkeiten in den Daten erkennen.

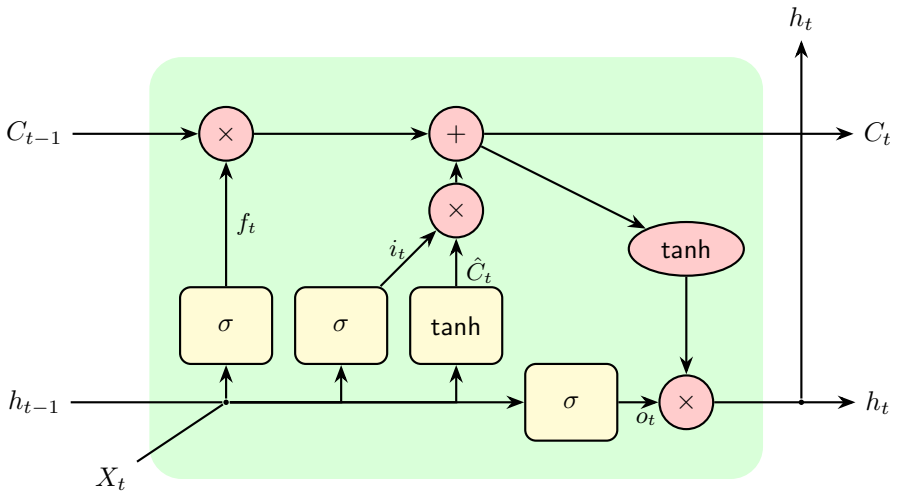

Abbildung 2.12: Schematische Darstellung einer Long Short-Term Memory Zelle. Quelle: frei nach [Che18]

Das Long Short-Term Memory (LSTM) ist eine spezielle Form des RNN, die besonders gut darin ist, langfristige Abhängigkeiten zu erfassen, und dabei typische Probleme wie das Verschwinden oder Explodieren von Gradienten vermeidet. Der Aufbau einer LSTM-Zelle ist in Abbildung 2.12 dargestellt.

Eine LSTM-Zelle besteht aus einem Zellzustand (C_t), der Informationen über viele Zeitschritte wie ein „Speicherband“ transportiert, sowie drei Torstrukturen, die den Informationsfluss steuern:

- **Vergessenstor (f_t):** entscheidet, welche Informationen aus dem Zellzustand gelöscht werden,
- **Eingangstor (i_t):** bestimmt, welche neuen Informationen aufgenommen werden,
- **Ausgangstor (o_t):** legt fest, welche Informationen als Ausgabe oder für den nächsten Zeitschritt weitergegeben werden.

Mithilfe dieser Tore kann der Zellzustand gezielt aktualisiert, gelöscht oder beibehalten werden. Dadurch ist das Netzwerk in der Lage, relevante Informationen über lange Zeiträume zu speichern.

Für die Arbeit mit Textdaten (sequentielle Buchstaben) ist es üblich, ein Wörterbuch oder Alphabet zusammen mit einer geeigneten Einbettung zu verwenden, um die Informationen in einen mittelgroßen Vektor für jedes Wort oder Zeichen zu kondensieren. Der Text wird in eine Matrix der Dimensionen $\text{Länge}_{\text{Text}} \times \text{Länge}_{\text{Embedding}}$ umgewandelt [HS97].

2.8.6 Self-Organizing Map (SOM)

Die Self-Organizing Map (SOM) ist eine unüberwachte neuronale Architektur, deren Ausgabeneuronen typischerweise ein- oder zweidimensional angeordnet sind. Dadurch ist eine

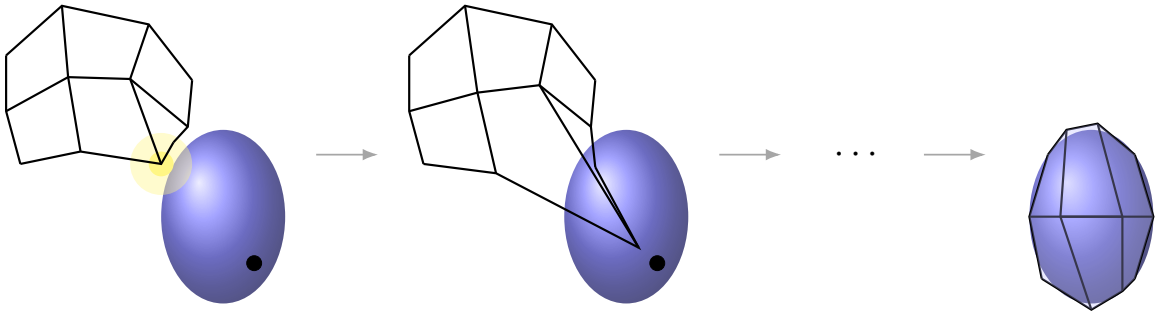

Abbildung 2.13: Schematische Darstellung einer Self-Organizing Map. Quelle: frei nach [Sto10]

topologische Projektion der Eingabedaten möglich. Dabei werden ähnliche Datenpunkte auf benachbarte Regionen der Karte abgebildet, sodass Clusterstrukturen sichtbar werden. Jedes Ausgabeneuron ist durch einen Gewichtsvektor im Eingaberaum charakterisiert. Wie in Abbildung 2.13 dargestellt, passt sich die SOM während des Trainings iterativ an die Datenverteilung an, indem die Gewichtsvektoren verschoben und geglättet werden. Für eine gegebene Eingabe wird das sogenannte Best Matching Unit (BMU) als Gewinnerneuron bestimmt. Es kann als Repräsentant der Eingabe sowie für deren Indexierung oder Klassifikation genutzt werden [Koh82].

2.8.7 Large Language Model (LLM)

Ein LLM wird auf großen Mengen von Textdaten trainiert, um Muster in natürlicher Sprache zu erkennen und kohärente Ausgaben zu erzeugen. Die zugrunde liegende Architektur basiert auf dem Transformer-Modell und nutzt den Self-Attention-Mechanismus. Ein Transformer ist eine speziell für die Verarbeitung von Sequenzen, wie etwa Text, entwickelte Netzwerkarchitektur. Im Unterschied zu klassischen RNNs verarbeitet ein Transformer alle Positionen einer Sequenz gleichzeitig. Dadurch kann er Zusammenhänge zwischen weit auseinanderliegenden Wörtern effizienter erfassen. Der zentrale Mechanismus im Transformer ist die sogenannte Self-Attention: Für jedes Wort wird berechnet, in welchem Maß es mit allen anderen Wörtern der Sequenz in Beziehung steht. So kann das Modell die Bedeutung eines Wortes im Kontext des gesamten Satzes oder Textes berücksichtigen und semantische Zusammenhänge effektiv erfassen. Zusätzlich können Beziehungen zwischen Wörtern auch über lange Distanzen hinweg erfasst und deren Bedeutung je nach Kontext unterschiedlich gewichtet werden [Vas+17]. Je nach Modelltyp wird ein Encoder (z. B. BERT [Dev+19]), ein Decoder (z. B. GPT [Rad+19]) oder eine Kombination aus Encoder und Decoder (z. B. T5 [Raf+20]) eingesetzt. Als Ergebnis können LLMs das nächste Wort oder die nächste Phrase mit hoher Wahrscheinlichkeit vorhersagen und konsistenten Text erzeugen. Zu den typischen Anwendungsbereichen zählen unter anderem die Beantwortung von Fragen, die Textzusammenfassung sowie die Generierung strukturierter Inhalte [Str25].

2.8.7.1 Modell-Anpassung

Die ML-Community stellt eine Vielzahl vortrainierter LLMs frei zur Verfügung. Diese umfassen häufig mehrere natürliche Sprachen (z. B. Englisch, Französisch, Deutsch) sowie Programmiersprachen (z. B. Python, JavaScript). Plattformen wie Hugging Face[1] ermöglichen einen direkten Zugriff auf diese Modelle. Obwohl sie sofort eingesetzt werden können, ist für spezifische Anwendungsfälle oft eine weitere Anpassung erforderlich, etwa durch *Fine-Tuning* oder *Prompt-Tuning* [Str25].

2.8.7.2 Fine-Tuning

Fine-Tuning beschreibt die gezielte Anpassung der Modellparameter eines vortrainierten LLM, um es auf eine konkrete Aufgabe oder ein spezielles Datenset abzustimmen. Dies kann beispielsweise die Verbesserung von Fähigkeiten in der Sentiment-Analyse, der Fragebeantwortung oder der Erzeugung strukturierter Daten (z. B. XML) umfassen. Der Vorgang erfordert ein zusätzliches Training auf einem zielgerichteten Datensatz, um die Parameter optimal anzupassen. Aufgrund der teils Milliarden umfassenden Parameter aktueller Modelle ist dieser Prozess rechenintensiv, benötigt spezialisierte Ressourcen wie Hochleistungs-Graphics Processing Units (GPUs) oder Tensor Processing Units (TPUs) und kann über längere Zeiträume andauern [Xu+21].

2.8.7.3 Prompt-Tuning

Da die umfassende Vorschulung von LLMs bereits eine breite Sprachkompetenz vermittelt, ist es oft ineffizient, alle Modellparameter neu anzupassen. Eine ressourcenschonendere Methode ist Prompt-Tuning, eine spezielle Parameter-Efficient Fine-Tuning (PEFT)-Technik [Pat24]. Dabei werden die ursprünglichen Gewichte des Modells eingefroren, während nur eine kleine Anzahl zusätzlicher Prompt-Parameter trainiert wird. Mithilfe gezielter Eingabe-Ausgabepaare wird das Modell so optimiert, dass es bestimmte Aufgaben zuverlässiger bearbeitet. Dieser Ansatz nutzt das in der Vorschulung erworbene Wissen. Er reduziert den Anpassungsaufwand erheblich und eignet sich daher besonders für spezialisierte Anwendungen bei komplexen Modellen [LAC21; Yon+23].

2.8.7.4 Modell-Inference

In der Inference-Phase nutzt ein LLM sein erlerntes Sprachmodell, um auf eine Eingabe (Prompt) Antworten und Ausgaben zu generieren. Dabei werden probabilistische Vorhersagen genutzt, um die wahrscheinlichste Fortsetzung des Textes zu erzeugen [Pla20; YG23]. Dies ermöglicht es dem Modell, auch auf unbekannte Daten zu reagieren und kohärente Texte, Code oder Antworten zu erstellen. Die Effizienz und Genauigkeit des Inference bestimmen maßgeblich den praktischen Nutzen und die Einsatzmöglichkeiten von

[1] https://huggingface.co/

LLMs. Fortschritte in Architektur, Trainingsmethoden und Rechentechnologien verbessern kontinuierlich die Leistungsfähigkeit dieser Modelle und erweitern ihre industrielle Anwendbarkeit [Str25].

Der Einsatz großer Sprachmodelle veranschaulicht die Funktionsweise probabilistischer Verfahren zur Mustererkennung in Textdaten. Doch auch in nichttextuellen Daten müssen Strukturen erkannt und abgegrenzt werden. Während LLMs auf Wahrscheinlichkeiten im Sprachraum basieren, nutzen andere Verfahren einen dichtebasierten Ansatz. Im Folgenden wird ein Algorithmus zur Identifikation beliebig geformter Cluster und zur robusten Behandlung von Ausreißern vorgestellt.

2.9 Density-Based Spatial Clustering of Applications with Noise (DBSCAN)

Density-Based Spatial Clustering of Applications with Noise (DBSCAN) ist ein dichtebasierter Clustering-Algorithmus, der Datenpunkte basierend auf deren lokaler Punktdichte in Cluster gruppiert und Ausreißer als Rauschen klassifiziert. Durch die Verwendung eines Nachbarschaftsparameters ε und einer minimalen Punktanzahl MinPts erkennt der Algorithmus zusammenhängende Regionen hoher Dichte, ohne dass die Anzahl der Cluster im Voraus bekannt sein muss. Dies ermöglicht die Identifikation von Clustern beliebiger Form und macht DBSCAN besonders robust gegenüber Ausreißern [Est+96].

Eine Weiterentwicklung stellt Hierarchical Density-Based Spatial Clustering of Applications with Noise (HDBSCAN) dar, ein Algorithmus, der Cluster mit variabler Dichte identifizieren kann. Im Gegensatz zu DBSCAN, das einen globalen Dichteparameter verwendet, analysiert HDBSCAN die Dichte-Struktur über verschiedene Dichtegrade hinweg und erstellt eine Hierarchie von Clustern. Anschließend werden die stabilsten Cluster ausgewählt, während Punkte, die keinem stabilen Cluster zugeordnet werden können, als Ausreißer betrachtet werden. Damit entfällt die Notwendigkeit, einen festen ε-Parameter zu definieren, und HDBSCAN ist besonders anpassungsfähig an komplexe Datenstrukturen [CMS13].

3 ESP32 Code-Injektion bei unverändertem Kontrollfluss mittels Binary Rewriting

Der Inhalt dieses Kapitels basiert auf einer gemeinsamen Veröffentlichung mit Benjamin Plach, Maximilian Müller, Roland Gröll, Martin Dukek und Ingmar Baumgart. Teile der präsentierten Ergebnisse wurden bereits in der unten aufgeführten Publikation veröffentlicht. Es wird ein neuartiges Binary-Rewriting-Framework für die ESP32-Plattform vorgestellt, das es ermöglicht, zusätzlichen Code in bestehende Firmware einzufügen, ohne deren ursprüngliche Funktionalität zu beeinträchtigen.

- Benjamin Plach, **Matthias Börsig**, Maximilian Müller, Roland Gröll, Martin Dukek und Ingmar Baumgart. „Binary-Level Code Injection for Automated Tool Support on the ESP32 Platform“. In: Secure IT Systems: 29th Nordic Conference, NordSec 2024 Karlstad, Sweden, November 6–7, 2024 Proceedings. Hrsg. von Leonardo Horn Iwaya, Liina Kamm, Leonardo Martucci und Tobias Pulls. Bd. 15396. Lecture Notes in Computer Science. Karlstad, Sweden: Springer-Verlag, Jan. 2025, S. 121–138. isbn: 978-3-031-79006-5. DOI: 10.1007/978-3-031-79007-2_7 [Pla+25].

3.1 Einleitung

In diesem Kapitel wird ein Ansatz zur Instrumentierung von ESP32-Firmware auf Binärebene vorgestellt. Das Ziel besteht darin, den Binärcode so zu erweitern, dass zusätzliche Analyseinformationen, beispielsweise über Speicherzugriffe, Funktionsaufrufe oder den Kontrollfluss, während der Laufzeit gewonnen werden können, ohne das ursprüngliche Verhalten der Firmware zu verändern. Auf diese Weise wird eine Grundlage geschaffen, um Fuzzing und Schwachstellenerkennung auch ohne Zugriff auf den Quellcode effektiv durchzuführen.

Eine zentrale Herausforderung bestand darin, Techniken des Binary Rewriting, die bislang ausschließlich für Complex Instruction Set Computer (CISC)-basierte x86-Architekturen verfügbar waren, auf die RISC-basierte Xtensa-ISA des ESP32 zu übertragen. Hierzu wurden bestehende Patching-Strategien analysiert, angepasst und erweitert. Besondere Sorgfalt war erforderlich, um Patches in spezifischen Bereichen, wie dem `.flash.text`-Segment (Speicherbereich im Flash, in dem der ausführbare Programmcode abgelegt ist), einzufügen, ohne den ursprünglichen Kontrollfluss der Firmware zu verändern.

Die wesentlichen Beiträge lassen sich wie folgt zusammenfassen:

- **Anpassung von Binary-Rewriting-Techniken an die Xtensa-Architektur:** Übertragung und Erweiterung bestehender Patching-Strategien von x86 auf die RISC-basierte Xtensa-ISA, um präzise Instrumentierung in ESP32-Firmware zu ermöglichen.
- **Implementierung eines eigenen Assemblers:** Entwicklung eines minimalen, erweiterbaren Assemblers zur Unterstützung der für die Instrumentierung notwendigen Xtensa-Befehle, einschließlich relativer Sprungbefehle und Adressberechnungen.
- **Entwicklung eines statischen Binary-Rewriting-Frameworks:** Aufbau eines Frameworks, das Code-Injektionen in ESP32-Firmware erlaubt, ohne die ursprüngliche Funktionalität und den Kontrollfluss zu beeinträchtigen.
- **Proof of Concept (PoC) für Fuzzing-Instrumentierung:** Implementierung und Evaluierung eines Tools, das Funktionsabdeckungsinformationen in modifizierte ESP32-Binaries integriert und somit die Eignung des Frameworks für sicherheitsrelevante Analysen belegt.

Damit trägt dieses Kapitel wesentlich zum Gesamtziel dieser Dissertation bei, eine modulare Grundlage für automatisierte Sicherheitstests von IoT-Geräten zu schaffen. Dieser Ansatz stellt die notwendige Infrastruktur bereit, um Laufzeitinformationen effizient zu erfassen und gezielt für nachgelagerte Fuzzing-Prozesse nutzbar zu machen.

3.2 Stand der Technik

In diesem Abschnitt werden Arbeiten vorgestellt, die unmittelbar als Grundlage für die Umsetzung dienten. Das Ziel besteht darin, den aktuellen Stand der Technik einzuordnen, die konzeptionelle Verwandtschaft zu verdeutlichen und den spezifischen Mehrwert des vorgestellten Ansatzes klar herauszustellen.

Duck et al. [DGR20] entwickelten mit *E9Patch* einen statischen Binary Rewriter, der auf einer Trampolin-Rewriting-Technik basiert und den langen relativen Sprung-Opcode `E9` der x86-Architektur nutzt. Dieser Ansatz hat den Vorteil, dass er nur minimale Annahmen über die zu modifizierende Binärdatei trifft und keinerlei Abhängigkeiten von Quellsprache, Compiler, Debugging-Informationen oder einer vollständigen Kontrollflussanalyse benötigt. Gleichzeitig schließt die Methode Binärdateien aus, die selbstmodifizierenden Code enthalten oder sich überschneidende Instruktionen nutzen.

Die Arbeit von Duck et al. basiert selbst zu Teilen auf der Idee von Chamith et al. [Cha+17] und der von ihnen vorgestellten Technik des „Instruction Punning“. Dabei werden bestehende Maschinenbefehle so überschrieben, dass sie gleichzeitig ausführbar bleiben aber auch als Anker für neue Sprungbefehle dienen. Dies ermöglicht das Einfügen von Analysecode, ohne den Programmfluss zu verändern.

Der vorgestellte Ansatz basiert auf dieser Grundidee, geht jedoch in zwei wesentlichen Punkten darüber hinaus. Zum einen wurde die Technik nicht für x86-64-Linux-Binärdateien eingesetzt, sondern auf die Xtensa-Architektur des ESP32 übertragen. Da diese mit

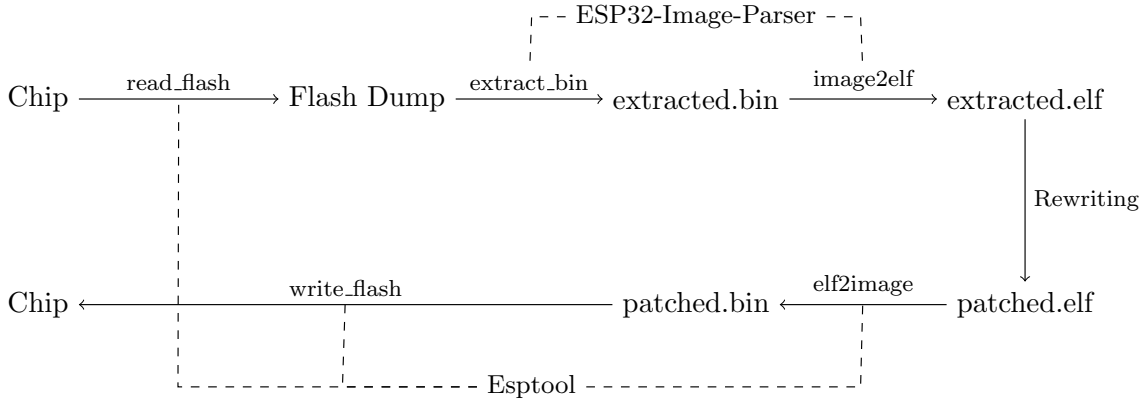

Abbildung 3.1: Prozess des Binary Recovery, des Rewritings und erneuten Flashens

einem völlig anderen Befehlssatz und abweichenden Speicherstrukturen arbeitet, war teilweise eine methodische Neuentwicklung erforderlich. Zum anderen liegt der Fokus nicht auf generischen Linux-Binärdateien, sondern auf proprietären IoT-Firmware-Images, die zusätzliche Herausforderungen wie stark begrenzte Speicherressourcen mit sich bringen.

3.3 Design

Der in diesem Kapitel vorgestellte Ansatz wird im Folgenden als ESP32 Binary Rewriting (EBR) bezeichnet, um eine konsistente Referenzierung zu ermöglichen.

Das Design besteht aus mehreren aufeinanderfolgenden Schritten: zunächst wird die originale Binärdatei extrahiert. Anschließend erfolgt die Modifikation der Binärdatei durch den gezielten Einsatz verschiedener Patching-Techniken. Abschließend wird die modifizierte Binärdatei wieder auf das ESP32-Gerät geflasht.

3.3.1 Binary Recovery

Das Binary Recovery beginnt mit dem Extrahieren eines vollständigen Flash-Dumps des ESP32-Geräts. Danach wird die Partitions-Tabelle wiederhergestellt und ihre Informationen verwendet, um die Anwendungs-Binärdatei zu identifizieren. Eine wichtige Umwandlung in diesem Prozess ist das Konvertieren der wiederhergestellten Anwendungs-Binärdatei in das ELF-Dateiformat, was die anschließende Analyse und Modifikation vereinfacht. Die allgemeine Idee dieses Extraktions- und Flash-Prozesses ist in Abbildung 3.1 dargestellt.

3.3.2 Rewriter

Der Rewriter besteht aus mehreren *Patching-Taktiken*, die jeweils spezifische Änderungsfälle im Code behandeln, sowie einer übergeordneten Strategie, die das gesamte Binärfile

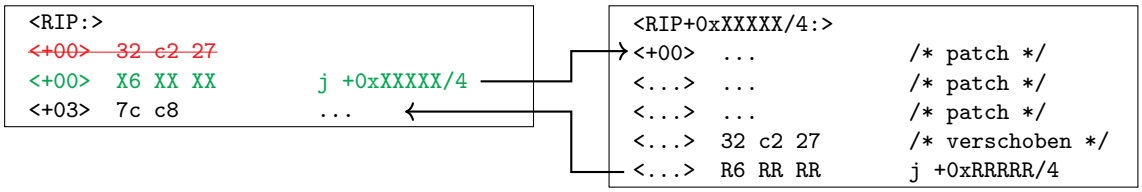

Abbildung 3.2: Anwendung der Jump-Taktik auf Xtensa

berücksichtigt. Die einzelnen Taktiken werden in einer festgelegten Reihenfolge angewendet. Schlägt eine Methode fehl, wird automatisch die nächste ausprobiert. Ein Beispiel: Kann die *Jump-Taktik* an einer Stelle keinen Patch setzen, wird als Nächstes die *Punned-Jump-Taktik* verwendet.

3.3.2.1 Patching-Taktiken

Die Rewriter-Komponente ist dafür verantwortlich, die wiederhergestellte ELF-Binärdatei zu modifizieren, um die notwendige Instrumentierung einzufügen. Hierfür werden verschiedene Patching-Taktiken angewendet, um dies zu erreichen, ohne den ursprünglichen Programmablauf zu verändern.

Jump-Taktik Die Jump-Taktik beinhaltet die Umleitung des Kontrollflusses vom ursprünglichen Code zur neu eingefügten Instrumentierung und dann zurück zum ursprünglichen Code. Dies wird erreicht, indem Sprungbefehle anstelle der ursprünglichen Instruktion eingefügt werden, die in den Trampolin-Code verschoben wird. Abbildung 3.2 zeigt die Anwendung der Jump-Taktik auf die Xtensa-Architektur, wobei die folgende Syntax verwendet wird: Die ursprüngliche Instruktion (rot) wird entfernt und durch den Sprungbefehl (grün) ersetzt, wobei `X` einen beliebigen wählbaren Wert darstellt. Der Opcode des Sprungbefehls hat sechs Bits, wodurch 18 Bits für den relativen Offset zum Zielort übrig bleiben. Diese 18 Bits sind als fünf `X`s dargestellt, wobei jedes ein halbes Byte in hexadezimaler Kodierung repräsentiert, aber mit `/4` (logische Rechtsverschiebung um 2 Bits) versehen wird, um anzugeben, dass die zwei Most Significant Bits (MSBs) abgeschnitten werden, da sie Teil des Opcodes sind. Dieser Sprungbefehl verweist nun auf den Beginn des Trampolins (rechte Seite in der Abbildung), an dem der Patch eingefügt wird, einschließlich der ursprünglichen Instruktion am Ende. Die letzte Instruktion im Trampolin zeigt auf die erste Instruktion nach dem Sprung.

Punned Jump-Taktik Die Punned Jump-Taktik ist eine Variante der Jump-Taktik, die verwendet wird, wenn die Zielinstruktion, die durch den Sprungbefehl ersetzt werden soll, eine kurze 16-Bit-Instruktion und keine 24-Bit-Instruktion ist. In solchen Fällen kann das erste Byte der folgenden Instruktion in die aktuelle Instruktion integriert werden, eine Technik, die als Instruction Punning [Cha+17] bekannt ist. Abbildung 3.3 zeigt die Punned

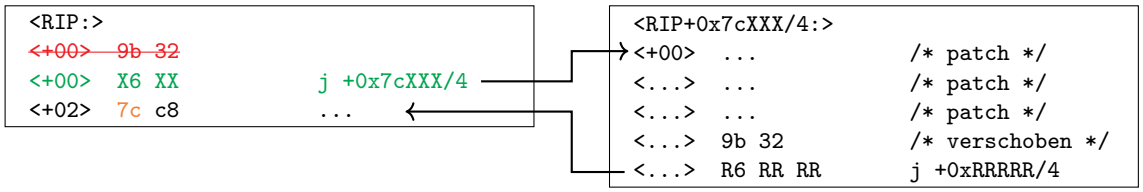

Abbildung 3.3: Anwendung der Punned Jump-Taktik auf Xtensa

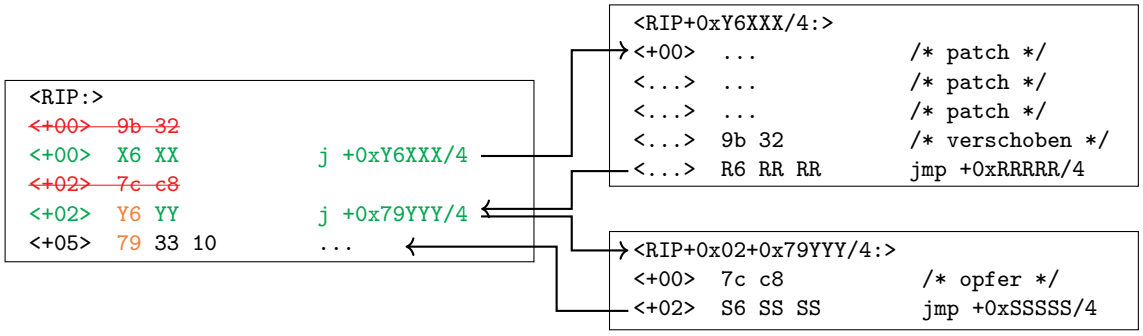

Abbildung 3.4: Anwendung der Successor Eviction-Taktik auf Xtensa

Jump-Taktik. Zusätzlich zur obigen Erklärung gibt es nun das *gepunte* Byte der folgenden Instruktion (orange), das nicht geändert werden kann und den Bereich des Sprungbefehls einschränkt. Je nach Programm kann es schwieriger sein, freien Speicherplatz für das Trampolin zu finden. Diese Einschränkung wird bewusst in Kauf genommen, um neue Möglichkeiten zu eröffnen: So ist es möglich, den Sprung in engen Bereichen einzufügen und den Patch in Situationen zu verwenden, die sonst unmöglich wären.

Successor Eviction-Taktik Wenn die Punned Jump-Taktik nicht erfolgreich ist, kann die Successor Eviction-Taktik verwendet werden, bei der auch die nächste Instruktion an einen anderen Codebereich verschoben wird (mit einer der oben genannten Taktiken). Wenn sowohl die ursprüngliche Instruktion als auch ihre Nachfolger verschoben werden, gibt es zusätzliche Optionen, um ungenutzte Codepositionen zu finden. Die verschobenen Instruktionen werden an eine neue Adresse innerhalb der Binärdatei verlegt, und der Kontrollfluss wird angepasst, um sicherzustellen, dass das Programm weiterhin korrekt ausgeführt wird. Abbildung 3.4 zeigt diese Taktik. Im Vergleich zur letzten Patching-Taktik gibt es nun zwei Ersetzungen, die für Situationen vorgesehen sind, in denen die Instruction-Punning-Taktik keinen geeigneten Bereich für den Trampolin-Code findet.

Neighbor Eviction-Taktik Die Neighbor Eviction-Taktik ist eine mögliche Option, falls die Successor Eviction-Taktik fehlschlägt und ist ihr ähnlich, aber sie verschiebt eine Instruk-

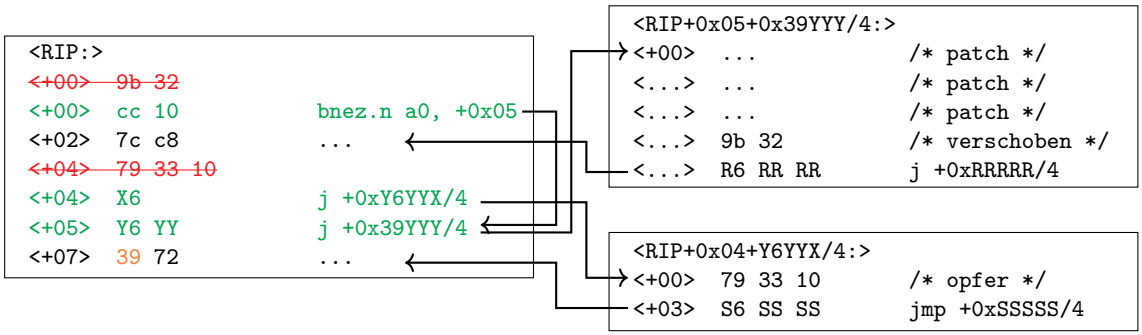

Abbildung 3.5: Anwendung der Neighbor Eviction-Taktik auf Xtensa

tion nach dem Patchpunkt. Dieser Ansatz, der in Abbildung 3.5 gezeigt wird, bietet noch mehr Flexibilität bei der Suche nach ungenutzten Codepositionen. Da die Xtensa ISA keinen kurzen relativen Sprung bietet, nutzt diese Taktik die `bnez.n`-Verzweigungsinstruktion, einen 16-Bit-Befehl, der einen relativen Sprung ausführt, wenn ein Register ungleich null ist. Das `a0`-Register enthält die Rücksprungadresse und sollte daher niemals null sein, was zu einem garantierten Sprung führt.

3.3.2.2 Patching-Strategie

Für jede Ersetzung werden so lange nacheinander verschiedene Taktiken versucht, bis eine erfolgreich angewendet werden kann. Wenn die letzte Taktik fehlschlägt, kann das Patch nicht angewendet werden.

Alle Ersetzungen werden in umgekehrter Reihenfolge angewendet, das heißt, das Patchen beginnt bei höheren Speicheradressen und setzt sich zu niedrigeren Adressen fort. Dieses Vorgehen verhindert, dass nachfolgende Bytes „blockiert“ werden, die ebenfalls verändert werden müssen. Zur Verdeutlichung sei das Beispiel der *Punned Jump*-Taktik in Abbildung 3.3 betrachtet: Dort müsste auch die zweite Instruktion verschoben werden. Würde man hingegen in normaler Reihenfolge patchen und zuerst die erste Instruktion ändern, wäre das erste Byte der zweiten Instruktion bereits überschrieben und somit für ein weiteres Patch nicht mehr zugänglich. Durch die umgekehrte Reihenfolge wird dieses Problem vermieden, da die zweite Instruktion zuerst gepatcht wird und das Instruction Punning der ersten Instruktion anschließend weiterhin korrekt mit dem neuen Byte `Y6` durchgeführt werden kann.

3.4 Implementierung

Im Folgenden wird die Implementierung des im vorherigen Abschnitt besprochenen Designs vorgestellt. Die Implementierung ist speziell auf das ESP32-WROOM-32-Modell ausgelegt, welches eine weit verbreitete Version des ESP32 ist.

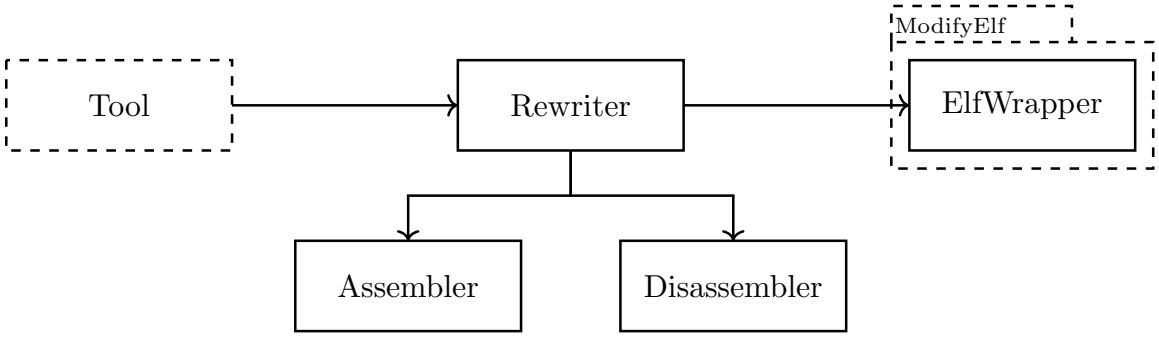

Abbildung 3.6: Beziehung der Hauptkomponenten des ESP32 Binary Rewriting Tools

3.4.1 Binary Recovery

Zunächst wird ein vollständiger Flash-Dump des Ziel-ESP32-Geräts mithilfe von Esptool[1] extrahiert. Anschließend wird das ESP32-Image-Parser-Tool[2] verwendet, um das Anwendungs-Image aus dem Flash-Dump zu lokalisieren und in das ELF-Format zu transformieren. Da dieses Tool veraltet ist, waren mehrere Fehlerkorrekturen erforderlich, um eine funktionsfähige Analyseumgebung bereitzustellen.

3.4.2 Rewriter

Das Rewriting-Tool ist der Hauptbeitrag und wird verwendet, um die wiederhergestellte Binärdatei zu modifizieren und die notwendige Instrumentierung für Aufgaben wie Fuzzing einzufügen. Abbildung 3.6 zeigt die Beziehung zu den anderen Komponenten.

Der Rewriter ist so strukturiert, dass er die zukünftige Entwicklung von Tools unterstützt. Es wurde ein neuer Assembler und ein Adapter für den Radare2-Disassembler implementiert. Die ModifyElf-Bibliothek ermöglicht die Manipulation von ELF-Binärdateien. Sie kapselt die Komplexität des ELF-Formats und bietet sowohl eine Low-Level- als auch eine High-Level-Schnittstelle. Die Low-Level-Schnittstelle wird von der `ElfRaw`-Klasse bereitgestellt, die detaillierte und präzise Modifikationen ermöglicht, während die `ElfWrapper`-Klasse eine abstraktere Schnittstelle für eine einfachere Handhabung bietet.

3.4.2.1 Patching-Taktiken

Jede Patching-Taktik nimmt den Patch-Standort und den Patch-Code als Eingabe und gibt zurück, ob der Patch-Versuch erfolgreich war oder nicht. Ein optionaler Parameter ermöglicht es, die Reihenfolge zu bestimmen, sodass die verschobene Anweisung vor oder

[1] `https://github.com/espressif/esptool/`
[2] `https://github.com/tenable/esp32_image_parser`

nach dem Patch-Code ausgeführt wird, wobei der Standardwert die Ausführung vor dem Patch-Code ist.

3.4.2.2 Patching-Strategie

Die Funktion mit dem Namen Patching-Strategie nimmt eine Liste von Patches und versucht, sie in umgekehrter Reihenfolge an ihren Zielort anzuwenden. Für jeden Patch wendet die Funktion sequenziell die verfügbaren Patching-Taktiken an. Derzeit wird zuerst die Jump-Taktik versucht, gefolgt von der Punned Jump-Taktik.

Die Methode gibt Feedback zum Erfolg jedes Patch-Versuchs und fasst am Ende die Abdeckungsinformationen zusammen.

3.4.2.3 Assembler

Der Assembler erzeugt Code, der keine weitere Verlinkung erfordert. Damit der Assembler Instruktionen mit relativen Offsets wie Sprüngen oder relativen Ladeanweisungen korrekt kodieren kann, wird eine Startadresse zusammen mit den Anweisungen als Stream als Eingabe verwendet.

Der Assembler unterstützt die grundlegenden Funktionen der Assemblersprache: die Kodierung mehrerer Xtensa-Assembler-Anweisungen aus der Xtensa ISA-Zusammenfassung, Assembler-Direktiven wie `.align 4` für 4-Byte-Ausrichtung, Labels für Code-Standorte und Kommentare [Cad22].

3.4.3 Flashen nach dem Binary Rewriting zurück auf das Gerät

Sobald das Rewriting abgeschlossen ist, kann die Binärdatei vom ELF-Format wieder in das ESP-Anwendungsformat konvertiert und zurück auf das Gerät geflasht werden.

Die im Rewriting-Befehl angegebene Adresse muss die gleiche Adresse sein, von der die Binärdatei wiederhergestellt wurde. Andernfalls kann der Bootloader die Datei nicht finden. Diese Adresse kann in der wiederhergestellten Partitionstabelle nachgeschlagen werden.

3.5 Proof of Concept

Der Binary Rewriter hat verschiedene Anwendungsmöglichkeiten, wie das Einfügen beliebigen Codes oder das Anwenden von Drittanbieter-Sicherheitspatches, ohne den Kontrollfluss des Originalprogramms zu ändern. Um das Potenzial des Tools zu demonstrieren, wurde ein PoC entwickelt, das sich auf die Instrumentierung für Fuzzing konzentriert. Dieses Tool wurde entwickelt, um das Potenzial des Binary Rewriters zu zeigen. Daher konzentriert es sich nur auf das Sammeln von Abdeckungsinformationen für Funktionsaufrufe, ohne Verzweigungen oder Schleifen zu verfolgen.

3.5.1 Entwicklung eines Beispiel-Tools

Drei Optionen wurden in Betracht gezogen, um die Zähler zu implementieren, die benötigt werden, um Abdeckungsinformationen für das Fuzzing zu sammeln:

- **Flash-Speicher:** Die Nutzung eines ungenutzten Bereichs des Flash-Speichers zur Speicherung von Zählervariablen kann zu Leistungsverbesserungen führen. Allerdings birgt dieser Ansatz Risiken, beispielsweise das Verlieren der gespeicherten Informationen bei einem Stromausfall oder einem Geräteabsturz. Dies ist besonders problematisch für Fuzzing, bei dem gezielt nach Abstürzen gesucht wird.
- **Non-Volatile Storage (NVS)-Partition:** Das Hinzufügen einer NVS-Partition zum Firmware-Image, um Zählerinformationen zu speichern, kann eine effiziente Lösung darstellen. Allerdings erfordert dies eine Modifikation der Partitionstabelle und die Implementierung des NVS-Zugriffs in Assembler.
- **Überwachungsnachrichten:** Das Senden von eindeutigen Funktionskennungen an ein verbundenes Gerät durch Überwachungsnachrichten ermöglicht das Zählen von Funktionsaufrufen. Auf dem ESP32 werden Daten, die an `stdout` und `stderr` gesendet werden, z. B. über `printf`, an ein Überwachungsgerät weitergeleitet. Diese Option ist am einfachsten zu implementieren, stößt jedoch an ihre Grenzen, wenn man Abdeckungsinformationen für alle Funktionen erfassen möchte. Der Grund dafür ist, dass das Einfügen eines Zählers am Beginn einer Funktion wie `printf` und aller von ihr aufgerufenen Funktionen eine Endlosschleife erzeugen würde.

Für den PoC wurden Überwachungsnachrichten über die `printf`-Funktion implementiert, da dies für die Aufgabe die natürlichste Option darstellt. Während das Fuzzing von Standardbibliotheksfunktionen wie `printf` für eine umfassende Software-Sicherheit grundsätzlich wichtig ist, stellt das Fehlen dieser Möglichkeit hier einen vernachlässigbaren Nachteil dar, da der Fokus auf benutzerdefiniertem Anwendungscode liegt.

3.5.2 Implementierung des Beispiel-Tools

In diesem Abschnitt wird die Implementierung des Beispiel-Tools näher erläutert. Dabei werden das Speicherlayout und das String-Handling auf dem ESP32, die Erzeugung von Patch-Code sowie die Auswahl vorhandener Strings und Register für die Instrumentierung behandelt. Abschließend werden die Anwendungspunkte der Patches und die Überwachungsstrategie erläutert.

3.5.2.1 Strings auf dem ESP32

Auf dem ESP32 werden Strings im Abschnitt `.flash.rodata` gespeichert. Die 24-Bit-Instruktionsgröße der Xtensa-ISA erlaubt es nicht, 32-Bit-absolute Adressen direkt zu kodieren, und die `l32r`-Instruktion, die zum Laden von 32-Bit-Werten verwendet wird, hat einen begrenzten Bereich. Daher werden Zeiger auf Strings am Anfang des Abschnitts

`.flash.text` abgelegt und mit der `l32r`-Instruktion in Register geladen, bevor `printf` aufgerufen wird.

```
<.flash.rodata:>
0x3f4041a8:    48 65 6c 6c 6f  ; Hello
               20 77 6f 72 6c  ;  Worl
               64 21 00        ; d!\0

<.flash.text:>
0x400d0618:    a8 41 40 3f     ; pointer to 0x3f4041a8

0x400d500f:    a1 82 ed        ; l32r a10, -0x049f7
0x400d5012:    e5 77 05        ; call8 <printf>
```

Listing 3.1: Strings auf dem ESP32

3.5.2.2 Patch-Code

Erste Versuche, benutzerdefinierte Strings im Patch-Code zu verwenden, schlugen fehl, vermutlich aufgrund von Einschränkungen beim Zugriff auf den Speicher oder Problemen bei der Codeausrichtung, die verhindern, dass die Hardware Bytes direkt aus dem Abschnitt `.flash.text` liest. Es zeigte sich jedoch, dass die Nutzung vorhandener Strings innerhalb der Binärdatei, wie sie von Systemfunktionen verwendet werden, eine effektive Alternative für Instrumentierungszwecke darstellt.

Es ist wichtig zu beachten, dass die Fähigkeit, benutzerdefinierte Strings in die Binärdatei einzufügen, in bestimmten Szenarien wünschenswert sein mag, jedoch keine kritische Anforderung für viele Formen der Sicherheitstests wie Fuzzing oder abdeckungsbasierte Instrumentierung darstellt. Das Hauptziel des Rewriters ist es, beobachtende Instruktionen einzufügen, ohne den Kontrollfluss der Firmware zu verändern, und dieses Ziel wird unabhängig von der Quelle der Strings erreicht. Daher bietet die Verwendung von bereits vorhandenen Strings eine praktische Lösung, ohne die Nützlichkeit oder Wirksamkeit des Frameworks zu beeinträchtigen.

Daher wurden bestehende Strings in der Binärdatei verwendet, wobei mehrere Kriterien berücksichtigt wurden:

- **Verfügbarkeit:** Der String muss in allen ESP32-Binärdateien vorhanden sein. Strings in FreeRTOS, einem kleinen Betriebssystem, das häufig in ESP32-Anwendungen verwendet wird, und in den in ESP32-Code kompilierten Funktionen erfüllen dieses Kriterium.
- **Position:** Der Zeiger auf den String muss nahe genug an der `l32r`-Instruktion liegen, damit diese ihn laden kann.
- **Struktur:** Der String muss einen 32-Bit-Integer als letzten Parameter akzeptieren. Dadurch kann die Datenstruktur, die die aktuelle Adresse des Funktionsaufrufs enthält, direkt ohne zusätzliche Konvertierungen ausgegeben werden.

Der String „`W (%lu) %s: Flash clock frequency round down to %d`“ wurde ausgewählt und gekürzt. Sein Zeiger und die Adresse der `printf`-Funktion müssen beim Initialisieren des Tools vorhanden sein.

Die Kernfunktion `add_fuzzing_counter` setzt den Patch-Code zusammen, lädt den String-Zeiger, schneidet den String ab, lädt die Zähleradresse und ruft `printf` auf.

```
def add_fuzzing_counter(self, addr:int):
    fuzzing_counter_patch = [
        "            l32r a10, " + hex(self.__addr_string_pointer),
        "            addi a10, a10, 52", # Kuerze den Anfang des Strings
        "            j jmplabel", # Springe (jump) ueber den Data Block
        "            .align 4",
        "addrlabel:  .uint32 " + hex(addr),
        "jmplabel:   l32r a11, addrlabel", # Immer -4
        "            call8 " + hex(self.__addr_printf_function)
    ]

    self.rewriter.add_patch(addr, fuzzing_counter_patch, moved_after_patch=
    True)
```

Listing 3.2: Definieren des Patches

Die Register `a10` und `a11` können gefahrlos verwendet werden, da das Registerfenster während des Funktionsaufrufs um 8 verschoben wurde und somit keine wichtigen Daten überschrieben werden.

Patches werden auf die `entry`-Instruktion angewendet, die den Startpunkt jeder Funktion markiert. Dadurch wird sichergestellt, dass jeder Zähler nur einmal pro Funktionsaufruf ausgelöst wird, da der Kontrollfluss innerhalb eines Funktionsaufrufs nie zur `entry`-Instruktion zurückkehrt.

3.5.2.3 Überwachung

Ein Überwachungsskript wurde implementiert, um die `printf`-Ausgaben auf dem Überwachungsgerät zu sammeln, Daten aus dem `stdout`-Stream zu filtern und Adressen zu zählen, die vom Patch-Code gesendet werden. Nach einer festgelegten Zeit wird das Zählen gestoppt und die Ergebnisse werden angezeigt.

3.5.3 Verwendung des Beispiel-Tools

Mithilfe des Beispiel-Tools kann nun Fuzzing-Instrumentierung in eine bestehende Binärdatei eingefügt werden. Nach dem Zurückflashen auf das Gerät werden die Ergebnisse mithilfe des Überwachungsskripts gesammelt.

Zunächst muss das Tool durch Angabe des Pfads zur extrahierten Binärdatei initialisiert werden. Anschließend können Fuzzing-Zähler hinzugefügt werden. Der Benutzer muss derzeit die Adressen selbst identifizieren, an denen die Zähler platziert werden sollen. Dies kann beispielsweise mithilfe eines Disassemblers erfolgen. Diese Adressen werden über die Methode `add_fuzzing_counter` eingebunden. Zudem muss das Framework darüber informiert werden, wo zusätzlicher Code sicher eingefügt werden kann, ohne die bestehende Funktionalität zu beeinträchtigen. Dies erfolgt mit der Methode `add_free_space`. Dieser Schritt ist entscheidend für die Integrität und die korrekte Ausführung des Programms.

Nach Abschluss dieser Schritte können die Patches angewendet werden. Dabei werden die neuen Instruktionen korrekt in die Binärdatei eingefügt, mit der bestehenden Code-Struktur ausgerichtet und Sprünge angepasst. Abschließend muss die umgeschriebene ELF-Binärdatei gespeichert und auf das Gerät zurückgeflasht werden.

```
inserter = Fuzzing_Instrumentation_Inserter('extracted.elf')

inserter.add_fuzzing_counter(0x400e248c)
inserter.add_fuzzing_counter(0x400d4fc0)
inserter.add_fuzzing_counter(0x400d4fdc)

inserter.add_free_space(0x400e23a8, 0x400e2489)

inserter.apply_patches()
util.save_file("patched.elf", inserter.get_elf_bytes())
```

Listing 3.3: Verwendung des Beispiel-Tools

Nachdem das Binary Rewriting abgeschlossen ist und die Binärdatei auf das Gerät zurückgeflasht wurde, kann das Überwachungsskript ausgeführt werden, um die Abdeckungsinformationen zu sammeln. Das Skript verfolgt die `printf`-Aufrufe, die vom Patch-Code ausgeführt werden. Nach Abschluss des Durchlaufs werden die Ergebnisse angezeigt.

```
 > python3 monitoring.py
[COUNTER] 0x400d4fdc
[COUNTER] 0x400d4fc0
  [...]
[COUNTER] 0x400e248c
[FINISHED] Found 3 counters in 30 seconds!
```

Listing 3.4: Ausführen des Überwachungsskripts

3.6 Einschränkungen und Ausblick

Während dieser Forschung wurden mehrere Einschränkungen festgestellt. Eine wesentliche Einschränkung ist der begrenzte Speicherplatz auf dem ESP32-Gerät für Patches.

Während kleine Patches in Pufferbereichen untergebracht werden können, erfordern größere Änderungen möglicherweise eine Erweiterung der bestehenden Codeabschnitte innerhalb der Binärdatei oder eine Modifikation des Bootloaders, um zusätzlichen Code beim Starten zu laden.

Ein Satz von Instruktionen, der derzeit nicht verschoben werden kann, sind Instruktionen mit relativen Offsets, die in diese eingebettet sind. Die Umsiedlung dieser Instruktionen würde eine Neuberechnung ihrer Offsets erfordern. Dies zu vermeiden, ist eine grundlegende Designphilosophie von Trampolin-Rewritern.

Die aktuelle Implementierung unterstützt nur eine begrenzte Anzahl von Assembler-Instruktionen, was die Komplexität der anwendbaren Patches einschränkt. Darüber hinaus sind die aktuellen Patch-Strategien auf die Jump- und Punned Jump-Taktiken beschränkt. Die Implementierung der Taktiken Successor Eviction und Neighbor Eviction, die beide eine tiefere Integration eines Disassemblers erfordern, würde die Erfolgsrate der erfolgreich angewendeten Patches erhöhen.

Zukünftige Arbeiten könnten sich auf die Behebung der identifizierten Einschränkungen und die Verbesserung der Fähigkeiten des Binary Rewriters konzentrieren. Ein möglicher Verbesserungsbereich ist die automatische Erweiterung von Binärabschnitten, insbesondere des `.flash.text`-Abschnitts, um zusätzlichen Raum für das Patchen zu schaffen, ohne andere Teile der Binärdatei zu beeinflussen. Darüber hinaus würde die Erweiterung der Assembler-Unterstützung auf den vollständigen Bereich der Xtensa ISA-Instruktionen oder die Integration eines externen Assemblers eine größere Flexibilität bei der Anwendung komplexerer Patches bieten.

Die Patch-Taktiken Successor Eviction und Neighbor Eviction waren für das PoC-Tool nicht erforderlich und wurden daher nicht implementiert. Komplexere Programme würden erheblich von ihrer Implementierung profitieren. Neben diesen beiden Taktiken könnte die Untersuchung neuer Patch-Taktiken, die spezifische Merkmale der Xtensa-Architektur nutzen, ebenfalls die Effizienz des Rewriters steigern.

Die Reverse-Order-Patching-Strategie ist zwar effektiv, aber möglicherweise nicht in allen Szenarien optimal. Die Verbesserung der Patch-Strategien durch Experimente mit heuristischen oder zufallsgestützten Ansätzen könnte die Erfolgsrate der Patch-Anwendungen weiter erhöhen.

Der Fuzzing-Instrumentierungs-Inserter wurde als PoC ausgewählt, da er ein vielversprechender Anwendungsfall für das Binary-Rewriting-Framework ist. Wie dargestellt wurde, zeigt das Tool großes Potenzial für Sicherheitstests von Drittanbietern, muss jedoch weiterentwickelt werden, bevor es in realen Szenarien angewendet werden kann. Dies könnte die automatische Erkennung der Fuzzing-Instrumentierungspositionen und die Hinzufügung von Zählern für Schleifen und Verzweigungen umfassen.

3.7 Zusammenfassung

Mithilfe des in diesem Kapitel vorgestellten Ansatzes wurde die bestehende Lücke bei der Unterstützung von Tools für unabhängige Sicherheitsexperten geschlossen, um proprietäre ESP32-Firmware analysieren und testen zu können. Der neuartige Ansatz besteht in der Entwicklung eines Binary-Rewriting-Frameworks, das erstmals die gezielte Instrumentierung proprietärer ESP32-Firmware ohne Veränderung der ursprünglichen Funktionalität und des Kontrollflusses ermöglicht. Durch die Integration von Fuzzing-Zählern und die Nutzung von Codeabdeckungsinformationen können verschiedene Ausführungspfade effizienter erkundet und potenzielle Sicherheitslücken gezielter identifiziert werden.

Das Framework vereinfacht den Prozess der Firmware-Analyse und -Modifikation, indem es die extrahierte Firmware in ein besser handhabbares Format konvertiert, wodurch präzise Änderungen vorgenommen werden können, während die Integrität des Originalcodes gewahrt bleibt. Es führt neue Patch-Methoden ein, die speziell auf die Xtensa-Architektur zugeschnitten sind und etablierte Techniken an die spezifischen Bedürfnisse von ESP32-Geräten anpassen. Die Wirksamkeit des Frameworks wurde durch ein Proof of Concept demonstriert, das erfolgreich Codeabdeckungsinformationen zu ESP32-Binärdateien hinzufügte. Dieser Ansatz umfasst das Einfügen eines Zählers, der die Anzahl der Ausführungen eines bestimmten Codeabschnitts erfasst. Durch die Verwendung dieses Feedbacks kann der Fuzzer verschiedene Ausführungspfade effizienter erkunden, was die Wahrscheinlichkeit erhöht, Fehler und Sicherheitslücken zu finden. Dies zeigt das Potenzial für die Weiterentwicklung des Frameworks, um die Sicherheit von ESP32-Firmware zu verbessern.

Zukünftige Arbeiten sollten sich darauf konzentrieren, die Vielseitigkeit des Frameworks durch die Implementierung zusätzlicher Patch-Taktiken zu erweitern und neue zu entwickeln, insbesondere um komplexere Patch-Szenarien anzugehen. Eine fortlaufende Verfeinerung dieses Frameworks wird seine Fähigkeiten erweitern und die Sicherheitsanalyse von ESP32-Geräten weiter unterstützen.

3.8 Fazit

Die vorgestellten Ergebnisse zum Binary Rewriting zeigen, dass eine gezielte Instrumentierung von ESP32-Firmware auf Binärebene technisch realisierbar ist. Damit wurde eine zentrale Grundlage geschaffen, um Laufzeitinformationen wie Codeabdeckung und Kontrollfluss direkt zu erfassen und für automatisierte Sicherheitsanalysen nutzbar zu machen.

Gleichzeitig hat sich jedoch gezeigt, dass Fuzzing und dynamische Analysen auf realer Hardware aufgrund begrenzter Ressourcen, langsamer Ausführungszeiten und eingeschränkter Skalierbarkeit nur eingeschränkt praktikabel sind. Die erzielten Ergebnisse verdeutlichen somit die Notwendigkeit, ergänzende Ansätze zu verfolgen, die eine effizientere und flexiblere Testdurchführung ermöglichen. Aus diesem Grund wird im folgenden Kapitel ein Framework zur Emulation von ESP32-Firmware in QEMU vorgestellt.

4 Fuzzing von ESP32-Mikrocontrollern mittels QEMU-Emulation

Für dieses Kapitel bietet eine gemeinsame Veröffentlichung mit Sven Nitzsche, Max Eisele, Roland Gröll, Jürgen Becker und Ingmar Baumgart die Grundlage. Teile der Ergebnisse wurden bereits in der unten genannten Publikation veröffentlicht. Es wird ein emulatorbasiertes Fuzzing-Framework für ESP32-IoT-Geräte vorgestellt, das es ermöglicht, ESP32-Anwendungen effizient in virtuellen Umgebungen zu testen und Sicherheitslücken deutlich schneller aufzudecken als durch klassisches Fuzzing auf realer Hardware.

- **Matthias Börsig**, Sven Nitzsche, Max Eisele, Roland Gröll, Jürgen Becker und Ingmar Baumgart. „Fuzzing Framework for ESP32 Microcontrollers“. In: 2020 IEEE International Workshop on Information Forensics and Security (WIFS). IEEE, Dez. 2020, S. 1–6. DOI: 10.1109/wifs49906.2020.9360889 [Bör+20].

4.1 Einleitung

Die im vorherigen Kapitel vorgestellten Techniken zur Instrumentierung auf Binärebene haben gezeigt, dass sich ESP32-Firmware gezielt erweitern lässt, um Laufzeitinformationen wie Codeabdeckung und Kontrollfluss direkt zu erfassen. Diese Methoden bilden die Grundlage für automatisierte Sicherheitsanalysen, stoßen jedoch auf praktische Grenzen: Begrenzte Hardware-Ressourcen, langsame Ausführung und eingeschränkte Skalierbarkeit limitieren das Fuzzing direkt auf dem IoT-Gerät.

Zur Überwindung dieser Einschränkungen wird in diesem Kapitel ein Fuzzing-Framework für die ESP32-Xtensa-Architektur vorgestellt. Es verlagert die Analyse von ESP32-Firmware vollständig in eine emulierte Umgebung. Dieser Ansatz entkoppelt den Testprozess von der physischen Hardware und ermöglicht umfangreiche, parallele und automatisierte Fuzzing-Kampagnen. Damit wird ein zentraler Baustein des Gesamtziels dieser Dissertation adressiert, nämlich die effiziente Identifikation von Schwachstellen in proprietärer ESP32-Firmware.

Das Fuzzing-Framework basiert auf dem Emulator *QEMU* und kombiniert einen angepassten *ESP32-Fork*[1] mit einer erweiterten Implementierung von *Honggfuzz*[2], einem QEMU-basierten feedbackgesteuerten Fuzzer. Um dies zu ermöglichen, waren mehrere technische

[1] `https://github.com/espressif/qemu`
[2] `https://github.com/thebabush/honggfuzz-qemu`

Erweiterungen notwendig, insbesondere die Anpassung der Emulationsumgebung an die Xtensa-Architektur und die Implementierung feingranularer Codeabdeckung-Feedback-Mechanismen.

Die wesentlichen Beiträge lassen sich wie folgt zusammenfassen:

- **Anpassung von QEMU an die Xtensa-Architektur:** Erweiterung des QEMU-Emulators zur präzisen Ausführung von ESP32-Firmware und zur Unterstützung der Instrumentierung für Laufzeitanalysen. Diese Funktionalität war zuvor in keinem verfügbaren Fuzzing-Tool vorhanden.
- **Integration von QEMU und Honggfuzz:** Entwicklung eines kombinierten Frameworks, das ESP32-spezifische Emulation mit einem QEMU-basierten feedbackgesteuerten Fuzzer verbindet und so automatisierte Whitebox-, Greybox- und Blackbox-Fuzzing-Kampagnen ermöglicht.
- **Implementierung feingranularer Codeabdeckung-Feedback-Mechanismen:** Nutzung der internen Strukturen des QEMU-Binary-Translators, um Basisblöcke und Vergleichsinstruktionen abzufangen und an *Honggfuzz* weiterzugeben. Dies erhöht deutlich die Präzision, indem die Codeabdeckung analysiert wird.
- **Praktische Evaluation und Machbarkeitsnachweis:** Validierung des Frameworks anhand einer kommerziellen IoT-Lampe (LIFX Mini). Dabei konnten mehrere Abstürze und eine sicherheitsrelevante Nullzeiger-Dereferenzierung identifiziert werden. Besonders hervorzuheben ist die signifikant höhere Performance: bis zu 320 Eingaben pro Sekunde im Greybox-Fuzzing, im Vergleich zu 80 Eingaben pro Sekunde für reines Whitebox-Fuzzing und nur 4 Eingaben pro Sekunde mit compilerbasierter Instrumentierung.

Das Framework unterstützt verschiedene Fuzzing-Ansätze, einschließlich Whitebox-, Greybox- und Blackbox-Fuzzing, und bietet dadurch eine hohe Flexibilität für unterschiedliche Anwendungsszenarien. Das Fuzzing-Framework schließt damit eine wesentliche Lücke in der Landschaft der verfügbaren Tools und zeigt, dass die Kombination aus Emulation und Fuzzing die Analyse proprietärer ESP32-Firmware deutlich effizienter gestaltet.

Als Machbarkeitsnachweis wurde das Fuzzing-Framework erfolgreich eingesetzt, um ein kommerzielles IoT-Gerät zu analysieren. Dabei konnten innerhalb weniger Minuten mehrere Fehler und eine potenziell sicherheitskritische Schwachstelle entdeckt werden, was die Leistungsfähigkeit und praktische Relevanz des Frameworks unterstreicht.

4.2 Stand der Technik

In diesem Kapitel werden Arbeiten vorgestellt, die entweder ähnliche methodische Ansätze wie die vorliegende Dissertation verfolgen oder unmittelbar als Grundlage für deren Umsetzung dienten. Ziel ist es, den aktuellen Stand der Technik einzuordnen, konzeptionelle Verwandtschaften aufzuzeigen und den Mehrwert des vorgestellten Ansatzes klar abzugrenzen.

Im Bereich der Integration von Fuzzern in Systememulatoren existieren relevante Vorarbeiten. Hertz und Newsham [HN17] entwickelten mit *TriforceAFL* ein QEMU-basiertes Fuzzing-Framework auf Basis von American Fuzzy Lop (AFL), das ganze Systeminstanzen testen kann. Allerdings basiert dieses Projekt auf einer veralteten QEMU-Version, die nicht mit der aktuellen ESP32-QEMU-Implementierung kompatibel ist. Die Grundidee dieser Integration diente als Inspiration für die Dissertation. Sie wurde jedoch in veränderter Form umgesetzt: Es wird eine aktuelle QEMU-Version genutzt und eine direkte Verbindung zu einer erweiterten ESP32-Emulation hergestellt.

Voss [Vos17] hat eine Technik entwickelt, um schwer erreichbaren Code gezielt zu testen. Hierzu wird der Code auf dem Zielsystem bis kurz vor der Verarbeitung der relevanten Funktion ausgeführt und der gesamte Systemzustand an diesem Punkt eingefroren. Anschließend wird dieser Zustand in einen Emulator übertragen, in dem generierte Testdaten injiziert und die Ausführung fortgesetzt wird. Gui et al. [Gui+20] erweiterten diesen Ansatz, indem sie einen zusätzlichen Analyseprozess integrierten, der relevante Codebereiche vor dem Fuzzing-Prozess automatisch identifiziert. Beide Arbeiten implementierten ihre Methoden im Unicorn-Emulator [QV15], der jedoch die Xtensa-Architektur des ESP32 nicht unterstützt. Die Dissertation überträgt das Prinzip des Zustandsübertrags auf eine QEMU-basierte ESP32-Emulation und umgeht somit diese Architekturbeschränkung.

Insgesamt greifen die in dieser Dissertation entwickelten Methoden zentrale Konzepte bestehender Ansätze auf, wie die Integration von Fuzzern in Systememulatoren und den gezielten Zustandsübertrag. Im Gegensatz zu bisherigen Arbeiten werden diese Konzepte jedoch erstmals in einer aktuellen QEMU-Umgebung mit vollständiger Unterstützung der Xtensa-Architektur des ESP32 zusammengeführt. Durch gezielte Erweiterungen der Emulationsumgebung und die nahtlose Anbindung eines modernen Fuzzers entsteht ein Setup, das den Funktionsumfang und die Anwendbarkeit bisheriger Lösungen deutlich übertrifft.

4.3 Konzeption

Für dieses Konzept wurde *Honggfuzz* als Fuzzer ausgewählt, da er eine gute Balance zwischen Leistungsfähigkeit, einfacher Integration und Unterstützung für verschiedene Fuzzing-Strategien bietet. Insbesondere ermöglicht *Honggfuzz* die Nutzung von Feedback über die Codeabdeckung, um die Eingabegenerierung gezielt zu steuern und die verschiedenen Ausführungspfade effizienter zu erreichen. Darüber hinaus verfügt er über eine bestehende Schnittstelle zur Kommunikation mit externen Fuzzing-Hooks, was den Einsatz auf dem ESP32 erheblich erleichtert. Ein Fuzzing-Hook ist ein kleines Programm oder Skript, das Eingabedaten vom Fuzzer entgegennimmt, sie an das Testziel übermittelt und dabei die Antwort bzw. die Lebenszeichen überwacht. Im Fehlerfall meldet er dies dem Fuzzer zurück.

Fuzzing besteht aus drei Schritten: Fehlererkennung, Zielausführung und Eingabegenerierung, die im Folgenden erläutert werden.

4.3.1 Fehlererkennung

Zunächst muss das Verhalten des ESP32 bei Speicherbeschädigungen untersucht werden. Dazu wurde eine Testanwendung entwickelt, die Anfragen über die WLAN-Schnittstelle verarbeitet und das absichtliche Auslösen der fünf Hauptursachen für Speicherbeschädigungen ermöglicht (Stack- und Heap-Pufferüberlauf, unsichere Verwendung von `printf`, Nullzeiger-Dereferenzierung und doppeltes Freigeben von Speicher), basierend auf [Mue+18].

Das Schreiben außerhalb der Grenzen eines zugewiesenen Puffers ist auf dem ESP32 möglich. Puffer im Stack befinden sich meist in der Nähe von Rücksprungadressen, während Puffer im Heap auch neben Funktionszeigern liegen können. Das Überschreiben einer dieser Adressen führt zu einem Absturz des Geräts, falls darauf zugegriffen wird. Werden keine wichtigen Werte überschrieben, bleibt das ESP32 funktionsfähig und der Fehler bleibt möglicherweise unentdeckt.

Mit Kontrolle über das erste Argument einer `printf`-Funktion ist es möglich, an bestimmte Adressen auf dem Stack zu schreiben. Falls diese Adressen auf ungültige Speicherbereiche verweisen, führt dies zu einem Absturz des Geräts, wodurch die Speicherbeschädigung erkennbar wird.

Beim Dereferenzieren eines Nullzeigers und beim erneuten Freigeben eines bereits deallokierten Speicherblocks stürzt der ESP32 immer ab. Diese Arten der Speicherbeschädigung sind daher stets beobachtbar.

Es ist wichtig zu beachten, dass der Fuzzing-Prozess eine Speicherbeschädigung nicht unbedingt beim ersten Auftreten erkennen muss. Während des Fuzzing-Prozesses werden zahlreiche Eingaben getestet, sodass ein vorhandener Fehler, der zu einer Speicherbeschädigung führt, höchstwahrscheinlich durch verschiedene Eingaben ausgelöst werden kann. In der Regel erfolgt die Detektion durch Beobachtung von Systemabstürzen. Im weiteren Verlauf dieses Abschnitts wird daher auf die Identifikation von Systemabstürzen zurückgegriffen, um durch das Fuzzing provozierte Fehler zu erkennen.

Zur Verbesserung der Fehlererkennung könnten Tools wie *AddressSanitizer* oder heuristische Methoden durch Emulation eingesetzt werden [Mue+18].

4.3.2 Zielausführung mit Fuzzing-Hooks

Da der ESP32 für IoT-Anwendungen entwickelt wurde, wird die Eingabe normalerweise über die WLAN-Schnittstelle empfangen. Daher ist das Senden der Fuzzing-Daten über WLAN die bequemste Methode beim Testen auf dem tatsächlichen Gerät. Dies geschieht durch einen sogenannten Fuzzing-Hook, der die Fuzzing-Eingabedaten wiederholt von der *Honggfuzz*-Schnittstelle abrufen und an die Netzwerkadresse des Ziels senden muss. Der Hook muss außerdem überprüfen, ob das Ziel auf die Anfrage geantwortet hat. Falls keine Antwort eingeht, wird angenommen, dass das Ziel abgestürzt ist, und der Fuzzing-Hook muss ein Fehlersignal an den Fuzzer senden.

Einige Ziele erfordern möglicherweise eine zusätzliche *Vitalitätsprüfung*, um zu untersuchen, ob das Ziel durch die Anfrage nicht abgestürzt ist. Daher wird nach dem Senden der Anfrage mit den Fuzzing-Daten eine zusätzliche Anfrage gesendet, von der bekannt ist, dass das Ziel darauf antwortet.

4.3.3 Feedbackgesteuerte Eingabegenerierung

Für die Eingabegenerierung wird der Mutationsmechanismus von *Honggfuzz* verwendet. Er kann entweder durch einfache Mutation der bereitgestellten Seed-Eingaben oder durch zusätzliche Berücksichtigung der Codeabdeckungsinformationen des Ziels erfolgen. Folgende Methoden zum Sammeln von Codeabdeckungsinformationen stehen zur Verfügung: Compiler-generierte Instrumentierung, Binary Rewriting und Emulation.

4.3.3.1 Compiler-generierte Instrumentierung

Der ESP32-Compiler unterstützt die Instrumentierung des Codes zur Generierung von Codeabdeckungsdaten, die dann zur Laufzeit der Anwendung im Speicher des Geräts gespeichert werden. Um die generierten Abdeckungsdaten zu verarbeiten, muss der Fuzzing-Hook diese Daten nach jeder getesteten Eingabe über eine JTAG-Debugging-Verbindung herunterladen und an den Fuzzer weiterleiten. JTAG ist eine standardisierte Schnittstelle zum Testen, Debuggen und Auslesen von Chips. Über sie ist ein direkter Zugriff auf interne Register, Speicher und Signalpfade möglich.

Leider werden bei dieser Methode der Codeinstrumentierung nur die ausgeführten Basisblöcke und keine Zweig- oder Pfadabdeckungen protokolliert.

4.3.3.2 Binary Rewriting zur Instrumentierung

Eine Möglichkeit, feedbackgesteuertes Greybox-Fuzzing auf dem ESP32 zu realisieren, besteht darin, den Binärcode der Anwendung gezielt zu instrumentieren. Durch diese Code-Modifikation können beispielsweise ausgeführte Basisblöcke oder Parameter von Vergleichsinstruktionen erfasst und an den Fuzzer zurückgemeldet werden.

Kapitel 3.1 beschreibt den Ansatz ESP32 Binary Rewriting, der gezielte Code-Injektionen auf Binärebene ermöglicht. Damit lassen sich zusätzliche Instruktionen direkt in bestehende ESP32-Firmware einfügen, ohne deren ursprüngliche Funktionalität oder den Kontrollfluss zu beeinträchtigen. Dieses Verfahren eröffnet grundsätzlich die Möglichkeit, Messpunkte zur Erfassung der Codeabdeckung effizient in die Firmware einzubetten.

Allerdings ist Binary Rewriting nur eine von mehreren möglichen Strategien zur Instrumentierung. In dieser Dissertation werden im Folgenden weitere Ansätze betrachtet und bewertet.

4.3.3.3 Codeabdeckung durch Emulation

Beim Ausführen einer Anwendung in einer Emulationsumgebung stehen alle Metadaten zur Programmausführung zur Verfügung. Diese Transparenz ermöglicht es, den Programmzähler und Vergleichsparameter abzufangen, um eine feingranulare Codeabdeckung zu berechnen.

Honggfuzz bietet hierfür eine modifizierte *QEMU*-Version (QEMU-HONGFUZZ) an, die feedbackgesteuertes Fuzzing unterstützt. Da die ESP32-Firmware jedoch nur in einer vollständigen Systememulation lauffähig ist, muss die offizielle ESP32-QEMU-Implementierung genutzt werden [Esp19]. Diese ist jedoch unvollständig und emuliert wesentliche Peripherien wie WLAN nicht. Daher müssen alternative Mechanismen zur Eingabeübertragung in die Anwendung entwickelt werden.

Die Nutzung einer vollständigen Systememulation für Fuzzing ist in der Forschung stark diskutiert, insbesondere hinsichtlich der Performance. Einige Studien berichten, dass Emulation zwei- bis fünfmal [Zha+18] und in Extremfällen bis zu zehnmal [Zhe+19] langsamer sein kann als das Ausführen auf echter Hardware. Andere Arbeiten zeigen jedoch, dass Emulatoren – gerade bei ressourcenbeschränkten Embedded-Geräten – sogar schneller sein können als das reale System [Mue+18]. Diese möglichen Unterschiede machen eine gezielte Evaluation der Performance von ESP32-QEMU für relevante Szenarien des Fuzzings notwendig.

4.4 Implementierung

Das Fuzzing-Framework für den ESP32 nutzt verschiedene Methoden, die speziell auf die Einschränkungen von ressourcenarmen IoT-Geräten abgestimmt sind. Das Ziel besteht in der Entwicklung effizienter Testverfahren, die auf realer Hardware oder in Emulationsumgebungen eingesetzt werden können. Je nach Verfügbarkeit des Quellcodes kommen dabei unterschiedliche Ansätze zum Einsatz: Blackbox-Fuzzing ohne Quellcodezugriff, Whitebox-Fuzzing mit vollem Zugriff sowie Greybox-Fuzzing auf Basis von Instrumentierung.

4.4.1 Blackbox-Fuzzing auf ESP32-Anwendungen

Blackbox-Fuzzing ohne Berücksichtigung der Codeabdeckung auf dem eigentlichen Gerät ist die einfachste Methode zum Testen von ESP32-Anwendungen. Obwohl sie nicht besonders vielversprechend ist, wurde sie implementiert, um eine Vergleichsgrundlage zu schaffen.

Einige Dienste warten auf bestimmte Symbole am Ende der Eingabedaten, bevor die Verarbeitung beginnt. Ein Header einer HTTP-Anfrage endet beispielsweise mit zwei Zeilenumbrüchen. Falls diese Zeichen nicht empfangen werden, bleibt die Verarbeitung der Daten stecken, was den gesamten Fuzzing-Prozess blockieren könnte. Daher müssen

diese Zeichen identifiziert und von der Fuzzing-Hook an die generierten Eingabedaten angehängt werden, um Deadlocks zu vermeiden.

Die Absturzerkennung des Ziels muss ebenfalls an den jeweiligen Dienst angepasst werden. Bei verbindungsorientierten Diensten wie TCP reicht es aus, zu überprüfen, ob die Verbindung korrekt beendet wurde oder ob eine Antwort vom Ziel empfangen wurde. Bei verbindungslosen Diensten wie UDP müssen komplexere Methoden verwendet werden, um festzustellen, ob das Ziel durch die Verarbeitung der Eingabedaten abgestürzt ist.

Eine Möglichkeit besteht darin, nach jeder getesteten Eingabe eine Vitalitätsprüfung durchzuführen. Solche Vitalitätsprüfungen sind jedoch nicht immer zuverlässig. Eingebettete Systeme können nach einem Absturz sehr schnell neu starten, sodass die Prüfung fälschlicherweise eine erfolgreiche Antwort registriert.

Eine ausgefeiltere Methode zur Absturzerkennung besteht darin, Absturzsignale über eine serielle Verbindung zum Zielgerät abzufangen. Wenn der ESP32 abstürzt, gibt er immer eine Fehlermeldung, gefolgt von einer Neustartnachricht auf der seriellen Schnittstelle, aus. Das Abfangen dieses Neustartsignals könnte ebenfalls eine zuverlässige Möglichkeit zur Absturzerkennung sein.

Mit dieser Methode konnte eine einfache HTTP-Server-Anwendung für den ESP32 mit einer Rate von etwa 30–40 Anfragen pro Sekunde getestet werden. Diese Rate ist jedoch zu niedrig für effektives Fuzzing, da sie die Anzahl und Vielfalt der Testfälle begrenzt und somit die Chance verringert, Fehler oder Sicherheitslücken zu finden. Zudem liefert sie bei wiederholten ähnlichen Eingaben kaum neue Erkenntnisse. Für aussagekräftige Ergebnisse sind deutlich höhere Durchsatzraten und abwechslungsreichere Eingaben erforderlich.

4.4.2 Whitebox-Fuzzing mit compilerinstrumentiertem Code

Die Berücksichtigung der Codeabdeckung einer getesteten Eingabe für die Eingabegenerierung ist notwendig, um die Effizienz des Fuzzing-Prozesses zu erhöhen. Der einfachste Weg, die Codeabdeckung zu erfassen, besteht darin, die vom Compiler generierte Codeinstrumentierung zu nutzen. Dies geschieht, indem jede zu instrumentierende Quelldatei mit der Compileroption `-coverage` neu kompiliert wird. Der Compiler fügt dadurch jedem Basisblock Code hinzu, der zählt, wie oft er während der Laufzeit ausgeführt wurde und speichert die Informationen im RAM.

Die Abdeckungsdaten werden dann über eine JTAG-Verbindung von dem Gerät in *Honggfuzz* übertragen. *Honggfuzz* erstellt eine große Bitmap im gemeinsam genutzten Speicher, in der die Adressen der ausgeführten Basisblöcke gespeichert werden. Jedes Bit in dieser Bitmap entspricht einer Adresse, wobei zunächst alle Bits auf null gesetzt sind. Wird ein Basisblock zum ersten Mal ausgeführt, wird das entsprechende Bit auf eins gesetzt. *Honggfuzz* kann diese Bitänderung erkennen und speichert die auslösende Eingabe als zusätzliches Seed im Eingabeordner.

Die Nutzung solcher Compiler-generierten Abdeckungsdaten verlangsamt den Fuzzing-Prozess um den Faktor 10, sodass beim Fuzzing der einfachen HTTP-Serveranwendung nur

etwa 4 Anfragen pro Sekunde erreicht werden. Darüber hinaus werden mit dieser Methode nur die ausgeführten Basisblöcke untersucht und es kann keine feingranulare Codeabdeckung generiert werden. Der größte Nachteil der Compiler-generierten Abdeckungsdaten besteht jedoch darin, dass der Quellcode verfügbar sein muss.

4.4.3 Whitebox-Fuzzing mit ESP32-QEMU-FUZZ

QEMU-HONGFUZZ untersucht die Codeabdeckung einer emulierten Benutzeranwendung und leitet sie an *Honggfuzz* weiter. Diese modifizierte Version von QEMU wurde in die von *Espressif* bereitgestellte ESP32-QEMU-Implementierung integriert. Da beide Implementierungen auf nahezu derselben Version von QEMU basieren und Modifikationen in verschiedenen Bereichen des Codes aufweisen, war die Zusammenführung der beiden Codebasen unkompliziert. Das Ergebnis ist ESP32-QEMU-FUZZ (EQF), eine Version von QEMU, die das Fuzzing von ESP32-Anwendungen ermöglicht.

Um feedbackgesteuertes Fuzzing zu ermöglichen, muss untersucht werden, welche Basisblöcke ausgeführt wurden, wie in Abschnitt 4.3.3 erläutert. Die binäre Übersetzungsengine von QEMU gruppiert Anweisungen in Basisblöcke, um ihre Ausführung ohne Unterbrechung zu ermöglichen. Dieser Gruppierungsmechanismus wird genutzt, um Basisblöcke für den Fuzzing-Prozess zu bestimmen.

Durch den Einsatz eines Emulators kann eine noch feingranularere Codeabdeckung erzielt werden, indem die Parameter von Vergleichsanweisungen berücksichtigt werden. Daher werden die beiden Parameter einer Vergleichsanweisung während der Übersetzung der Maschinenanweisung innerhalb des Emulators abgefangen. Die ESP32-Architektur stellt dafür die Funktionen `strcmp` und `strcasecmp` bereit, die zwei Zeichenfolgen entweder exakt oder unter Ignorierung der Groß- und Kleinschreibung vergleichen. Alle String-Vergleichsfunktionen befinden sich an festen Adressen im nicht veränderbaren ROM des ESP32. Dadurch ist es möglich, die Parameter der Funktionen abzufangen, wenn eine solche Funktion innerhalb des Emulators aufgerufen wird. Der Code von *Honggfuzz* muss erweitert werden, um diese Parameter zusätzlich zu verarbeiten. Die Parameter und die Adresse, von der aus die String-Vergleichsfunktion aufgerufen wird, können auf dieselbe Weise an *Honggfuzz* übergeben werden wie die Parameter normaler Vergleichsanweisungen.

Zum Zeitpunkt dieser Untersuchung verfügte ESP32-QEMU noch nicht über eine Emulation des integrierten WLAN-Moduls. Erst nach der Veröffentlichung der zugrunde liegenden Publikation [Bör+20] wurde eine entsprechende Erweiterung veröffentlicht (siehe Related Work Abschnitt 8.3). Deshalb musste auch ESP32-QEMU-FUZZ ohne diese Komponente auskommen. Es wurde die von QEMU bereitgestellte Ethernet-Schnittstelle genutzt, um den Emulator mit dem Netzwerk des Hosts zu verbinden. Dabei muss die Anwendung gegen den Ethernet-Treiber gelinkt werden, wodurch diese Kommunikationsmethode ausschließlich für Whitebox-Szenarien einsetzbar ist.

Um die Eingabedaten des Fuzzers an die Netzwerkadresse des Hosts weiterzuleiten, wird ein Fuzzing-Hook erstellt. Dieser ist dafür verantwortlich, die Fuzzing-Eingabedaten iterativ von *Honggfuzz* abzurufen und sie an die richtige Netzwerkadresse weiterzuleiten.

Zur Fehlererkennung reicht es aus, den HALT-Interrupt abzufangen, der ausgelöst wird, wenn der ESP32-Emulator abstürzt. Dieser Interrupt signalisiert dem Emulator oder Debugger, dass ein kritischer Fehler oder eine Ausnahmesituation aufgetreten ist, die die weitere Ausführung stoppt.

Diese Einrichtung bietet eine ausgezeichnete Möglichkeit für Whitebox-Fuzzing von ESP32-Anwendungen. Für die TCP-Testanwendung wurden mit einem einzelnen Thread auf einem Standard-Notebook etwa 80 Anfragen pro Sekunde erreicht. Daher kann davon ausgegangen werden, dass die ESP32-QEMU-FUZZ-Implementierung hinsichtlich der gesamten Netzwerk- und Datenverarbeitung schneller ist als das eigentliche Gerät.

Zusätzlich wird eine feingranulare Codeabdeckung für jede Eingabe zur Generierung neuer Eingaben berücksichtigt, was die Effizienz des Fuzzing-Prozesses erhöht. Diese Methode kann leicht skaliert werden, indem mehrere Instanzen genutzt werden. In diesem Fall muss jede Instanz an eine eigene Netzwerkschnittstelle gebunden werden.

Mithilfe dieser Whitebox-Fuzzing-Implementierung in QEMU lassen sich automatisierte Fuzz-Tests in moderne Continuous-Integration- und Continuous-Delivery-Entwicklungszyklen integrieren. Dazu wird die Anwendung separat mit dem erforderlichen Ethernet-Treiber sowie optionalem Platzhalter-Code für nicht emulierbare Hardwareteile kompiliert. Dies betrifft beispielsweise spezielle Sensoren oder Aktoren, die hardwareseitig direkt angeschlossen sind und nicht per QEMU emuliert werden können.

4.4.4 Blackbox- und Greybox-Fuzzing mit ESP32-QEMU-FUZZ

Wie bereits erwähnt, unterstützt ESP32-QEMU keine WLAN-Funktionalität. Da sich Blackbox-Firmwares mit WLAN-Funktionalität nicht emulieren lassen, ist Blackbox-Fuzzing nur bei netzwerkloser Firmware möglich. Da ESP32-basierte Mikrocontroller jedoch häufig gerade wegen des kostengünstigen WLAN-Moduls eingesetzt werden, stellt dies eine erhebliche Einschränkung dar. Eine mögliche Lösung wäre die Emulation der WLAN-Funktionalität in ESP32-QEMU. Für eine korrekte Implementierung sind jedoch detaillierte Kenntnisse der Hardware und der Treiberanforderungen erforderlich. Leider sind die WLAN-Treiber des IoT Development Framework (IDF) Closed Source, und es gibt keine Dokumentation zur entsprechenden WLAN-Hardware in der Dokumentation[3]. Um die internen Kommunikationsmechanismen zu analysieren, wäre umfangreiches manuelles Reverse Engineering erforderlich. Daher wurde die WLAN-Funktionalität nicht implementiert und kein Blackbox-Fuzzing umgesetzt. Stattdessen wird ein Greybox-Ansatz für das Fuzzing von Firmware-Images verfolgt.

Um Greybox-Binary-Fuzzing zu ermöglichen, wurde die Technik aus [Vos17] implementiert. Diese erlaubt es, die Datenzufuhr über Netzwerkschnittstellen zu umgehen. Dies geschieht, indem ein Zustand des tatsächlichen Geräts nach dem Empfang der Daten gespeichert und anschließend in die Emulation übertragen wird. Für den Fuzzing-Prozess reicht es aus, den Code vom Beginn der Datenverarbeitung bis zu dessen Ende auszuführen. Dies beschleunigt den Fuzzing-Prozess zusätzlich, da irrelevante Teile nicht mehr

[3] `https://docs.espressif.com/projects/esp-idf/en/stable/esp32/get-started/index.html`

ausgeführt werden. Um dies umzusetzen, müssen die Ein- und Ausstiegspunkte der Datenverarbeitung innerhalb der Firmware identifiziert werden. Allerdings stellt fast jede zufällige Bitfolge von zwei oder drei Bytes eine gültige Instruktion im *Xtensa*-Befehlssatz dar. Infolgedessen können selbst professionelle Disassemblierungsprogramme wie *IDA Pro* den *Xtensa*-Code nicht korrekt disassemblieren. Das Auffinden der Ein- und Ausstiegspunkte erfordert deshalb einen hohen manuellen Aufwand.

Ein Ansatz, um diese Codebereiche zu identifizieren, ist die schrittweise Ausführung der Firmware mit GNU Debugger (GDB). Ein tiefgehendes Verständnis des Codes ist erforderlich, das durch das Setzen von Breakpoints und die Beobachtung der Ausführung erlangt werden kann. Das Ziel ist es, Datenverarbeitungsfunktionen und Codeabschnitte zu finden, die sich gut für das Fuzzing eignen. Wenn der Codeabschnitt nicht an einer einzigen Stelle endet, müssen mehrere Ausstiegspunkte definiert werden. Zudem muss der Speicherbereich, in dem sich die Eingabedaten befinden, manuell identifiziert werden.

Sobald geeignete Ein- und Ausstiegspunkte gefunden wurden, muss der Zustand des Zielgeräts beim Erreichen des Einstiegspunkts gespeichert werden. Der Zustand des ESP32 besteht aus den Werten der 16 Register, dem Program Counter und den 512 kB statischen RAM. Alle diese Daten können über eine JTAG-Debugging-Verbindung ausgelesen und somit leicht gespeichert werden. Um diesen gespeicherten Zustand in die Emulatorinstanz zu laden, muss die QEMU-Implementierung modifiziert werden, sodass das Speicherabbild in den entsprechenden Speicherbereich geladen und alle Registerwerte korrekt gesetzt werden. Die Eingabedaten befinden sich dann im Speicher des Emulators und können modifiziert werden.

Damit die Ausführung im Emulator ordnungsgemäß fortgesetzt werden kann, darf der Zustand nicht vor Abschluss der Initialisierungsroutinen der Firmware geladen werden. Vielmehr muss die Firmware zunächst alle wichtigen Initialisierungsroutinen durchlaufen, um sicherzustellen, dass erforderliche Module funktionieren. Der Punkt, an dem die Initialisierung des Betriebssystems abgeschlossen ist, wird als Setup-Punkt bezeichnet und muss ebenfalls manuell identifiziert werden. Eine geeignete Methode, um den Setup-Punkt einer Firmware zu bestimmen, ist die Ausführung der Firmware für einige Sekunden im Emulator, gefolgt von einem Stopp mit dem Debugger. In der Regel befindet sich das Gerät dann in einem Leerlaufzustand und wartet auf Eingaben.

Bei jeder Fuzzing-Iteration werden die Eingabedaten durch die vom Fuzzer generierten Daten überschrieben, und der Längenwert wird entsprechend angepasst. Sobald einer der definierten Ausstiegspunkte erreicht wird, beginnt der Prozess von vorn. Daher wird jede getestete Eingabe ausgeführt, während sich der Emulator stets im exakt gleichen Zustand befindet — einzig die Eingabedaten variieren.

Das erneute Laden des gesamten Zielgerätezustands nach jeder getesteten Eingabe bringt jedoch einen erheblichen Performance-Nachteil mit sich. Daher wurde eine Technik namens *Fork Server* implementiert, die im Folgenden erläutert wird.

Auf *UNIX*-Systemen wird der *fork*-Aufruf nach der *copy-on-write*-Richtlinie realisiert. Das bedeutet, dass der neu erstellte Prozess den Speicher mit dem übergeordneten Prozess

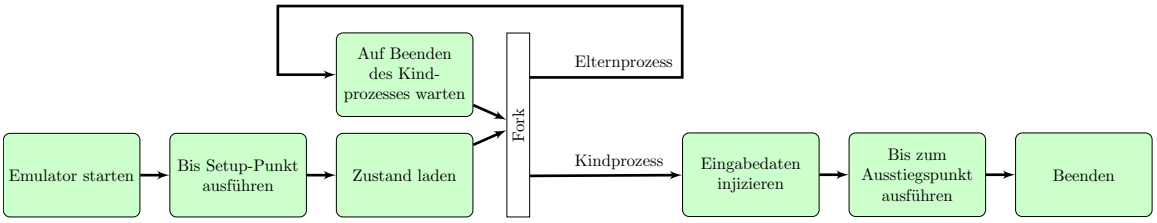

Abbildung 4.1: Fork-Join-Fuzzing-Prozess

teilt und eine gesamte Speicherseite erst dann kopiert wird, wenn einer der Prozesse eine Schreiboperation ausführt. Der *Fork-Join*-Fuzzing-Prozess ist in Abbildung 4.1 dargestellt.

Das Verhalten des *fork*-Aufrufs kann im Fuzzing-Prozess genutzt werden, um sicherzustellen, dass in jeder neuen Fuzzing-Iteration der gleiche Zustand des Emulators vorhanden ist. Dazu wird ein Kindprozess unmittelbar vor der Injektion der Eingabedaten in den Emulator erstellt. Der Elternprozess wartet mittels *join*-Systemaufruf auf das Beenden des Kindprozesses. Anhand des Rückgabecodes kann bestimmt werden, ob der Kindprozess aufgrund eines Absturzes oder des Erreichens eines Ausstiegspunkts beendet wurde. Anschließend kann ein neuer Kindprozess mit neuen Eingabedaten des Fuzzers erzeugt werden. Durch den *Fork-Join*-Mechanismus wird der Performance-Verlust beim erneuten Laden des Gerätezustands erheblich reduziert. Mit diesem Mechanismus kann der oben dargestellte Performance-Nachteil des Greybox-Binary-Fuzzings minimiert werden.

4.5 Evaluation

Im Folgenden werden die Ergebnisse der Evaluation präsentiert. Grundlage hierfür sind eine TCP-basierte Testanwendung auf dem ESP32 sowie ein kommerzielles IoT-Gerät. Mithilfe dieser Anwendung werden die Fehlerdetektion, Abdeckung und Stabilität der implementierten Fuzzing-Methoden untersucht. Das Ziel besteht darin, die Leistungsfähigkeit der verschiedenen Ansätze in einem praktischen Einsatzumfeld zu bewerten.

4.5.1 Fuzzing der TCP-Testanwendung

Wie in Abschnitt 4.3.1 erwähnt, wurde eine TCP-Testanwendung verwendet, um die Implementierung von ESP32-QEMU-FUZZ zu verifizieren. Die Funktionsweise ließ sich am Beispiel des feedbackgesteuerten Whitebox-Fuzzings überprüfen. Innerhalb weniger Stunden wurde eine Eingabekombination gefunden, durch die alle implementierten Fehler ausgelöst wurden. Zudem wurden weitere Testszenarien durchgeführt, um die Geschwindigkeit der einzelnen Versuchsarten zu vergleichen. Die erzielten Anfragen pro Sekunde sind in Tabelle 4.1 aufgeführt. Daraus geht hervor, dass das Fuzzing direkt auf dem Gerät in diesem Szenario deutlich weniger Anfragen pro Sekunde verarbeiten kann als die Emulation. Der größte Unterschied zeigt sich zwischen Whitebox-Fuzzing mit kompilierinstrumentiertem Code (ca. vier Anfragen pro Sekunde) und Greybox-Fuzzing mit

Tabelle 4.1: Vergleich der Fuzzing-Versuche auf einem Intel i7-6600U basierten Standard-Notebook mit einem Kern

Test	Anfragen pro Sekunde
Blackbox-Fuzzing auf ESP32-Anwendungen	40
Whitebox-Fuzzing mit Compiler-instrumentiertem Code	4
Whitebox-Fuzzing mit ESP32-QEMU-FUZZ	80
Greybox-Fuzzing mit ESP32-QEMU-FUZZ	320

ESP32-QEMU-FUZZ (ca. 320 Anfragen pro Sekunde): Hier ergibt sich eine rund 80-fache Steigerung.

4.5.2 Greybox-Fuzzing der LIFX Mini

Um die Effektivität der implementierten Fuzzing-Methode zu beweisen, wurde sie an dem kommerziellen Produkt *LIFX Mini* getestet. Die *LIFX Mini* ist eine smarte Glühbirne, die einen ESP32 als Steuerungseinheit enthält. Sie ist ein typisches IoT-Endverbrauchergerät, das über WLAN gesteuert wird. Für die Steuerung wird eine Smartphone-App angeboten, mit der mehrere smarte Glühbirnen gleichzeitig verwaltet werden können. Wie bei kommerziellen Geräten zu erwarten, ist kein Quellcode verfügbar.

4.5.2.1 Vorbereitung des Zielobjekts

Bei der ersten Untersuchung des Zielgeräts wurde festgestellt, dass der JTAG-Port deaktiviert wurde. Die permanente Deaktivierung des JTAG-Ports ist eine Sicherheitsfunktion des ESP32, die nicht rückgängig gemacht werden kann. Daher musste die Firmware der Glühbirne auf eine ESP32-Entwicklungsplatine übertragen werden, um die JTAG-Verbindung nutzen zu können. Der 4 MB große Flash-Speicher des ESP32, der die Firmware enthält, kann über eine serielle Verbindung ausgelesen werden. Um die Pins für die serielle Verbindung zu erreichen, musste die Glühbirne geöffnet werden, wie Abbildung 4.2 zeigt.

Anschließend wurde die Firmware mit dem *esptool* ausgelesen, das von IDF bereitgestellt wird. Das etwa 750 kB große Firmware-Image wurde anschließend in den Flash-Speicher der ESP32-Entwicklungsplatine geschrieben, die mit einem JTAG-Adapter verbunden ist. Dies ermöglichte es, Breakpoints an beliebigen Stellen zu setzen und den Zustand des Geräts auszulesen.

4.5.2.2 Fuzzing des Initialisierungsprozesses

Für die erste Konfiguration betreibt die Glühbirne ihren eigenen WLAN-Zugangspunkt. Die Smartphone-App lässt das Telefon mit diesem Zugangspunkt verbinden und eine

Abbildung 4.2: Eine demontierte LIFX Mini smarte Glühbirne.

sichere TLS-Verbindung zum Gerät über den TCP-Port 56700 herstellen. Die Anmeldedaten des WLAN-Zugangspunkts des Benutzers werden somit über einen sicheren Kanal ausgetauscht. Nach der ersten Konfiguration verbindet sich die Glühbirne mit dem WLAN-Zugangspunkt des Benutzers, und weitere Steuerbefehle werden über dieses Netzwerk übertragen. Der Fokus des ersten Fuzzing-Ansatzes für die Glühbirne liegt auf diesem Initialisierungsprozess.

Zunächst mussten die Ein- und Ausstiegspunkte mit GDB gefunden werden. Als nächstes mussten der gespeicherte Gerätezustand und die Metadaten dem ESP32-QEMU-FUZZ über die JTAG-Debugging-Schnittstelle zur Verfügung gestellt werden. Als Ergebnis konnten mehrere Abstürze innerhalb weniger Minuten beobachtet werden. Es wurden zwei verschiedene Wege gefunden, das Gerät in eine Endlosschleife zu versetzen, bei der das Gerät etwa 30 Sekunden lang nicht nutzbar war. Nach dieser Zeit startet das Gerät neu. Dieser Fehler könnte für DoS-Angriffe ausgenutzt werden, um das Gerät unbrauchbar zu machen. Außerdem wurde eine Eingabe gefunden, die das Gerät zum Absturz brachte und einen Neustart auslöste. Der Neustart scheint jedoch durch einen eingebauten Neustart-Befehl ausgelöst zu werden. Speicherbeschädigungen oder Fehler-Signale konnten durch die manuelle Analyse des Absturzes nicht beobachtet werden.

Nach etwa 45 Minuten explorierte der Fuzzing-Prozess keine neuen Codebereiche mehr. Selbst das weitere Ausführen des Fuzzing-Prozesses über 72 Stunden führte zu keiner neuen Codeabdeckung. Es kann daher davon ausgegangen werden, dass die meisten zugänglichen Code-Teile in den ersten 45 Minuten abgedeckt wurden.

4.5.2.3 Fuzzing auf offenem TCP-Port

Nach der Initialisierung bietet die Glühbirne auch einen HTTP-Server auf TCP-Port 80 an. Nach wenigen Minuten Fuzzing konnte eine Nullzeiger-Ausnahme gefunden werden. Die Ausnahme resultiert aus einer unsicheren Verwendung der `strchr`-Bibliotheksfunktion, die sich im nicht modifizierbaren ROM des ESP32 befindet. Wenn das Zeichen nicht im String enthalten ist, gibt diese Funktion einen Zeiger auf das geforderte Zeichen innerhalb

des Strings oder einen Nullzeiger zurück. In diesem Fall wurde der zurückgegebene Zeiger nicht überprüft, sondern dereferenziert, was die Ausnahme auslöste.

Es gibt bisher noch keine Arbeiten darüber, wie sich diese Nullzeiger-Ausnahmen auf der *Xtensa*-Architektur ausnutzen lassen. Es wurden aber bereits mehrere Möglichkeiten gefunden, diese Art von Ausnahmen auf anderen Architekturen auszunutzen. Nullzeiger-Dereferenzierungen sind in der *2019 CWE Top 25 Most Dangerous Software Errors* [Cor19] auf Platz 14 gelistet. Dieser Fehler sollte daher als potenziell ernsthafte Sicherheitslücke betrachtet werden. Die Entdeckung dieses Fehlers zeigt die Effektivität des feedbackgesteuerten Greybox-Fuzzings.

4.6 Einschränkungen und Ausblick

Die modifizierte Version von QEMU, die in dieser Dissertation entwickelt wurde, ist auf den ESP32 beschränkt. Sie könnte jedoch auf alle eingebetteten Systeme erweitert werden, die von QEMU unterstützt werden. Dies würde es ermöglichen, die implementierten Methoden für eine viel größere Anzahl von Mikrocontroller-Architekturen zu nutzen. Da QEMU jedoch nicht alle verfügbaren Architekturen unterstützt, bleibt die Fähigkeit, beliebige Firmware-Images vollständig zu emulieren, wie in [Mue+18] erwähnt, ein offenes Problem.

4.7 Zusammenfassung

Für einige IoT-Entwicklungsplattformen wurden in der Vergangenheit Fuzzing-Techniken entwickelt. Es gab jedoch keine veröffentlichten Tools für IoT-Geräte, die auf der ESP32-Plattform basieren. Das Ergebnis dieser Publikation ist das erste veröffentlichte Fuzzing-Framework, das speziell für ESP32-Anwendungen entwickelt wurde. Es wurden verschiedene Techniken zum Fuzzing von ESP32-Anwendungen in unterschiedlichen Szenarien implementiert und bewertet. Besonders effektiv sind zwei Methoden, die auf Fuzzing in einem Emulator statt auf dem tatsächlichen Gerät basieren.

Whitebox-Fuzzing ermöglicht automatisierte, kontinuierliche Tests während des Anwendungsentwicklungsprozesses und fügt sich nahtlos in moderne, agile Entwicklungsabläufe ein. Greybox-Fuzzing ist ein leistungsstarkes Tool für Sicherheitsanalysten und erlaubt gezielte Analysen von Firmware-Komponenten, die als potenziell anfällig gelten. Diese Methoden wurden an einem kommerziellen Gerät getestet und konnten innerhalb kurzer Zeit Fehler finden. So wurde unter anderem ein Nullzeiger-Dereferenzierungsfehler entdeckt, der eine potenzielle Sicherheitslücke darstellt. Zwar gibt es derzeit keine veröffentlichten Methoden, um diesen Fehler auf der ESP32-Plattform auszunutzen, doch mit deren zunehmender Popularität ist es nur eine Frage der Zeit, bis Angreifer einen finanziellen Anreiz für Exploits sehen.

4.8 Fazit

Die Virtualisierung der ESP32-Firmware durch EQF hat grundlegend verändert, wie Eingaben an das PUT übermittelt werden. Anstelle der langsamen und ressourcenbegrenzten Kommunikation über WLAN oder andere physikalische Schnittstellen ermöglicht die emulierte Umgebung eine direkte und effiziente Interaktion zwischen Fuzzer und Zielsystem. Dadurch wird erstmals eine Infrastruktur geschaffen, die die gezielte Erzeugung, Aufzeichnung und Analyse großer Mengen an Eingabedaten ermöglicht.

Klassische Fuzzing-Verfahren basieren häufig auf zufällig generierten Eingaben, die vom Zielsystem sofort als ungültig verworfen werden. Dadurch bleibt der zugrunde liegende Protokollzustandsraum weitgehend ungetestet, insbesondere komplexe Logikpfade oder sicherheitskritische Schwächen bleiben unentdeckt. Die Effektivität des Testverfahrens steigt jedoch erheblich, wenn mehr Wissen über Aufbau, Semantik und zulässige Strukturen der Eingaben verfügbar ist.

Vor diesem Hintergrund ergibt sich eine direkte Motivation für die Untersuchung und automatisierte Rekonstruktion der Protokollstrukturen von ESP32-Anwendungen. Nur durch ein gezieltes Verständnis der zugrunde liegenden Datenformate und Kommunikationsprotokolle kann das Fuzzing auf semantisch valide Eingaben ausgedehnt werden. Im nächsten Kapitel wird daher ein Ansatz zum automatisierten PRE von proprietären Netzwerkprotokollen vorgestellt.

5 Protocol Reverse Engineering mittels neuronaler Netze

Für die Inhalte dieses Kapitels wurde eine gemeinsame Veröffentlichung mit Valentin Kiechle, Sven Nitzsche, Ingmar Baumgart, Jürgen Becker, Nico Rausch und Martin Dukek als Grundlage verwendet. Teile der Ergebnisse wurden bereits in den unten genannten Publikationen veröffentlicht. Es wird ein neuartiger Ansatz zum PRE vorgestellt, der auf neuronalen Netzen basiert und somit die bisher manuellen Prozesse automatisiert. Durch den Einsatz verschiedener neuronaler Netzarchitekturen wie CNNs, AE, GANs, LSTMs und SOMs ermöglicht dieser Ansatz die effektive Rekonstruktion textbasierter Netzwerkprotokolle wie HTTP und FTP. Außerdem wird untersucht, inwieweit durch maschinell gelernte Modelle realistische Paketsequenzen generiert werden können, die gezieltes und effektives Fuzzing ermöglichen. Zu diesem Zweck wird ein Evaluationsframework auf Basis von ProFuzzBench entwickelt und auf die Serveranwendungen LightFTP und eine selbst entwickelte Web-Server-Applikation angewendet.

- Valentin Kiechle, **Matthias Börsig**, Sven Nitzsche, Ingmar Baumgart und Jürgen Becker. „PREUNN: Protocol Reverse Engineering using Neural Networks“. In: Proceedings of the 8th International Conference on Information Systems Security and Privacy - ICISSP. **ICISSP 2022 Best Poster Award**. INSTICC. SciTePress, Feb. 2022, S. 345–356. isbn: 978-989-758-553-1. DOI: 10.5220/0010813500003120 [Kie+22].
- Nico Rausch. „Evaluation eines Machine-Learning-basierten Ansatzes zum Protocol Reverse Engineering für effizientes Fuzzing von Netzwerkanwendungen“. Betreuer: **Matthias Börsig** und Martin Dukek, Erstgutachter: PD Dr.-Ing. Ingmar Baumgart, Zweitgutachter: Prof. Dr. Ralf H. Reussner. Masterarbeit am Karlsruher Institut für Technologie, Sep. 2023 [Rau23].

5.1 Einleitung

Während Deep Learning seit 2012 erhebliche Fortschritte in der automatisierten Feature Extraction und Klassifikation erzielt hat [KSH12], blieb das Gebiet des PRE weitgehend von diesem Trend unberührt. Die meisten Publikationen in diesem Bereich stammen aus den Jahren 2004 bis 2013, und neuere Ansätze zur Nutzung neuronaler Netze wurden bislang kaum untersucht.

PRE zielt darauf ab, die Spezifikation eines unbekannten Anwendungsschichtprotokolls aus den Artefakten seiner Kommunikation zu rekonstruieren. Die gewonnenen Informationen

können in einer Vielzahl sicherheitsrelevanter Anwendungen eingesetzt werden, insbesondere im Fuzzing. Das Wissen über die Nachrichtenstruktur und den internen Zustand des Protokolls kann die Abdeckung von Fehlern und Randfällen signifikant erhöhen.

Vor diesem Hintergrund untersucht die Dissertation die Frage, ob und wie sich neuronale Netze zur Automatisierung von PRE-Aufgaben einsetzen lassen. Der Fokus liegt dabei auf der Anwendung und Evaluierung moderner Deep-Learning-Methoden für textbasierte Netzwerkprotokolle sowie der Entwicklung eines modularen Frameworks, das unterschiedliche Architekturen kombiniert.

Die wesentlichen Beiträge lassen sich wie folgt zusammenfassen:

- **Konzeption eines modularen PRE-Ansatzes:** Entwicklung eines vollständig neuronalen, modularen Frameworks zur Protokollanalyse, das verschiedene Architekturen wie AE, CNNs, SOMs und LSTMs integriert.
- **Automatisierte Feature Extraction:** Einsatz von AE und CNNs zur automatischen Identifikation syntaktischer und semantischer Features von Protokollnachrichten, ohne manuelles Feature-Engineering.
- **Clustering von Protokollnachrichten:** Kombination von AE-Features mit SOMs zur Gruppierung von Nachrichten in Nachrichtentypen. Hierdurch konnte die Clustering-Qualität gegenüber einer Baseline um bis zu 19 % verbessert werden.
- **Sequenzgenerierung und Zustandsmodellierung:** Systematische Untersuchung der Wirksamkeit des Ansatzes anhand der Protokolle HTTP 1.1 und FTP. Verwendung von LSTMs zur Generierung neuer, syntaktisch valider Nachrichten (bis zu 67,6 % gültige HTTP- und 100 % gültige FTP-Nachrichten) sowie zur Modellierung der Zustandsübergänge in zustandsbehafteten Protokollen.

5.2 Stand der Technik

Ansätze des PRE lassen sich systematisch nach den verwendeten Datenquellen klassifizieren. Einige Verfahren basieren ausschließlich auf der Analyse von Netzwerknachrichten, während andere zusätzlich dynamische Laufzeitinformationen einbeziehen. Diese werden beispielsweise durch Instrumentierung oder Tracing von Binärdateien gewonnen. Darüber hinaus unterscheiden sich die Ergebnisse: Manche Methoden rekonstruieren lediglich das Nachrichtenformat, während andere auch Zustandsautomaten ableiten. Eine Übersicht der wichtigsten Ansätze gemäß dieser Kriterien zeigt Tabelle 5.1. Der vorgestellte Ansatz positioniert sich im unteren linken Quadranten der Taxonomie und unterscheidet sich von bestehenden Ansätzen durch eine vollständig automatisierte Kombination von Struktur-, Kontext- und Zustandsmodellierung.

Einen der frühesten und einflussreichsten Ansätze stellt *Discoverer* von Cui, Kannan und Wang [CKW07] dar. Dabei wird ausschließlich Netzwerkverkehr in Echtzeit analysiert, um eine abstrakte Protokollspezifikation zu erzeugen. Nachrichten werden dazu in einzelne Token zerlegt, die zunächst zu kleinen Clustern gruppiert und anschließend

Tabelle 5.1: Taxonomie zur Klassifizierung von PRE-Ansätzen nach Anforderungen (Spalten) und Ergebnissen (Zeilen)

PRE-Taxonomie Anforderungen und Ergebnisse	Nur Netzwerknachrichten	Nachrichten und ausführbare Binärdatei zur Laufzeit
Abgeleitetes Nachrichtenformat	Discoverer [CKW07]	Wondracek [Won+08]
Abgeleiteter Zustandsautomat	**PREUNN**	Prospex [Com+09]

rekursiv zusammengeführt werden. Die Annahme, dass Protokolle häufig standardisierte Trennsymbole wie Kommas, Leerzeichen oder Zeilenumbrüche verwenden, ermöglicht die Identifikation von Nachrichtenfeldern mittels Heuristik. Zusätzlich werden Datentypen wie Text- oder Binärfelder abgeleitet und Abhängigkeiten zwischen einzelnen Feldern erkannt. Damit ermöglicht *Discoverer* die automatische Rekonstruktion der Nachrichtenstruktur ohne jegliches Vorwissen. Allerdings bleibt dieser Ansatz auf eine oberflächliche Strukturerkennung beschränkt und berücksichtigt weder semantische Zusammenhänge noch komplexe Zustandsabhängigkeiten. In dieser Dissertation werden die Stärken von *Discoverer* hinsichtlich der Feldidentifikation aufgegriffen und durch weitergehende Modellierungstechniken ergänzt, um eine detailliertere Darstellung komplexer Protokolle zu ermöglichen.

Ein weiterer bedeutender Ansatz ist *Prospex* von Comparetti et al. [Com+09]. Dieser erweitert das *Discoverer*-Verfahren durch eine Kombination aus Netzwerkanalyse und Laufzeitinformationen. Hierbei werden zusätzliche Ausführungstraces von Binärdateien analysiert, wodurch tiefere semantische Einblicke in die Protokollstruktur gewonnen werden können. Die Methode besteht aus mehreren Schritten: Zunächst werden charakteristische Features der Nachrichten mittels Byte-Tainting (einzelne Bytes von Daten markieren und während der Programmausführung verfolgen) und Speicheranalyse extrahiert. Anschließend erfolgt ein Clustering, das neben dem Format der Nachrichten auch deren erzeugte Reaktionen und Antworten berücksichtigt. Auf dieser Grundlage wird ein Akzeptor-Automat konstruiert, der gültige Nachrichtenfolgen modelliert. Ein Akzeptor-Automat ist ein endlicher Automat, der prüft, ob eine Sequenz von Nachrichten den erlaubten Zuständen und Abläufen eines Protokolls entspricht. Er akzeptiert nur gültige Sequenzen und verwirft alle ungültigen, was insbesondere für Fuzzing von zustandsbehafteten Protokollen von Vorteil ist. Der Automat wird anschließend mithilfe des *Exbar*-Algorithmus [Lan99] minimiert, um seine Komplexität zu reduzieren. Das resultierende Modell wird schließlich in den Peach Fuzzer[1] integriert, um gezieltes und effektiveres Fuzzing zu ermöglichen. Durch die Kombination von Netzwerkanalyse, Binärinformationen und Zustandsmodellierung liefert Prospex eine deutlich umfassendere Spezifikation als Discoverer. Allerdings erfordert dieser Ansatz eine aufwendige Vorverarbeitung und hängt stark von heuristischen Annahmen ab. In dieser Dissertation werden ähnliche Grundprinzipien verfolgt, jedoch durch eine vollständig automatisierte, modulare Architektur ergänzt, die explizit Kontextinformationen berücksichtigt und dadurch robustere Modelle erzeugt.

[1] `http://peachfuzzer.com`

Neben heuristischen Ansätzen wurden in jüngeren Arbeiten vermehrt Methoden des maschinellen Lernens für PRE-verwandte Aufgaben eingesetzt. Fu et al. [FC17] kombinieren beispielsweise Blackbox-Fuzzing mit einem Sequenz-zu-Sequenz-LSTM-Modell [SVL14], um Protokollsemantik zu erfassen und neue Eingaben zu generieren. Dieser Ansatz eröffnet neue Möglichkeiten zum PRE, bleibt jedoch auf die Generierung syntaktisch plausibler Nachrichten beschränkt und verzichtet auf Verfahren wie Clustering oder kontextbasierte Modellierung. Das *GANFuzz*-Framework von Hue et al. [Hu+18] nutzt hingegen Sequence Generative Adversarial Nets (SeqGAN) [Yu+17] in Kombination mit Reinforcement Learning. Für jeden Nachrichtentyp wird dabei ein separates Modell trainiert, wobei die Typen heuristisch durch Clustering bestimmt werden. Dieser Ansatz berücksichtigt zwar unterschiedliche Nachrichtentypen, integriert jedoch keinen konsistenten Workflow und basiert bei der Generierung auf Symbolfolgen statt vollständigen Protokollmodellen. Die Dissertation greift dagegen die Ideen des Deep Learning auf und integriert sie in einen ganzheitlichen, automatisierten Prozess, der Nachrichtenstruktur, Kontext und Zustandsabhängigkeiten gleichzeitig modelliert.

Zusammenfassend lässt sich festhalten, dass die bestehenden Ansätze jeweils nur Teilaspekte der Protokollspezifikation erfassen. Discoverer konzentriert sich auf die Nachrichtenstruktur, Prospex erweitert diese um Zustandsautomaten und Machine-Learning-basierte Verfahren verbessern die Generierungsfähigkeit syntaktisch plausibler Nachrichten. Die in dieser Dissertation vorgestellte Methode kombiniert die Stärken dieser Ansätze und geht darüber hinaus: Durch die explizite Integration von Kontextinformationen, den modularen Aufbau und den Einsatz moderner Machine-Learning-Techniken wird eine präzisere, vollständig automatisierte Rekonstruktion komplexer Protokolle ermöglicht. Dadurch werden realistischere Eingaben für Analyse- und Fuzzing-Verfahren erzeugt.

5.3 Hauptansatz

In diesem Abschnitt wird eine neuartige Methode zur systematischen Betrachtung der Aufgabenstruktur beim PRE vorgestellt. Um eine konsistente Referenzierung zu gewährleisten, wird der in diesem Kapitel entwickelte Ansatz im Folgenden als Protocol Reverse Engineering using Neural Networks (PREUNN) bezeichnet.

PREUNN ist in mehrere, überwiegend sequenziell auszuführende Verarbeitungsschritte gegliedert. Die hier vorgestellte modulare Architektur orientiert sich am Vorbild klassischer Verfahren des maschinellen Lernens aus der Spracherkennung [WL90]. Dabei werden komplexe Aufgaben gezielt in Teilschritte wie Merkmalsextraktion, Clustering und Sequenzmodellierung zerlegt. Dieses Design ermöglicht es, einzelne Subsysteme unabhängig voneinander zu optimieren und bei Bedarf durch leistungsfähigere Modelle zu ersetzen. Damit unterstützt PREUNN sowohl die Anpassung an unterschiedliche neuronale Netzwerkarchitekturen als auch die direkte Integration neuer KI-Methoden in den Gesamtprozess.

5.3.1 Datenerfassung

Für das Trainieren neuronaler Netze wird ein repräsentativer Datensatz benötigt. Um die Auswertung der Ergebnisse zu erleichtern, wurde ein Satz textbasierter Artefakte von Anwendungsprotokollen als Grundlage ausgewählt. Dies ist wichtig, da mit dem vorgestellten Ansatz kein direkter Vergleich mit klassischen PRE-Ansätzen möglich ist. Bei einem textbasierten Protokoll lässt sich leicht manuell überprüfen, ob die generierten Daten korrekt sind. Die gewählten Protokolle sind HTTP v1.1 und FTP, da diese häufig verwendet werden, in großer Menge verfügbar sind und keine Verschlüsselung aufweisen. Es werden mehrere Quellen (siehe Abschnitt 5.4.1) von Datensätzen verwendet, um eine breitere Mischung von Implementierungen und Nachrichtenverteilungsarten abzudecken.

5.3.2 Feature Extraction

In diesem ersten Teil werden Features extrahiert, die klar unterscheidbar sind. Sowohl in Prospex als auch in Discoverer war die Auswahl der Features ein wesentlicher Bestandteil der Arbeit, aber die Features wurden von den Forschern ausgewählt. Es ist beabsichtigt, diesen Prozess mit neuronalen Netzwerken zu automatisieren. Von besonderem Interesse sind Schlüsselwörter, Interpunktion, syntaktische Zeichen und andere Muster, die zwischen verschiedenen Nachrichten unterscheiden. Pseudo-zufällige Zeichenfolgen wie Tags und Cookies werden vermieden, da sie in der Regel nicht mit den Protokollspezifikationen in Verbindung stehen.

5.3.3 Reverse Engineering von Features

Beim traditionellen Reverse Engineering von Protokollen dient die Analyse dazu, Regeln, Variablenlisten, Konstanten und Grammatiken aus Kommunikationsartefakten abzuleiten. Bei neuronalen Netzwerken ist das erlernte Wissen hingegen intrinsisch nicht repräsentativ, was eine menschliche Interpretation erschwert. Zur Bewertung der erlernten und rekonstruierten Features durch die jeweiligen Architekturen kommt ein generativer Bewertungsansatz zum Einsatz. Mithilfe dieser Methode werden neue Proben aus der Trainingsverteilung erzeugt, wodurch Einblicke in die internen Lernprozesse der neuronalen Netzwerke ermöglicht werden.

5.3.4 Clustering

Protokollnachrichten können normalerweise in Typen gruppiert werden, unabhängig davon, ob das Protokoll diese Gruppen explizit spezifiziert hat oder nicht. Diese Nachrichten können unter Verwendung von Informationen wie der sequentiellen Reihenfolge, Funktionalität und allgemeinem Format geclustert werden. Da das Clustering verschiedene Nachrichtentypen impliziert, eignet es sich besonders gut für den Einsatz neuronaler Netzwerke [BLP05].

5.3.5 Zustandserkennung

Eine typische Kommunikationssitzung besteht aus mehreren gesendeten und empfangenen Nachrichten. In zustandsbehafteten Protokollen führen bestimmte Nachrichtenfolgen zu komplexeren Zustandsübergängen zwischen den Kommunikationspartnern. Diese sequenziellen Muster repräsentieren einen Zustandsautomaten, der den inneren Zustand des Protokolls beschreibt. Ein neuronales Netzwerk, das sequentielle Abhängigkeiten erfassen kann, lernt dabei sowohl die Reihenfolge verschiedener Nachrichtentypen als auch deren Auftretenswahrscheinlichkeiten. Auf dieser Grundlage lassen sich Methoden der Zeitreihenanalyse anwenden, um den jeweils nächsten Zustand des Protokolls vorherzusagen.

5.3.6 Sequenzgenerierung

Im abschließenden Schritt werden alle trainierten Modelle zu einer generativen PREUNN-KI zusammengeführt. Das Ziel dieser Phase besteht darin, neue Nachrichtensequenzen zu erzeugen, die den in den Trainingsdaten beobachteten Kommunikationsmustern entsprechen. Dabei berücksichtigt das Modell Kontextinformationen wie den Clusterindex sowie sequentielle Abhängigkeiten zwischen Nachrichten, um realistische und konsistente Sequenzen zu generieren. Durch die Integration dieser Informationen kann die KI die zugrunde liegende Struktur und Dynamik des Protokolls nachbilden. Auf diese Weise entstehen synthetische, aber gültige Nachrichtenfolgen, die nicht im Trainingsdatensatz enthalten sind, jedoch ähnliche statistische und semantische Eigenschaften aufweisen.

5.4 Implementierung von PREUNN

In diesem Abschnitt werden die Implementierungsdetails des Hauptansatzes sowie die durchgeführten Experimente zur Evaluierung verschiedener neuronaler Netzwerkarchitekturen beschrieben. Die Hyperparameter der Modelle wurden auf Grundlage empirischer Tests manuell festgelegt. Dabei erfolgte die Auswahl anhand der Trainingsstabilität, der Konvergenzgeschwindigkeit und der erzielten Validierungsgenauigkeit. Eine systematische, automatisierte Hyperparameteroptimierung wurde nicht durchgeführt, da die verfügbare Hardware keine effiziente Exploration größerer Suchräume erlaubte.

Alle Experimente wurden in Python 3 unter Verwendung eines objektorientierten Programmieransatzes implementiert, um eine modulare Struktur und einfache Erweiterbarkeit auf weitere Protokolle sicherzustellen. Als Deep-Learning-Framework kam PyTorch[2] zum Einsatz. Die Entscheidung fiel zugunsten von PyTorch aufgrund seiner dynamischen Berechnungsgraphen, der guten Integration in die Python-Umgebung sowie der hohen Flexibilität bei der Entwicklung und dem Debugging experimenteller Modelle. Sämtliche Experimente wurden auf einer NVIDIA-GTX-970-GPU ausgeführt.

[2] `https://pytorch.org/`

5.4.1 Datenvorverarbeitung

In den Test wurden zwei Datensätze verwendet. Der erste besteht aus der Kombination mehrerer HTTP-Quellen [Gar08; Shi+12; Goo+19; Sha+19], die ausgewählt wurden, um verschiedene Implementierungen und Szenarien abzudecken. Der zweite Datensatz besteht aus FTP-Nachrichten [PP03]. Bevor Experimente durchgeführt werden konnten, wurden die Daten auf Ausreißer und Unregelmäßigkeiten untersucht. Datenengineering nimmt in der Regel einen erheblichen Teil der Zeit in jedem Machine-Learning-Projekt in Anspruch. Dieser Prozess konnte jedoch mit Netzwerkaufzeichnungen in Form von pcap-Dateien (einem Format zum Speichern von Netzwerkpaketen) verkürzt werden. Das Netzwerk-Analysetool Wireshark[3] bietet einen weit verbreiteten Parser für die Analyse von Netzwerkprotokollen. Es wurde mithilfe des Parsers von Wireshark nach gültigen HTTP- und FTP-Paketen gefiltert und der Rest verworfen. Auf diese Weise wurde sichergestellt, dass sich keine ungültigen oder fehlerhaften Pakete im Datensatz befanden.

Die Länge der Anwendungsdatagramme der Anwendungsschicht ist durch das zugrunde liegende TCP-Protokoll begrenzt. Eine Analyse der neuen pcap-Datei zeigte, dass lange Nachrichten nur bei Bildübertragungen (HTTP) oder benutzerdefinierten Nachrichten (FTP) auftreten und anschließend ohne signifikanten Verlust von Informationen abgeschnitten werden können. Das Längenlimit von 1024 Bytes wurde für alle Pakete festgelegt, um die Eingaben des neuronalen Netzwerks zu vereinheitlichen. Nur 0,33 % der HTTP-Daten und 0,18 % der FTP-Daten überschritten dieses Limit und wurden entsprechend abgeschnitten. Die geringe Überschreitungsrate zeigt, dass der gewählte Grenzwert in der Praxis eine sinnvolle Balance zwischen Datenvollständigkeit und Modellkompatibilität darstellt.

Als weiterer Schritt in den HTTP-Experimenten wurde der gesamte Inhalt nach dem Nachrichtenkopf entfernt. Dadurch sollten XML und andere, nicht HTTP-basierte Daten ausgeschlossen werden. Dies wurde erreicht, indem jede Nachricht an der Stelle, an der ein doppelter Zeilenumbruch (\r\n\r\n) auftrat, geteilt wurde. Dabei wurde nur der erste Teil beibehalten. Der doppelte Zeilenumbruch markiert das Ende des HTTP-Headers und den Beginn des Payloads. Somit stellt er eine einfache Methode zur gezielten Bereinigung der Daten dar. Für die FTP-Daten wurde hingegen kein entsprechender Filter angewendet, da die Trennung von Befehlen und Nutzdaten durch separate Steuer- und Datenkanäle bereits sichergestellt ist.

Dataset-Bias ist eine häufige Falle bei der Entwicklung von Machine-Learning-Lösungen. Es beschreibt eine ungleichmäßige Verteilung von Klassen innerhalb der Daten, die zu suboptimalem Lernen der Features in den neuronalen Netzwerken führt. Die Protokolle selbst spezifizieren keine Klassen von Nachrichten direkt, die manuell erstellten Klassifikationen für beide Protokolle orientieren sich jedoch an der Baseline für das Klassenungleichgewicht. Es ist wünschenswert, dass die neuronalen Netzwerke weiterhin lernen, welche Typen häufiger sind als andere, aber das Ungleichgewicht in dem Datensatz ist deutlich zugunsten von zwei oder drei häufigen Nachrichtentypen. Um die Anzahl der Nachrichten

[3] `https://www.wireshark.org/`

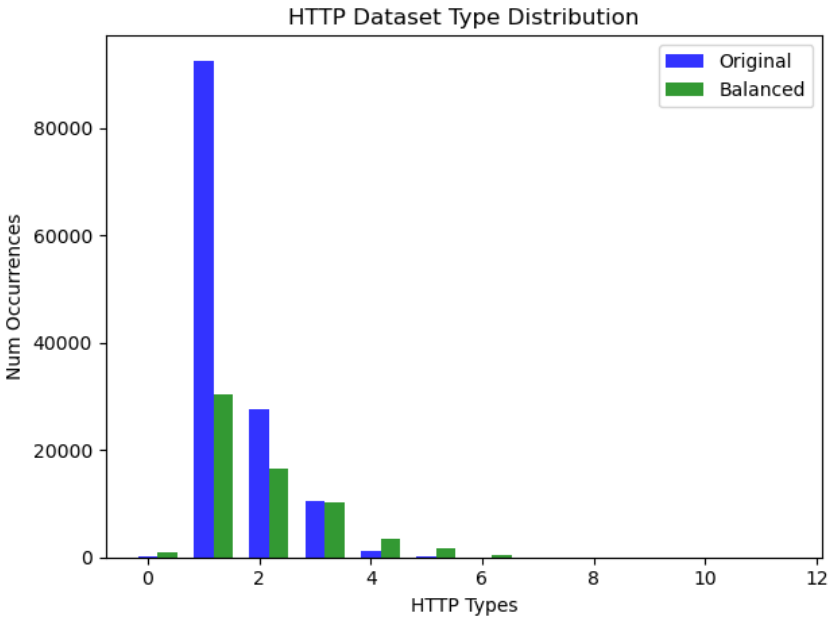

Abbildung 5.1: HTTP-Datensatzverteilungen in Original (blau) und ausgeglichen (grün). Das deutliche Ungleichgewicht zugunsten einzelner Nachrichten wurde gemildert, während dieser Typ weiterhin am häufigsten erhalten blieb.

pro Klasse auszugleichen, wurde diese Formel entwickelt:

$$\text{Anzahl}_{\text{Nachrichten pro Klasse}} = \sqrt{\text{Anzahl}_{\text{Vorkommen}}} * 100$$

Die visualisierte Verteilung der Nachrichtentypen ist in Abbildung 5.1 und in Abbildung 5.2 zu sehen.

Für jede Klasse wurden Netzwerpakete zufällig ausgewählt, bis ein zuvor definiertes Limit erreicht war. Dabei erfolgte eine gezielte Vervielfachung seltener Nachrichtentypen, um in allen Klassen eine ausreichende Stichprobengröße sicherzustellen. Experimente, die im Rahmen dieser Dissertation ohne diesen Ausgleich durchgeführt wurden, zeigten in mehreren Fällen deutliche Anzeichen von Overfitting. Dies unterstreicht die Notwendigkeit einer ausbalancierten Datenverteilung. Um die natürliche Häufigkeitsverteilung der Nachrichten zu bewahren, wurden die Klassen nicht vollständig auf ein einheitliches Limit normiert, sondern proportional zu ihrer ursprünglichen Auftretenswahrscheinlichkeit angepasst.

5.4.2 Feature Extraction

Neuronale Netzwerke sind dafür bekannt, hochgradig flexible, selbstlernende Feature-Extraktoren für unterschiedliche Aufgaben zu sein. In diesem Ansatz wird jedes Zeichen einer Nachricht als Ganzzahlwert interpretiert und die gesamte Nachricht als eindimensionales Bild betrachtet, sodass Verfahren aus bildbasierten Aufgaben angewendet werden können.

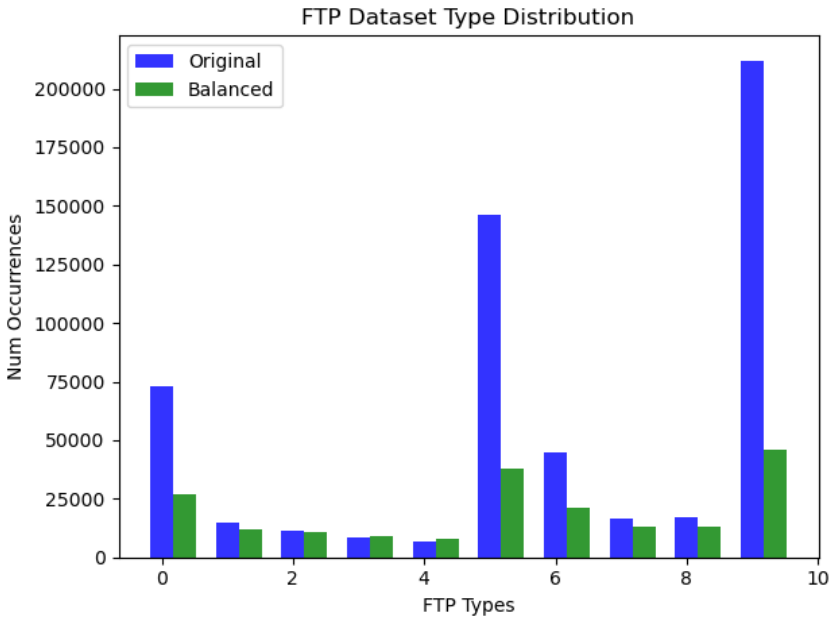

Abbildung 5.2: FTP-Datensatzverteilungen in Original (blau) und ausgeglichen (grün). Bei drei dominanten Typen und mehreren unterrepräsentierten Typen wurde die Umverteilung erheblich geglättet, wobei Tendenzen erhalten blieben.

Zur Feature Extraction wurden zwei Architekturen ausgewählt: ein AE und ein CNN. Die in diesen Experimenten von den Modellen gelernten Zuordnungen von Features werden anschließend als Eingabe für das Clustering verwendet. Aufgrund der Blackbox-Natur neuronaler Netzwerke kann die Qualität der Zuordnungen von Features nicht direkt gemessen werden. Stattdessen werden die Ergebnisse des späteren Clusterings zur Bewertung beider Architekturen herangezogen.

Autoencoder AE arbeiten nur mit Daten fester Länge und lernen eine kompakte Darstellung der Daten. Die Netzwerknachrichten, die als Daten verwendet werden, variieren in der Länge, überschreiten jedoch in der Regel nicht 1024 Bytes. Um sie zu vereinheitlichen, werden kürzere Nachrichten mit Nullen aufgefüllt, um eine Länge von 1024 Bytes zu erreichen. Kleinere Tests haben ergeben, dass das Auffüllen bzw. Kürzen auf diese Länge die Leistung des Modells nicht signifikant beeinträchtigt. Da die erwartete Ausgabe für die AE-Architektur gleich der Eingabe ist, wird die Hamming-Distanz als Leitmaß für den Erfolg während des Experimentierens verwendet. Der AE wurde mit den folgenden Schichtgrößen trainiert: $1024 \rightarrow 256 \rightarrow 128 \rightarrow 256 \rightarrow 1024$. Bei den Tests wurden die Softplus-Aktivierungen verwendet und der Adam-Optimizer auf eine Lernrate von 0,0005 gesetzt. Die Verlustfunktion ist der Mean Squared Error (MSE) und die Batch-Größe = 128 für 10 Epochen. Die resultierende 128 Neuronen breite Ausgabe des Encoder-Teils wurde als Kodierung von Features für eine Nachricht verwendet. Die Daten für dieses Experiment wurden als Pixel interpretiert und haben anschließend eine kontinuierliche Natur im Intervall $[0, 255]$. Die für Trainingsdaten mit einer Länge von 1024 Bytes er-

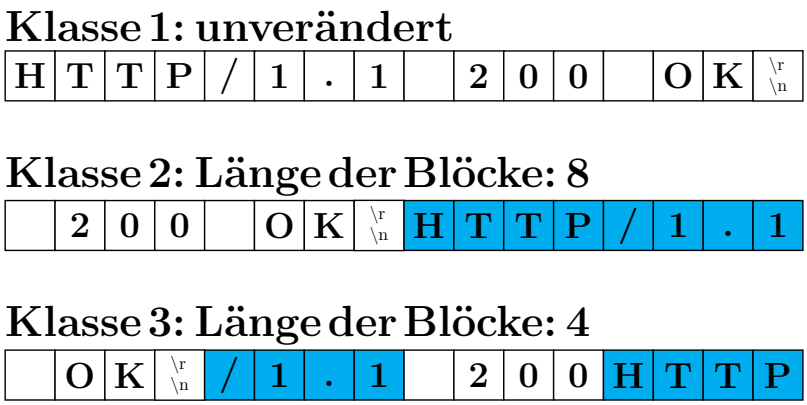

Abbildung 5.3: Eine HTTP-Anweisung, die in 3 Klassen aufgeteilt und vermischt wurde (unverändert sowie Blöcke der Länge 8 und 4). Die Farbe gibt die Blockgröße an. Die Reihenfolge der Blöcke wurde zufällig angeordnet.

reichte Distanz beträgt im Durchschnitt 254,44 für HTTP und 41,28 für FTP. Die sehr hohe Hamming-Distanz resultiert aus der kontinuierlichen Natur der Byte-Interpretationen, selbst die Füllsymbole (0) konnten nicht vollständig rekonstruiert werden. Es ist festzustellen, dass Füllsymbole oft denselben ASCII-Wert um 1 oder 2 Werte verfehlen, wenn die ASCII-Tabelle als Skala von 0 bis 255 interpretiert wird. In einem Bild wären solche kleinen Farbfehler kaum wahrnehmbar. Für das Alphabet als kontinuierliche Skala sehen die Ergebnisse oft falsch aus. Das Hauptziel eines Autoencoders ist jedoch, die Dimensionalität von Features mit minimalem Verlust von Informationen zu reduzieren und Muster zu lernen. In diesem Fall ist es gelungen, die Dimension von 1024 Bytes auf 128 Bytes zu reduzieren und dabei zufriedenstellende Features für die Rekonstruktion während des Experiments zu bewahren. Die niedrigere durchschnittliche Hamming-Distanz für FTP lässt sich durch die viel kürzere durchschnittliche Nachrichtenlänge erklären.

Convolutional Neural Network CNNs werden üblicherweise für die überwachte Bildklassifikation verwendet [KSH12]. Die für diese Dissertation gewählten Trainingsdaten enthalten jedoch keine Labels, die für das überwachte Lernen verwendet werden können. Deshalb wurde eine eigene unüberwachte Lerntechnik unter Verwendung von Datenaugmentation entwickelt. Bei der enwickelten Lerntechnik werden Nachrichten repliziert und in mehrere bekannte Klassen von Augmentierungstypen modifiziert, um Informationen über den syntaktischen Kontext zu extrahieren. Die zugrunde liegende Idee besteht darin, dass ein CNN durch das Training an der Unterscheidung zwischen diesen Augmentierungsklassen in die Lage versetzt wird, die charakteristischen Muster der Syntax effizient zu erkennen und sich darauf adaptiv einzustellen. HTTP- oder FTP-Anweisungen werden wiederum auf eine feste Länge von 1024 Bytes aufgefüllt/gekürzt und dann in Segmente verschiedener Längen (1024 Bytes, 32 Bytes, 16 Bytes, 8 Bytes, 4 Bytes) unterteilt, die dann innerhalb derselben Anweisung in zufälliger Reihenfolge eingegeben werden. Dieser Ansatz führt zu 5 Klassen: unverändert und in Blöcke der Längen 32, 16, 8 und 4 Bytes durcheinander gemischt. Ein vereinfachtes Beispiel wird in Abbildung 5.3 illustriert.

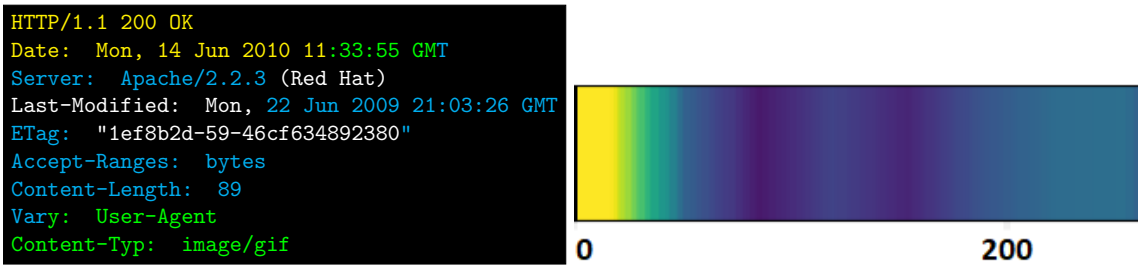

Abbildung 5.4: Beispiel einer HTTP-Anweisungs-Klassifizierung, visualisiert durch Grad-CAM. Die aufgefüllten Nullen wurden weggelassen. Die semantisch signifikanten und konsistenten Teile werden hervorgehoben, während pseudo-randomisierte Teile im ETag ignoriert werden. Die Textmarkierungen in dieser Abbildung sind eine Annäherung.

Es wurde eine Architektur gewählt, die Blöcke mit 1D-Faltung, 1D-Batch-Normalisierung, Softplus-Aktivierung und 20%-Dropout in insgesamt 5 Schichten verwendet, wobei die Kanalgrößen wie folgt aussehen: $1 \longrightarrow 128 \longrightarrow 64 \longrightarrow 32 \longrightarrow 16 \longrightarrow 8$. Die Klassifikationsaufgabe wurde durch zwei vollständig verbundene Schichten durchgeführt, die nach dem Training entfernt wurden, um eine Feature Map mit $8 * 30 = 240$ Neuronen zu erhalten, die größer ist als die des Autoencoders. Eine Visualisierung der gelernten Features wird als zusätzliche Analysemethode zur Evaluierung der Leistung verwendet. Die Visualisierung der Pixelwichtigkeit, auch Grad-CAM [Sel+17] genannt, wird als Qualitätsmessung für die syntaktischen Features verwendet, die vom Modell gefunden wurden. Die Experimente zeigten gemischte Ergebnisse für HTTP und FTP. Bezüglich HTTP wird deutlich, dass verschiedene Teile einer Protokollnachricht unterschiedlich hervorgehoben werden, siehe Abbildung 5.4. Die hervorgehobenen Teile stimmen oft mit syntaktisch relevanten Abschnitten überein, während pseudo-randomisierte Strings ignoriert werden. Dies ist genau die Art der syntaktischen Feature Extraction, die für dieses Experiment gewünscht wird.

Die Ergebnisse für das FTP-Protokoll sind in Grad-CAM weniger deutlich sichtbar, da die durchschnittliche Nachrichtenlänge deutlich geringer ist. Auch der Trainingsprozess konvergierte insgesamt flacher als bei der HTTP-Variante.

5.4.3 Feature Reverse Engineering

Ein zentrales Ergebnis des PRE ist die Darstellung des Zielprotokolls in Form von Regeln oder Clustern. Neuronale Netze erlauben jedoch keinen direkten Einblick in ihre interne Repräsentation der erlernten Features. Um ein greifbares Ergebnis zu erhalten, das über das Clustering und die Sequenzrekonstruktion hinausgeht, wird das Ziel verfolgt, neue Nachrichten generieren zu können – als Beleg dafür, dass das Modell die semantischen Strukturen tatsächlich erfasst hat.

Dazu werden generative neuronale Netzwerke genutzt und deren Ausgaben analysiert. Dabei stehen zwei Interpretationsansätze für Textnachrichten zur Verfügung: Entweder

werden sie als Byte-Folge mit bildähnlicher Struktur dargestellt – wie bereits bei der Feature Extraction verwendet – oder als sequenzielle Folge von ASCII-Zeichen.

Beide Varianten wurden erprobt. Für die bildähnliche Byte-Interpretation kam eine Standard-GAN-Architektur zum Einsatz, die sich an den Empfehlungen der Originalautoren orientiert [Goo+14; RMC15]. Für die sequenzielle Interpretation wurde ein LSTM-Modell verwendet, das um eine angepasste Embedding-Schicht ergänzt wurde, um die inhärente Zufälligkeit bestimmter Protokollbestandteile (z. B. Cookies, Adressen) realistisch abzubilden.

Generative Adversarial Net Ein GAN besteht aus zwei Netzwerken – dem Generator und dem Diskriminator –, die parallel zueinander trainiert werden. Der Generator verwendet vier 1D-Transponierte-Faltungsschichten mit folgenden Parametertupeln für (Kernelgröße, Schrittweite): (2, 2), (4, 4), (8, 8) und (16, 16) in aufsteigender Reihenfolge. Die ersten drei Schichten werden jeweils durch eine 1D-Batch-Normalisierung und eine ReLU-Aktivierung ergänzt, während die letzte Schicht mit einer Sigmoid-Aktivierung endet. Die Kanalanzahl ist wie folgt: $1024 \longrightarrow 1024 \longrightarrow 128 \longrightarrow 32 \longrightarrow 1$.

Der Diskriminator nutzt vier 1D-Faltungsschichten mit Batch-Normalisierung (außer in der ersten Schicht), Leaky ReLU mit einer Steigung von 0,2 sowie 20 % Dropout. Das Netzwerk endet in einer vollvernetzten Schicht mit 360 Neuronen. Die Kanalanzahl lautet: $1 \longrightarrow 10 \longrightarrow 20 \longrightarrow 60 \longrightarrow 90 \longrightarrow 1$.

Um ein Ungleichgewicht im Training zu verhindern, wurde eine Regel implementiert: Wenn der Fehler eines Netzwerks einen bestimmten Schwellwert überschreitet *oder* der Fehler des anderen Netzwerks unter einen festgelegten Wert fällt, wird das überlegene Netzwerk temporär vom Training ausgeschlossen, bis das andere aufholt.

Beide Netzwerke verwenden den Adam-Optimizer mit einer Lernrate von 0,0005 und den Parametern `betas` = (0,5, 0,99). Als Verlustfunktion kommt Binary Cross-Entropy (BCE) zum Einsatz.

Das Training des GAN-Modells zeigte keine signifikante Konvergenz zu einem stabilen Zustand. Die ungenaue Byte-zu-Bild-Interpretation begrenzt das Modell bereits von Anfang an und in Kombination mit der inhärenten Instabilität von GAN-Architekturen überrascht das enttäuschende Ergebnis kaum. Die Idee, Text aus einer kontinuierlichen Datenrepräsentation zu generieren, erschien zunächst vielversprechend – insbesondere bei der Betrachtung der statischen Struktur von Protokollen. Jedoch konnten nicht einmal die aufgefüllten Nullen konsistent reproduziert werden (Abweichungen von etwa ±3 ASCII-Werten). Weitere Experimente mit dieser Architektur wurden daher nicht durchgeführt.

Long Short-Term Memory LSTMs sind fortgeschrittene rekurrente Netzwerke, die auf Sequenzen fester Länge trainiert werden. Das Auffüllen mit Nullen ist hierbei jedoch ungeeignet, da es künstliche sequenzielle Abhängigkeiten einführen würde. Stattdessen wird ein Trainingsskript verwendet, das mehrere Protokollnachrichten verknüpft, sodass eine Sequenz entsteht, die mindestens viermal so lang ist wie die Zielsequenz. Anschließend

wird ein zufälliger Abschnitt der gewünschten Länge extrahiert. Dieses Verfahren kommt ausschließlich während des Trainings zum Einsatz, um echte Zeichen-zu-Zeichen-Abhängigkeiten zu vermitteln.

Vor und nach jeder Nachricht wird jeweils ein eindeutiges Symbol für „Start of Package (SOP)“ bzw. „End of Package (EOP)“ eingefügt, damit das Modell unterschiedliche Nachrichten voneinander unterscheiden kann. Die Daten werden als One-Hot-Encoding des ASCII-Zeichensatzes dargestellt.

Die Architektur beginnt mit einer Embedding-Schicht, gefolgt von einer Faltungsschicht mit einer Kernelgröße von 4 und einer Schrittweite von 4, um das Embedding anzupassen. Dieser Ansatz, hier als *Convolutional Embedding* eingeführt, dient als Mittelweg zwischen Zeichen- und Wort-basiertem Embedding, insbesondere bei Daten mit hoher zufälliger Zeichenvariation.

Die Dimension des Embeddings wird dabei mit der Feature-Länge getauscht: Die Faltungsschicht behandelt die versteckte Breite als Kanäle und die Feature-Länge als Bilddimension, wobei die Batch-Größe unverändert bleibt. Nach der Faltung wird diese Transposition rückgängig gemacht, sodass ein eingebetteter Tensor mit einem Viertel der ursprünglichen Länge entsteht. Dieser wird dann in ein einfaches, einschichtiges LSTM-Modell eingespeist.

Das Ergebnis durchläuft denselben Vorgang rückwärts: eine transponierte Faltung, gefolgt von einer vollvernetzten Schicht mit identischen Hyperparametern und erneutem Dimensionswechsel. Abbildung 5.5 zeigt den Aufbau dieses Embeddings.

Diese Art der Darstellung kann als lernbares, gewichtetes 4-Gramm interpretiert werden. Die Architektur muss jeweils nur das nächste Zeichen vorhersagen (Indexverschiebung von 0–1023 auf 1–1024). Die Kombination aus lokalem Kontext im 4-Gramm und den durch die Faltung verkürzten Langzeitabhängigkeiten erleichtert dem Modell das Lernen und verbessert die Stabilität bei der Generierung.

Für das Training wird der Adam-Optimizer mit einer Lernrate von 0,005 und Standardparametern verwendet. Als Verlustfunktion kommt Negative Log-Likelihood (NLL) zum Einsatz, da der Fehler auf Zeichenebene gemessen wird. Dies erfordert, dass die Eingabesequenzen eine Länge besitzen, die durch vier teilbar ist.

Dieser sequenzbasierte Versuch, HTTP- und FTP-Anfragen nachzubilden, zeigt gute Ergebnisse. Die LSTM-Architektur konvergiert nach weniger als einer Epoche auf einen minimalen Verlust, was auf exzellentes strukturelles Lernen und wiederkehrende Muster hinweist. Dies lässt sich durch die Natur eines textbasierten Protokolls wie HTTP und FTP erklären, das Schlüsselwörter, eine feste Grammatik und ein konsistentes Alphabet verwendet. Beim Abtasten des LSTM (Buchstabe für Buchstabe) wird eine Zeichenkette mit einer gültigen Nachricht erzeugt, die mit den `SOP`- und `EOP`-Symbolen geparst werden kann. Die resultierenden Anfragen werden mit gültigen, aber zufälligen TCP- und IP-Headern umschlossen, um vollständige Netzwerkpakete in einer pcap-Datei zu erzeugen. Das Netzwerk-Analysetool Wireshark[4] zeigte, dass 67,6 % der HTTP-Nachrichten gültig waren, während die restlichen als TCP mit zufälligem Payload klassifiziert wurden. Für FTP wurde

[4] `https://www.wireshark.org/`

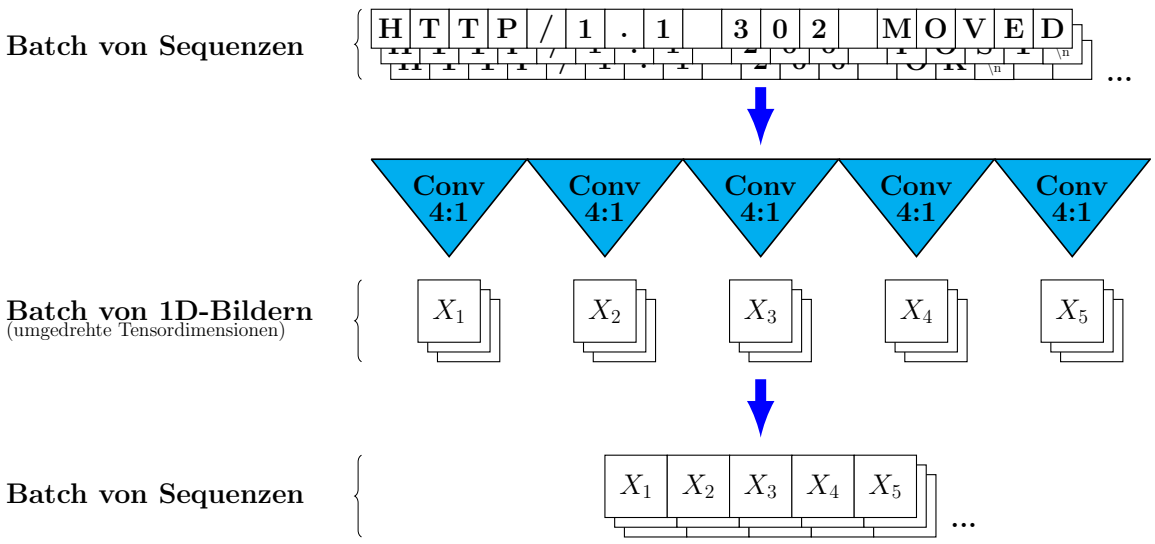

Abbildung 5.5: Eine Veranschaulichung des Prozesses von Convolutional 4:1, bei dem eine Reihe von Daten durch zwei unterschiedliche Faltungen geschickt wird.

```
HTTP/1.1 200 OK
Date: Mon, 14 Jun 2010 13:20:25 GMT
Server: Apache
Last-Modified: Mon, 21 Jun 2010 14:18:09 GMT
ETag: "2de9573-2b-486717fb77ac0"
Accepted-Ranges: bytes
Content-Length: 43
Connection: close
Content-Type: text/html; charset=iso-8859-1
```

```
HTTP/1.1 200 OK
Date: Mon, 14 Jun 2010 13:20:25 GMT
Server: Apache
Content-Length: 43
Connection: close
Content-Type: text/html; charset=iso-8859-1
```

Abbildung 5.6: Zwei von LSTM generierte HTTP-Anfragen. Während die Grundstruktur unverändert bleibt, wurden Inhalte und optionale Felder angepasst.

eine Quote von 100 % erreicht, allerdings können FTP-Nachrichten recht einfach sein, um gültig zu sein. Einige Nachrichten wiederholen sich häufig in den Trainingsdaten, was auch in der Ausgabe dieses Experiments zu sehen ist. Zwei generierte Beispiele sind in Abbildung 5.6 zu sehen. Diese Ergebnisse sind als eine solide Grundlage für weitere Experimente zu sehen.

5.4.4 Clustering

Für das Clustering sind die Ergebnisse der Feature Extraction relevant, um die Datensätze in ein kleineres Format zu kodieren. Drei SOMs wurden initialisiert und ihre Ergebnisse verglichen: ein Basismodell mit gekürzten und aufgefüllten Nachrichten auf eine feste Länge von 1024 Byte, ein zweites SOM-Modell unter Verwendung der AE-Kodierung und ein drittes Modell unter Verwendung der CNN Feature Map. Diese drei Modelle unterscheiden sich in ihrer Eingabedimension, haben aber alle eine identische Ausgabekartendimension von 16×1 für HTTP und 64×1 für FTP. Die unterschiedlichen Zahlen

Tabelle 5.2: Übersicht über die FTP-Cluster: Die FTP-Typen wurden manuell in Cluster von Schlüsselwörtern/Codes mit ähnlicher Bedeutung oder ähnlichem Zweck gruppiert

0	Sonstiges (MISC)
1	ACCT, ADAT, AUTH, CONF, ENC, MIC, PASS, PBSZ, PROT, QUIT, USER
2	230, 331, 332, 530, 532
3	PASV, EPSV, LPSV
4	227, 228, 229
5	ABOR, EPRT, LPRT, MODE, PORT, REST, RETR, TYPE, XSEM, XSEN
6	125, 150, 221, 225, 226, 421, 425, 426
7	ALLO, APPE, CDUP, CWD, DELE, LIST, MKD, MDTM, PWD, RMD, RNFR, RNTO, STOR, STRU, SYST, XCUP, XMKD, XPWD, XRMD
8	212, 213, 215, 250, 257, 350, 532
9	120, 200, 202, 211, 214, 220, 450, 451, 452, 500, 501, 502, 503, 504, 550, 551, 552, 553, 554, 555

für die Ausgabedimensionen für beide Protokolle basieren auf Experimenten und entsprechen im Großen und Ganzen der Vielfalt der verschiedenen Nachrichtentypen für jedes Protokoll. Dies kann für jedes neue Protokoll oder zu Optimierungszwecken durch Parameter angepasst werden. Das Training wird mit einer Lernrate von 0,005 und Sigma = 1,5 für HTTP und Sigma = 3 für FTP durchgeführt. Details zu den Parametern finden sich in der „MiniSom“-Bibliotheksdokumentation[5].

Für diese Auswertung müssen die tatsächlichen Klassen bzw. Typen für beide Protokolle im Voraus bekannt sein. Dies ist nicht einfach, da keines der Protokolle explizite Typen/Gruppen von Nachrichten festlegt, abgesehen von Anfragen und Antworten. Für HTTP haben insbesondere die Antworten eine breite Bedeutungsrange, die für diese Analyse nach der ersten Ziffer des jeweiligen Codes gruppiert sind, da diese eine grundlegende Bedeutung des Codes darstellen. Das bedeutet, dass sich alle Nachrichten mit den Nummern von 200 bis 299, alle von 300 bis 399 und alle von 400 bis 499 als denselben Typ betrachten lassen. Zusammen mit allen gültigen Schlüsselwörtern, mit denen eine HTTP-Nachricht beginnen kann (GET, POST, HEAD, DELETE, OPTIONS, PUT, TRACE, CONNECT), ergibt sich eine Gesamtzahl von 11 Clustern mit einem zusätzlichen „Sonstiges“-Cluster (MISC) für Nachrichten, die keinem anderen Cluster zugeordnet werden können. Für FTP wurden manuell Gruppen von Schlüsselwörtern und -codes mit ähnlicher Bedeutung oder ähnlichem Zweck definiert, wie in Tabelle 5.2 gezeigt. Dies war ein manueller Prozess, und bei dieser beispielhaften Gruppierung bleibt sicher Optimierungsbedarf.

Für das Experimentieren mit Clustering wurde ein Multi-Modell-Ansatz verwendet. Dabei werden drei verschiedene Konfigurationen von SOMs in Bezug auf ihre Leistung bei den Clustering-Metriken miteinander verglichen. Als erstes wird für die 128 Neuronen breite

[5] `https://github.com/JustGlowing/minisom`

SOM die Kodierung aus dem AE-Modell verwendet. Zweitens wird für die 240 Neuronen breite SOM die Feature Map der CNN-Architektur verwendet. Schließlich wird als Baseline eine rohe SOM mit 1024 Neuronen Breite als Vergleichsgrundlage verwendet.

Es wurden zwei Metriken verwendet, um die Effektivität des Clusterings für jedes Setup zu beurteilen. Die erste ist die Genauigkeit: Wie viele erkannte Cluster stimmen mit einem Nachrichtentyp des Protokolls mit mehr als 50 % Vertrauen überein, was als „dominantes" Cluster bezeichnet wird. Hier wird Vertrauen definiert als der relative Anteil eines Typs unter allen Nachrichten, die einem Cluster zugewiesen sind. Wenn es 120 Nachrichten vom Typ A und 80 Nachrichten vom Typ B gibt, die alle in ein Cluster gepackt wurden, dann wird das Vertrauen dieses Clusters, Typ A zu repräsentieren, mit $\frac{120}{120+80} = 60\,\%$ berechnet. Die zweite Metrik ist das durchschnittliche Vertrauen aller Cluster. Außerdem werden beide Metriken nur für dominante Typen (größer als 50 % für einen Typ) angegeben, um leere und sehr kleine Cluster aus dem Durchschnitt zu entfernen.

Tabelle 5.3 zeigt die Ergebnisse der Experimente. Es lässt sich festhalten, dass die Leistungsfähigkeit durch die Einbindung eines AE vor der SOM deutlich erhöht wird. Beim HTTP-Clustering steigt die Genauigkeit von 75 % auf 87,5 % und das durchschnittliche Klassifikationsvertrauen von 58,34 % auf 69,24 %. Ein vergleichbarer Effekt zeigt sich beim FTP-Clustering: Die Architektur AE + SOM erreicht 67,19 % bzw. 86 % im dominanten Cluster und übertrifft damit sowohl das Basismodell als auch CNN + SOM. Das durchschnittliche Vertrauen erhöht sich hierbei von 51,8 % auf 56,11 %. Einige Setups, die dedizierte Feature-Extraktoren verwenden, können die Baseline signifikant übertreffen. Der AE scheint besser für diese Aufgabe geeignet zu sein. Ein möglicher Grund dafür könnte sein, dass die CNN-Architektur einen Workaround mit Datenaugmentation benötigte, um überhaupt ein Training zu ermöglichen. Für weitere Experimente, die Cluster-Indizes erfordern, werden die AE- und SOM-Architekturen zu einer Pipeline kombiniert, um Nachrichten durch ihren Cluster-Index zu ersetzen.

5.4.5 Zustandserkennung

Um tiefere Zustände in einem Protokoll zu erkennen, werden die Korrelation von Nachrichtenfolgen verwendet, wie sie chronologisch im Datensatz erscheinen. Dafür wurden alle Nachrichten durch ihre zugewiesenen Cluster aus dem Clustering-Modell ersetzt. Ein einfaches LSTM mit passenden Dimensionen zum SOM-Ausgang reicht aus, um hochkorrelierte Sequenzen anzuzeigen, indem es das Vertrauen des Netzwerks für den möglichen nächsten Nachrichtentyp präsentiert. Die zentrale Idee des SOM-Lernens besteht darin, ähnliche Nachrichtentypen in der Nähe voneinander anzuordnen, sodass die Verwendung der MSE-Verlustfunktion eine Annäherung an den korrekten Nachrichtentyp im LSTM-Ausgang ermöglicht. Die Verwendung der Cross-Entropy (CE)-Verlustfunktion hat keine Konvergenz gezeigt. Für das Training wurde ein Adam-Optimizer mit einer Lernrate von 0,005 und Betas = (0,3, 0,9) verwendet.

Die Anordnung ist in Abbildung 5.7 dargestellt. Eine einfache Zeitreihenprognose auf Basis eines LSTMs liefert insbesondere für FTP vielversprechende Ergebnisse. Dort lassen sich

Tabelle 5.3: Ergebnisse der Clustering-Experimente im Vergleich. Gegenüber dem Basismodell lässt sich bei Verwendung des Autoencoders eine Verbesserung erkennen.

(a) HTTP-Clustering

Architektur	Genauigkeit (dominant)	Durchschnittliches Vertrauen (dominant)
Baseline SOM	75 % (75 %)	58,34 % (58,34 %)
CNN + SOM	68,75 % (68,75 %)	53,61 % (53,61 %)
AE + SOM	**87,5 %** (**87,5 %**)	**69,24 %** (**69,24 %**)

(b) FTP-Clustering

Architektur	Genauigkeit (dominant)	Durchschnittliches Vertrauen (dominant)
Baseline SOM	60,94 % (72,22 %)	51,8 % (61,4 %)
CNN + SOM	29,69 % (29,69 %)	18,19 % (18,19 %)
AE + SOM	**67,19 %** (**86 %**)	**56,11 %** (**71,82 %**)

implizite Protokollzustände zuverlässig modellieren. Bei 64 möglichen Clustern stimmt das LSTM in 42 % der Fälle mit dem tatsächlich folgenden Nachrichtentyp überein. Zum Vergleich: Eine zufällige Vorhersage hätte eine Trefferwahrscheinlichkeit von lediglich $1/64 \approx 1,56\,\%$.

Obwohl 42 % isoliert betrachtet nicht besonders hoch erscheinen, stellt dies im Kontext von Fuzzing, wo zahlreiche Versuche möglich sind, eine substanzielle Reduktion des Suchraums dar und erhöht die Effizienz zielgerichteter Testfälle deutlich.

Dieses Experiment wurde auch für das HTTP-Protokoll durchgeführt. Die Ergebnisse für HTTP-Protokolle sind allerdings weniger beeindruckend. Von den 16 implizierten Clustern wurden 72 % korrekt vorhergesagt. Diese Zahl ist deutlich höher als für die FTP-Protokolle, sie basiert allerdings auf der Vorhersage der durchschnittlichen Clusterzahl, die mit der Verzerrung des Datensatzes übereinstimmt. Anders ausgedrückt: Die Vorhersage gibt zwei oder drei abwechselnde Typen für die GET-Nachricht an, was die Vorhersagegenauigkeit höher treibt als bei anderen Vorhersagemustern. Der Datensatz konnte in diesem Experiment nicht für Klassen ausgeglichen werden, da vermieden werden sollte, durch zufälliges Auswählen von Nachrichten sequentielle Abhängigkeiten zu verändern. Infolgedessen weisen die Eingabedaten für HTTP eine erhebliche Verzerrung zugunsten der GET-Nachrichtentypen auf, wie es die Originaldaten auch tun, was letztlich dieses Verhalten des LSTM erklärt.

Die Ergebnisse in diesem Abschnitt verdeutlichen die zentralen Risiken beim Einsatz maschineller Lernverfahren. Dies gilt insbesondere, wenn unterschiedliche Embeddings und Modellansätze kombiniert werden und die Bewertung ausschließlich anhand einer vordefinierten Metrik erfolgt. Die Experimente mit dem zustandsabhängigen Protokoll

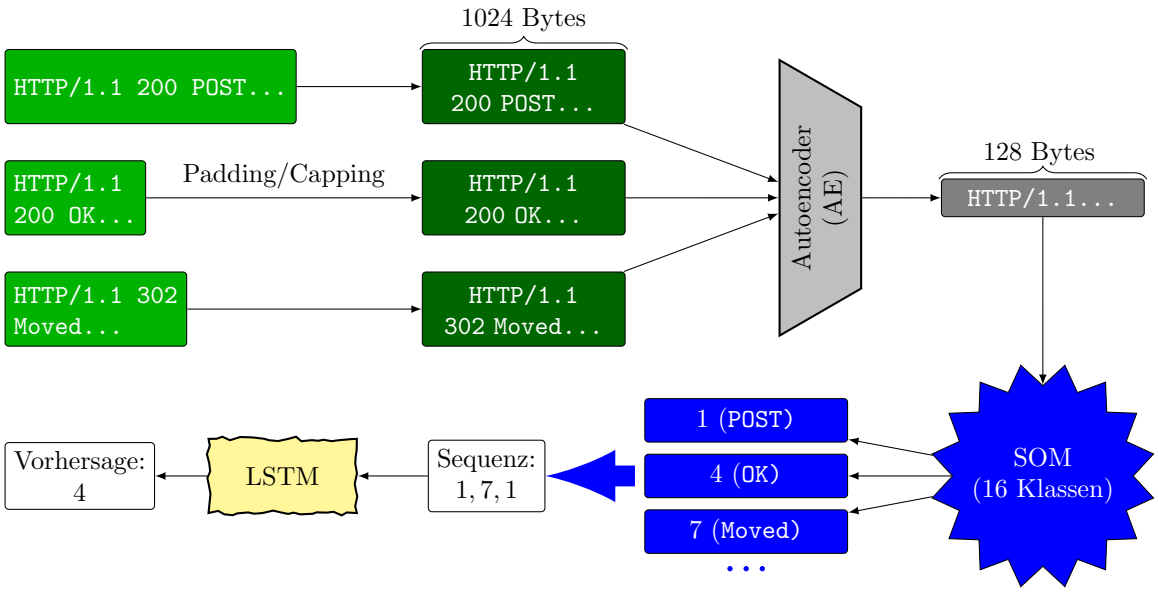

Abbildung 5.7: Beispiele von HTTP-Anfragen und ihrer Verarbeitung für die Zustandserkennung.

waren hingegen erfolgreich und erfüllten das ursprüngliche Ziel. Die Übertragung derselben Methoden auf ein zustandsloses Protokoll führte jedoch zu deutlichen Schwächen. Dies macht deutlich, dass die alleinige Betrachtung der Metrik zu Fehlschlüssen führen kann, da wichtige strukturelle Unterschiede der Daten unberücksichtigt bleiben. Die Analyse beider Protokolle zeigt, dass die Metrik auf den ersten Blick irreführend wirkt. Nur eine vertiefte, analytische Auswertung der tatsächlichen Vorhersageergebnisse ermöglicht es, die Diskrepanzen zu erkennen und die Ergebnisse korrekt zu interpretieren. Dieses Beispiel unterstreicht die Notwendigkeit, maschinelle Lernverfahren nicht isoliert anhand quantitativer Kennzahlen zu bewerten, sondern die Modelle und deren Vorhersagen stets im Kontext der zugrunde liegenden Datenstruktur zu analysieren.

5.4.6 Sequenzgenerierung

Um das Verhalten eines Protokolls vollständig nachzubilden, müssen syntaktisch korrekte Nachrichten mit einer realistischen Verteilung erzeugt werden. Dazu wird ein LSTM-Modell eingesetzt, das dem für das Feature Reverse Engineering verwendeten Modell ähnelt, jedoch um zusätzlichen Kontext erweitert wurde. Anstelle generischer `SOP`- und `EOP`-Symbole zur Markierung von Nachrichtenbeginn und -ende werden für jede Nachricht spezielle Symbole eingeführt, die nach Clustertyp differenziert sind. Dies führt zu 2×16 zusätzlichen Symbolen für HTTP und 2×64 Symbolen für FTP, die dem ASCII-Alphabet in den jeweiligen Setups hinzugefügt und anschließend mittels One-Hot-Encoding dargestellt werden.

Die dem LSTM zugeführten Sequenzen verwenden eine versteckte Größe von 100 Neuronen in zwei Schichten, um die zusätzliche Komplexität abzubilden. Innerhalb jeder Sequenz zeigt ein spezielles Cluster die Position von Nachrichtenbeginn und -ende durch

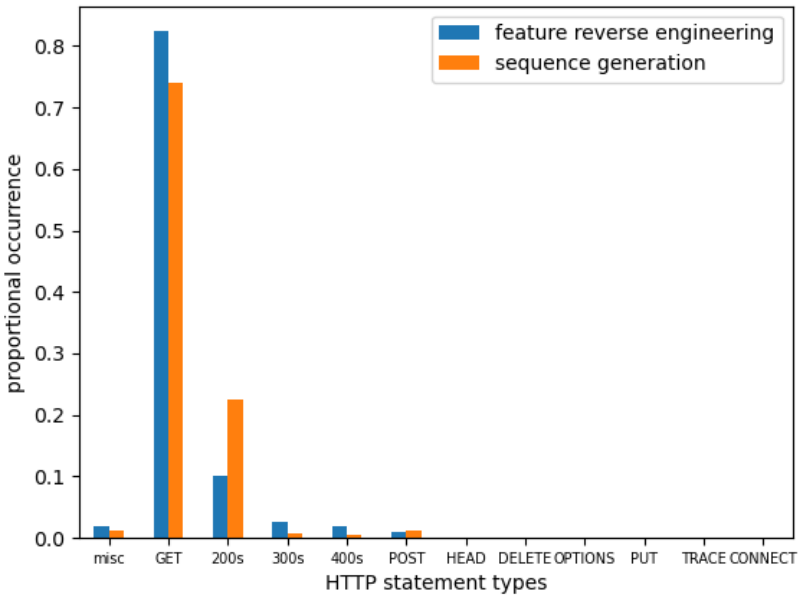

Abbildung 5.8: Ergebnisse hinsichtlich der Typverteilung für HTTP Feature Reverse Engineering (blau) und Sequenzgenerierung (orange).

die entsprechenden `SOP`- und `EOP`-Symbole an. Die übrige Architektur entspricht der des Modells für Feature Reverse Engineering, einschließlich der konvolutionalen Embeddings. Für das Training kommen der Adam-Optimizer mit einer Lernrate von 0,005 und den Standard-Betas sowie eine NLL-Verlustfunktion zum Einsatz.

Die beim Feature Reverse Engineering mit dem LSTM verwendete Metrik kommt auch hier zur Anwendung. Die generierten Sequenzen wurden in HTTP- bzw. FTP-Aussagen geparst, in gültige Protokoll-Header eingebettet und als pcap-Datei gesammelt. Eine anschließende Validierung mit Wireshark ergab, dass 63 % der HTTP-Sequenzen und 100 % der FTP-Sequenzen syntaktisch valide waren. Das hervorragende Abschneiden bei FTP erklärt sich, wie bereits beim Feature Reverse Engineering, durch die vergleichsweise einfache Struktur der FTP-Nachrichten und der Trainingsdaten.

Abbildung 5.9 zeigt die Typverteilungen für FTP sowohl beim Feature-Reverse-Engineering als auch bei der Sequenzgenerierung und visualisiert damit den Einfluss der Cluster-Indizierung auf die resultierende Verteilung. Auffällig ist eine dominante Klasse, die entweder auf ihre geringe Komplexität oder auf eine mögliche Fehlkategorisierung innerhalb der FTP-Typen zurückzuführen ist. Insgesamt deutet die Verteilung jedoch auf eine breitere und ausgewogenere Typaufteilung hin.

Die im Vergleich niedrigere Validierungsrate für HTTP geht mit einer stärkeren Korrelation zwischen Anfragen und Antworten einher (siehe Abbildung 5.8). Dieser Befund legt nahe, dass der zusätzliche Typkontext dem Modell dabei hilft, Anfrage-Antwort-Beziehungen zu lernen. Komplexere Abhängigkeiten im HTTP-Verkehr erschweren jedoch die vollständige

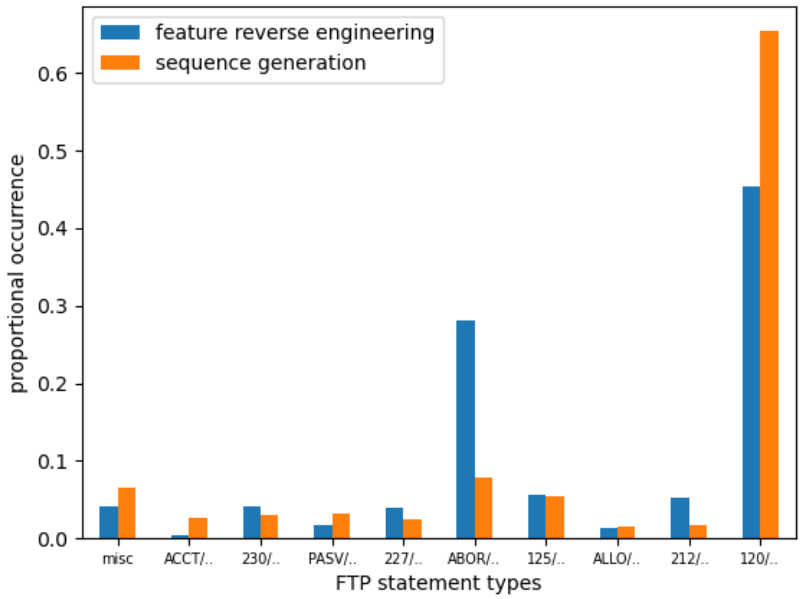

Abbildung 5.9: Ergebnisse hinsichtlich der Typverteilung für FTP Feature Reverse Engineering (blau) und Sequenzgenerierung (orange).

Reproduktion. Leistungsfähigere und größere Modellarchitekturen dürften die beobachtete Effektivität weiter steigern.

Diese Ergebnisse bilden die empirische Grundlage dafür, pro Protokoll ein Deep-Learning-basiertes Fuzzing zu realisieren, da der hier verfolgte PRE-Ansatz generalisierbar ist und die automatische Erzeugung syntaktisch valider Testeingaben ermöglicht.

5.5 Weiterentwicklung

Im Rahmen dieser Dissertation wurden zwei aufeinander aufbauende PREUNN-Ansätze entwickelt. Zur eindeutigen Unterscheidung wird der weiterentwickelte Ansatz im Folgenden sowie in der Evaluation als PREUNN2 bezeichnet. PREUNN2 wurde konzipiert, um die Leistungsfähigkeit des ursprünglichen Modells insbesondere für die Protokolle HTTP und FTP zu verbessern.

In diesem Abschnitt werden die konzeptionellen und technischen Erweiterungen von PREUNN zu PREUNN2 detailliert beschrieben. Abbildung 5.10 zeigt die Gesamtarchitektur des weiterentwickelten ML-PRE-Modells PREUNN2. Die einzelnen Komponenten der Architektur sind farblich gekennzeichnet.

5.5.1 Vorverarbeitung der Daten

Die ursprüngliche Vorverarbeitung in PREUNN ignoriert Protokollinformationen der Transportebene und trennt nicht zwischen Requests und Responses. Mithilfe der Aus-

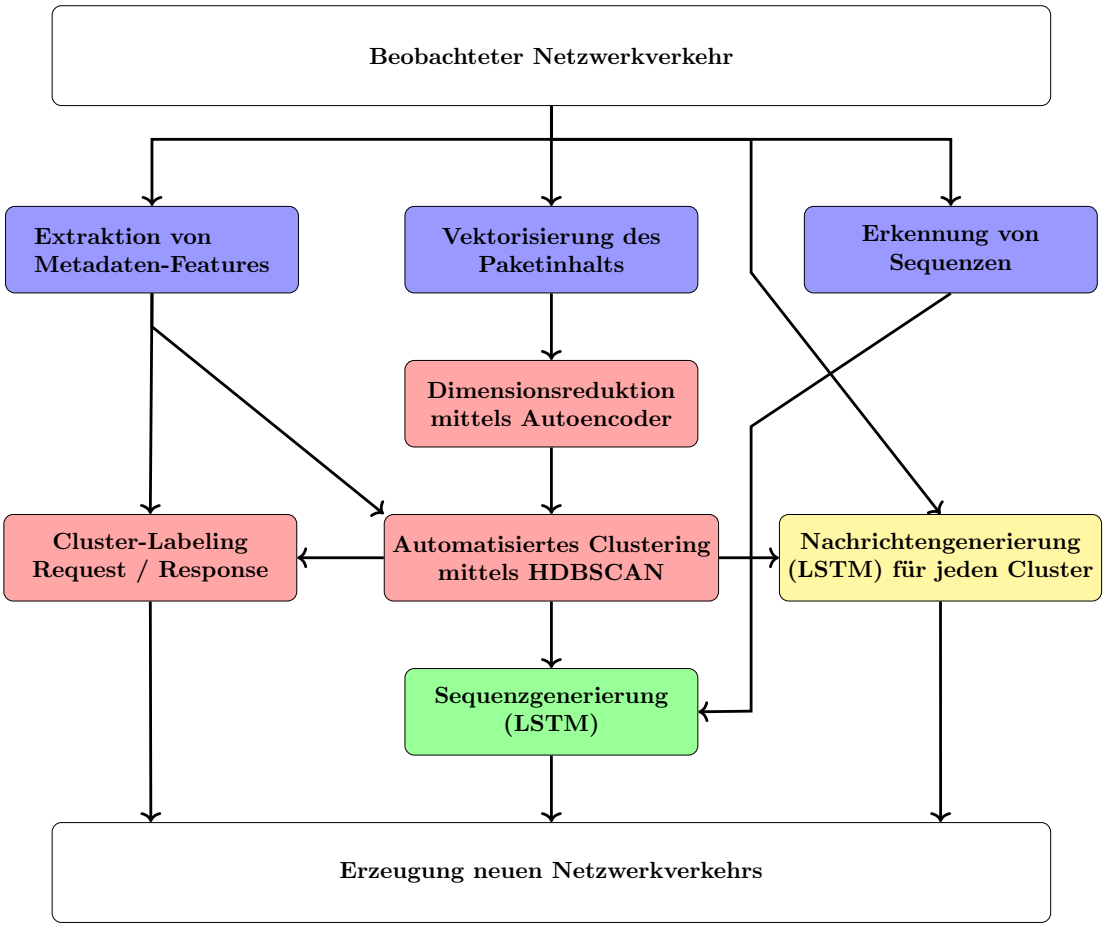

Abbildung 5.10: Architektur des weiterentwickelten PREUNN2-Modells. Vorverarbeitung (blau), Clustering (rot), Nachrichtengenerierung (gelb) und Sequenzgenerierung (grün).

wertung von TCP-Headerdaten wie Sequenznummern oder Ports sollen vollständige Kommunikationssequenzen erkannt und die Richtung der Pakete eindeutig bestimmt werden.

Ein zentrales Problem stellt der Balancing-Algorithmus dar, der auf manuell definierten Paketklassen basiert. Dies setzt tiefes Protokollwissen voraus und widerspricht dem automatisierten PRE-Ansatz. Zudem führt der aktuelle Algorithmus durch die strikte Begrenzung von Klassengrößen mitunter zum Verlust seltener, aber relevanter Pakete. Auch das Auffüllen mit Nullen auf Maximalgröße erzeugt künstliches Rauschen, da relevante Nutzdaten oft nur einen Bruchteil des Pakets ausmachen.

Schließlich erzeugt die bildbasierte Byte-Repräsentation eine Ähnlichkeit zwischen semantisch ungleichen Zeichen. Dadurch kann die Generalisierung des Modells erschwert werden, insbesondere in Kombination mit den Verzerrungen durch die aufgefüllten Nullen.

Zur Verbesserung und Abschwächung dieser Probleme werden relevante Metadaten bereits bei der Paketextraktion gespeichert und normiert. Anstelle einer manuellen Klassifizierung erfolgt eine clusterbasierte Typisierung der Pakete. Ein überarbeiteter Balancing-Ansatz berücksichtigt nun relative Clusteranteile und soll flexibler auf unterschiedliche Datenverteilungen reagieren. Die Optimierungen dieser weiteren Schritte werden in den folgenden Abschnitten detailliert beschrieben.

5.5.2 Klassifizierung von Nachrichtentypen und Zustandsübergängen

Die ursprüngliche PREUNN-Klassifizierung nutzt eine SOM, um Nachrichten zu Clustern zusammenzufassen, die den Protokollzuständen entsprechen sollen. Diese Methode zeigt jedoch Schwächen: Bei HTTP wurden 72 % der Zustände korrekt vorhergesagt, bei FTP lediglich 42 % (siehe Abschnitt 5.4.5). Ein Hauptproblem ist, dass viele Pakete demselben Cluster zugeordnet werden. Zudem werden in der Vorverarbeitung alle Nachrichten unabhängig vom TCP-Kontext betrachtet, wodurch Verbindungen nicht korrekt getrennt werden und falsche Sequenzabhängigkeiten gelernt werden.

Zur Verbesserung wird ein mehrstufiger Ansatz gewählt: Nachrichten werden tokenisiert, wobei häufige Token in einem Wörterbuch gespeichert und über Multi-Hot-Vektoren repräsentiert werden. Tokens am Anfang eines Pakets, die typischerweise den Nachrichtentyp kodieren, werden stärker gewichtet.

Anschließend reduziert ein AE die Vektordimension und filtert irrelevante Informationen [BKG20]. Für das anschließende Clustering wird HDBSCAN verwendet, das sich gegenüber klassischen Verfahren wie DBSCAN durch eine höhere Robustheit gegenüber unbalancierten Daten auszeichnet. Damit lassen sich strukturell ähnliche Nachrichten wie `PASS` und `PASV` zuverlässig unterscheiden [CMS13].

5.5.3 Erlernen des Nachrichtenaufbaus

Die in PREUNN eingesetzten LSTM-Netzwerke erzeugen syntaktisch korrekte Nachrichten und imitieren den Aufbau realer Pakete zuverlässig, insbesondere bei weniger

restriktiven Protokollen wie HTTP und FTP. Daher wird das LSTM-Modell auch in der Weiterentwicklung beibehalten.

Eine Schwäche des bisherigen Ansatzes liegt in der unausgewogenen Verteilung der Trainingsdaten: Durch den Balancing-Algorithmus werden nur ausgewählte Teile der Daten verwendet, wodurch seltene Nachrichtentypen vernachlässigt werden. Zudem erfolgte das Training nur in einer Epoche, was für LSTM-Modelle suboptimal ist.

Zur Verbesserung wird das LSTM nun epochenbasiert trainiert. Dabei werden in jeder Epoche neue, zufällig gesampelte Beispiele aus allen Clustern verwendet, um eine gleichmäßigere Abdeckung aller Nachrichtentypen zu gewährleisten. Der optimierte Clustering-Ansatz unterstützt dieses Sampling zusätzlich.

Als Eingabe erhält das Modell neben One-Hot-Vektoren für Zeichen auch den zugehörigen Clusterindex. Dadurch wird ein gezieltes Generieren von Nachrichten bestimmter Typen möglich.

5.5.4 Generierung neuer Testfälle

Die bisherige Sequenzgenerierung in PREUNN kombiniert LSTM-basiertes Lernen mit Cluster-Markierungen, um neue Paketfolgen zu erzeugen. Grundsätzlich lassen sich dadurch verschiedene Pakettypen korrekt kombinieren, es treten jedoch mehrere Probleme auf. So fehlen beispielsweise oft die gültigen Startpakete `USER`/`PASS`, da das Modell aufgrund der festen Eingabelänge von 1024 Bytes keinen sinnvollen Sequenzbeginn lernen kann. Zudem führen ungenaue Clusterzuweisungen und die fehlende Trennung von TCP-Streams im Training zu inkonsistenten Paketabfolgen. Eine Überrepräsentation häufig vorkommender Pakete wie `PORT` verschärft das Problem zusätzlich.

Auch die pcap-Generierung ist eingeschränkt. Da jede Nachricht eine eigene IP-Adresse und Port erhält, fehlt eine klare Zuordnung zu zusammenhängenden Kommunikationsströmen. Dadurch wird die Nachverfolgbarkeit und Analyse der generierten Sequenzen erschwert.

In PREUNN2 wird deshalb ein zweistufiger Ansatz verfolgt. Zunächst erzeugt ein LSTM auf Basis von Cluster-Indizes eine sinnvolle Paketreihenfolge. Anschließend generiert das bereits trainierte Nachrichtenmodell zu jedem Index ein konkretes Paket. Zwar lassen sich so keine inhaltlichen Abhängigkeiten zwischen den Nachrichten abbilden, doch solche Verknüpfungen traten im ursprünglichen Datensatz kaum auf.

Auch die pcap-Generierung wurde verbessert: Innerhalb einer Sequenz bleiben IP-Adressen und Ports konsistent. Mithilfe der Clusterinformationen kann außerdem automatisch zwischen Client- und Serverpaketen unterschieden werden. So entstehen strukturierte, realistisch wirkende Kommunikationsabläufe.

5.6 Implementierung von PREUNN2

Zur Evaluierung der zuvor identifizierten Optimierungspotenziale wird mit PREUNN2 ein alternatives ML-PRE-Modell realisiert. Es dient als Vergleich zur ursprünglichen PREUNN-Implementierung und basiert auf *TensorFlow*[6] mit *Keras*[7].

5.6.1 Vorverarbeitung

Die Pakete werden anhand von Zielports gefiltert, zu TCP-Strömen gruppiert und durch extrahierte Features wie Länge, Struktur, Richtung sowie Zeichenverteilung beschrieben. Eine gewichtete Tokenisierung bildet den Paketinhalt als Sparse Vector ab und ermöglicht eine differenzierte Eingaberepräsentation.

5.6.2 Clustering

Zur Dimensionsreduktion wird ein AE trainiert, dessen latente Repräsentation gemeinsam mit den Metadaten in ein Clustering überführt wird. Der HDBSCAN-Algorithmus gruppiert die Daten effizient und ohne Vorwissen in robuste Cluster, wobei auch seltene Pakettypen erfasst werden.

5.6.3 Generierung neuer Pakete und Sequenzen

Ziel der Nachrichtengenerierung ist es, syntaktisch korrekte Pakete auf Basis der Clusterstruktur zu erzeugen. Dazu wird ein LSTM-Modell mit zwei Eingaben verwendet: einem kontextbasierten Zeichenstrom und einem Vektor für den zugehörigen Clusterindex. Mithilfe dieser Informationen ist eine kontextsensitive Vorhersage des nächsten Zeichens möglich. Die Architektur kombiniert mehrere Schichten mit Dropout und nutzt eine Softmax-Ausgabe zur Byte-Wahrscheinlichkeitsverteilung.

Das Training erfolgt über einen Generator, der Paketsequenzen dynamisch erzeugt und mit `SOP`/`EOP`-Markierungen versieht. So können aus ca. 250 Clustern pro Epoche über 15 Millionen Trainingsbeispiele generiert werden. Zur Generierung neuer Nachrichten wird zunächst der Clusterindex mit einem `SOP`-Zeichen eingespeist. Anschließend wird iterativ ein Zeichen generiert, bis das `EOP`-Zeichen erreicht ist. Eine variable Sampling-Temperatur sorgt dabei für diversifizierte Ausgaben.

Auch für die Sequenzgenerierung kommt ein LSTM zum Einsatz. Dieser sagt auf Basis vorheriger Clusterabfolgen den nächsten Cluster voraus. Die Eingabe besteht aus One-Hot-Vektoren der Clusterindizes, ergänzt um `SOP`/`EOP`. Aufgrund der typischen Länge

[6] `https://www.tensorflow.org/`
[7] `https://keras.io/`

von Protokollkonversationen wird eine Kontextlänge, also die Anzahl der Input-Vektoren, von 20 verwendet. Das Modell erreicht nach 20 Epochen eine stabile Trainings- und Validierungsperformance (0,17 Categorical Cross-Entropy (CCE)).

Zur Erstellung von Fuzzing-Testfällen wird geprüft, ob die Cluster eher clientseitige oder serverseitige Nachrichten enthalten. Somit können bei der Generierung nur solche Pakete ausgewählt werden, die aus einem Cluster von Client-Nachrichten stammen.

5.7 Evaluation

Zur Bewertung des entwickelten Ansatzes kommt das Test-Framework *ProFuzzBench* zum Einsatz. Dieses ist ein Open-Source-Benchmark für das Fuzzing von Netzwerkprotokollen. Es unterstützt verschiedene Fuzzer, darunter auch *AFLNet* [NP21].

5.7.1 ProFuzzBench

Für jede Zielanwendung stellt das Framework ein Docker-Image bereit, das die Zielsoftware (mit und ohne Debug-Symbolen), den Fuzzer, vorbereitete Testfälle sowie ein Skript zur Steuerung des Fuzzing-Prozesses enthält. Dabei werden neue Testfälle erzeugt, die Anwendung überwacht und im Anschluss die Codeabdeckung aller Testfälle ausgewertet. Alle Daten werden automatisch erfasst und gespeichert.

Zusätzliche Skripte ermöglichen die parallele Ausführung mehrerer Fuzzing-Prozesse und die visuelle Auswertung der Abdeckungsdaten. Die Integration eigener Zielanwendungen ist durch die dokumentierte Struktur einfach umsetzbar.

5.7.2 AFLNet

Der mutationsbasierte Fuzzer *AFLNet* wurde gewählt, da dieser keine kontinuierliche Generierung neuer Testfälle zur Laufzeit erfordert und somit eine klare Trennung zwischen der Testfallerzeugung und dem eigentlichen Fuzzing-Prozess ermöglicht. Dies ist notwendig, um eine objektive Baseline-Messung durchzuführen, bei der das ML-Modell nicht in den Fuzzing-Prozess eingreift. Stattdessen werden vorab erzeugte Testfälle verwendet, die durch die Extraktion aus dem Netzwerkverkehr stammen [PBR20].

5.7.3 Integration der Machine-Learning-Methoden

Zur Erweiterung des Frameworks wird ML-gestützte Testfallgenerierung in den Fuzzing-Prozess integriert. Dabei kommt *AFLNet* sowohl für die Baseline als auch für die Evaluation zum Einsatz. Ausgangspunkt ist ein Datensatz mit TCP-Client-Paketen, aus dem einerseits direkt Testfälle extrahiert und andererseits ein ML-Modell trainiert werden.

Dieses Modell generiert äquivalente Eingaben, die anschließend ebenfalls durch *AFLNet* mutiert werden. So lassen sich mutations- und generationsbasierte Fuzzing-Vorteile kombinieren.

Es wurden die folgenden drei Integrationsansätze untersucht:

- **Vollintegrierte Generierung:** Bei der vollintegrierten Generierung werden Training und Testfallerzeugung zur Laufzeit im Docker-Container ausgeführt. Dadurch wird die komplette Laufzeit berücksichtigt und es entstehen unterschiedliche Modelle je Instanz. Der Aufwand ist jedoch hoch, insbesondere bei rechenintensiven Modellen wie PREUNN.
- **Externe Generierung:** Bei der externen Generierung werden der ML-Prozess und das Fuzzing vollständig getrennt. Testfälle werden einmalig vorab erzeugt und beim Start des Containers verwendet. Diese Methode ist ressourcenschonend, erlaubt komplexe Modelle und erfasst keine Trainingszeiten, was dynamische Anpassungen erschwert.
- **Integrierte Generierung mit Vortraining:** Bei der integrierten Generierung mit Vortraining werden vortrainierte Modelle im Container genutzt, die bei Bedarf weiter trainiert oder direkt zur Laufzeit verwendet werden können. Dieser Ansatz verbindet Flexibilität mit realistischer Ausführung und ermöglicht eine automatische Zeiterfassung bei hoher Modellkomplexität.

Aufgrund des besten Verhältnisses von Aufwand, Flexibilität und Praxisnähe wird der letzte Ansatz der integrierten Generierung mit Vortraining umgesetzt.

5.7.4 Implementierung der Fuzzing-Ziele

Für *ProFuzzBench* lassen sich neue Fuzzing-Ziele gemäß der Repository-Anleitung[8] integrieren. Im Folgenden werden zwei HTTP-basierte Ziele hinzugefügt: der Webserver *nginx* sowie eine gezielt verwundbare, minimalistische Serverapplikation auf Basis von *Express*.

5.7.4.1 Protokollimplementierungen für HTTP

HTTP wird von zahlreichen Serveranwendungen wie *Apache* und *nginx* unterstützt und bietet sich aufgrund seiner weiten Verbreitung und standardisierten Struktur als geeignetes Protokoll für das Benchmarking an. Diese gelten als ausgereift und gut getestet, weshalb Fuzzing bei ihnen voraussichtlich nur begrenzte Ergebnisse liefert. Zur Validierung dieser Annahme soll eine prototypische *nginx*-Implementierung eingesetzt werden. Zusätzlich wird eine eigens entwickelte, bewusst angreifbare HTTP-Implementierung verwendet, um gezielte Analysen zu ermöglichen und um das Serververhalten flexibel anpassen zu können.

[8] `https://github.com/profuzzbench/profuzzbench/blob/7779866f04/README.md`

nginx Der nginx Server kann korrekt instrumentiert und über angepasste Konfigurationen gesteuert werden. Erste Experimente zeigen jedoch, dass ein Großteil des Codes bereits beim Serverstart durchlaufen wird, wodurch spätere Fuzzing-Anfragen kaum zusätzliche Abdeckung erzeugen. Zudem erschwert die aufwändige Wiederinitialisierung pro Testfall eine präzise Auswertung. Als exploratives Fuzzing-Ziel eignet sich nginx daher nur bedingt, kann aber als realitätsnaher Referenzpunkt dienen.

Express Zur besseren Evaluierung des Fuzzing-Verhaltens wird eine eigene HTTP-Serveranwendung mit dem *Express*-Framework[9] entwickelt. Dieses erlaubt es, Endpunkte gezielt mit vordefinierten Fehlern und ungewöhnlichem Verhalten auszustatten, beispielsweise durch bewusste Abstürze, Timeouts oder inkonsistentes Antwortverhalten. Der gesamte Datenverkehr sowie auftretende Fehlerzustände werden protokolliert [Exp25].

Da eine klassische Instrumentierung bei JavaScript nicht möglich ist, wird die Codeabdeckung zur Laufzeit über externe Tools erfasst. Diese Messung erfolgt in Intervallen, um die Performance nicht zu beeinträchtigen. Die Anwendung lässt sich vollständig in ProFuzzBench integrieren und stellt eine leichtgewichtige, gut kontrollierbare Umgebung zur Analyse des entwickelten Fuzzing-Ansatzes dar.

5.7.4.2 Protokollimplementierungen für FTP

Auf verbreitete FTP-Server wie *ProFTPD* und *Filezilla Server* wird zugunsten einer leichter analysierbaren Alternative verzichtet. Stattdessen wird *LightFTP*[10] eingesetzt, da es in mehreren Studien [AC22; NP21; Nat22] als Fuzzing-Ziel dient, Open Source ist und vielfältige Konfigurationsmöglichkeiten bietet.

5.7.5 Auswertung der Ergebnisse

Zur Bewertung des ML-gestützten PREs wurden Fuzzing-Tests mit *AFLNet* auf den Zielen *LightFTP* und *Express* durchgeführt, um die Effizienz von PREUNN und PREUNN2 zu testen. Als Baseline dient eine zufällige Auswahl von 100 TCP-Sequenzen aus dem Trainingsdatensatz.

5.7.5.1 Fuzzing von Express

Beim Fuzzing des Express-Servers wurden je acht Instanzen pro Strategie zwölf Stunden lang ausgeführt. Aufgrund der geringeren Diversität der HTTP-Daten wurden pro Instanz nur 50 Testfälle generiert. Während PREUNN ursprünglich für HTTP entworfen wurde, musste PREUNN2 ohne spezielle Anpassungen auf das Protokoll generalisieren.

[9] `https://expressjs.com/`
[10] `https://github.com/hfiref0x/LightFTP`

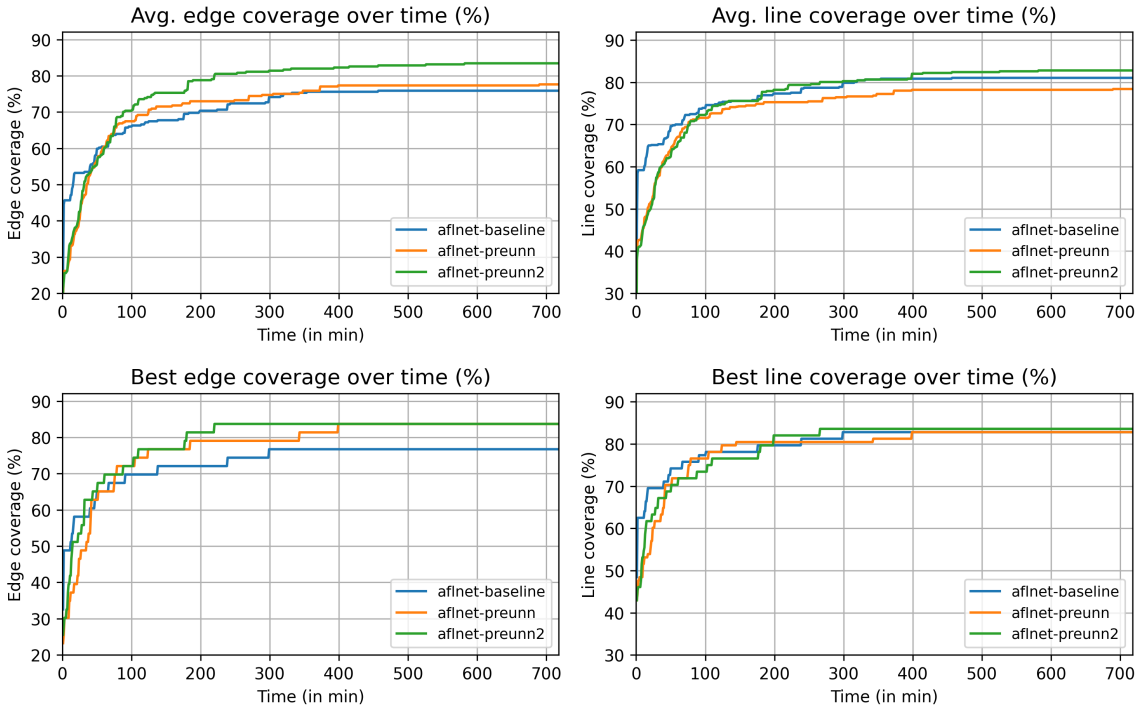

Abbildung 5.11: Verlauf der Codeabdeckung beim Fuzzing von *Express* mit den Testfallstrategien Baseline (blau), PREUNN (orange) und PREUNN2 (grün).

Trotzdem lieferte PREUNN2 durchweg die beste Codeabdeckung und in allen Instanzen zuverlässige Ergebnisse von über 80 %. Es war zudem der einzige Ansatz, der in jeder Instanz Abstürze und Hänger identifizieren konnte. PREUNN schnitt besser ab als die Baseline, erreichte jedoch weniger konstante Resultate als PREUNN2. In der Zeilenabdeckung lag die Baseline im Mittel leicht vorne.

Eine Analyse der gefundenen Abstürze zeigt, dass alle Strategien unterschiedliche Fehlerarten entdecken. PREUNN2 identifizierte den Absturz durch einen ungültigen Date-Header, die Baseline entdeckte einen Dateizugriffsfehler und nur PREUNN fand beide. Zwei weitere bekannte Abstürze wurden von keiner Methode erfasst – vermutlich aufgrund fehlender Repräsentation in den Trainingsdaten oder zu kurzer Laufzeit.

Insgesamt erzielte PREUNN2 die besten Ergebnisse. Der Test zeigt jedoch auch, dass eine hohe Codeabdeckung nicht automatisch zu einer besseren Fehlersuche führt. Am sinnvollsten erscheint daher eine Kombination aus beobachteten Netzwerkdaten und ML-generierten Testfällen.

5.7.5.2 Fuzzing von LightFTP

Für das LightFTP-Ziel wurden je acht Fuzzing-Instanzen mit den drei Testfallgenerierungen gestartet. Alle Versuche liefen parallel über einen Zeitraum von ebenfalls zwölf Stunden. Die Dauer der Testfallgenerierung hatte dabei keinen nennenswerten Einfluss auf die Ergebnisse.

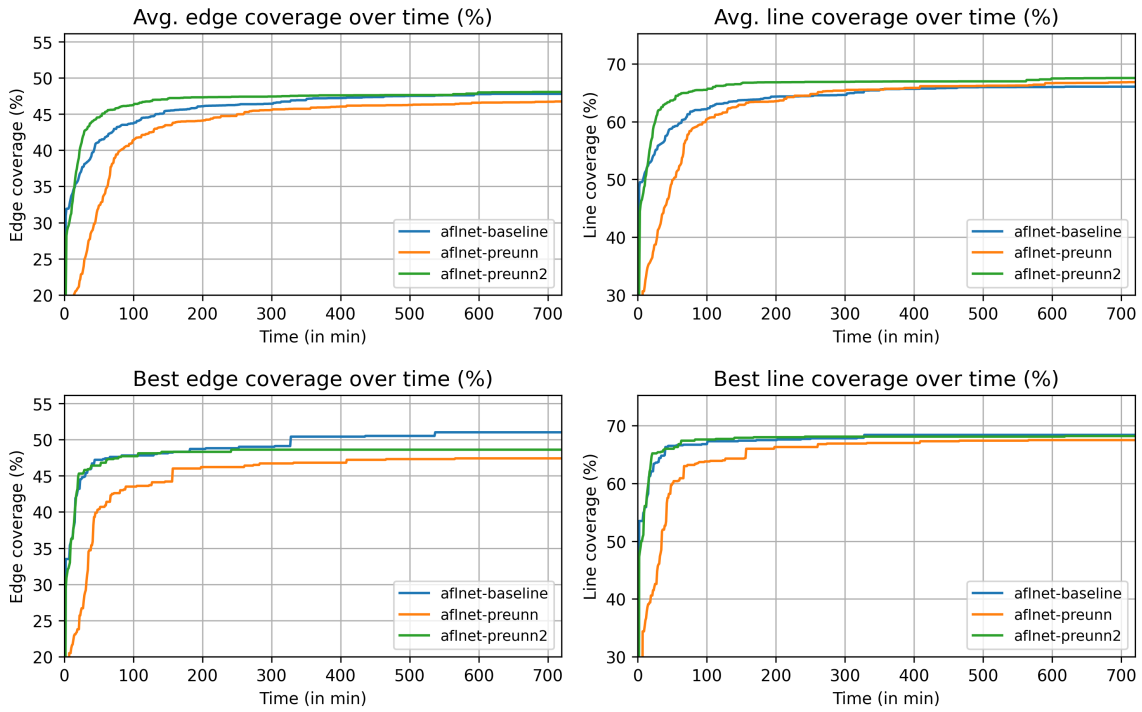

Abbildung 5.12: Verlauf der Codeabdeckung beim Fuzzing von *LightFTP* mit den Testfallgenerierungen Baseline (blau), PREUNN (orange) und PREUNN2 (grün).

PREUNN2 erzielte über die Laufzeit hinweg die stabilsten Resultate und erreichte im Mittel die höchste Codeabdeckung mit 48 % Zweig- und 67,6 % Zeilenabdeckung. In der Zweigabdeckung wurde es allerdings vereinzelt von der zufallsbasierten Baseline übertroffen, die mit 68,4 % auch den höchsten Einzelwert bei der Zeilenabdeckung erzielte. Insgesamt zeigte sich PREUNN2 jedoch effizienter: 95 % der maximalen Zweigabdeckung wurden in sieben von acht Instanzen erreicht, gegenüber nur zwei bei der Baseline und keiner bei PREUNN.

Keiner der Versuche führte zu echten Abstürzen oder Hängern. Auffällige Testfälle in der Baseline wurden vom Fuzzer fälschlich als Absturz klassifiziert, da sie besonders lange Antwortzeiten simulierten, ohne das Ziel tatsächlich zu blockieren.

Die ursprüngliche PREUNN-Implementierung lag in den meisten Metriken zurück. Ursache dafür waren schwächere Startsequenzen und weniger realistische Nachrichtenfolgen, die zu längeren Laufzeiten und geringerer Fuzzing-Effektivität führten. Dennoch erreichte PREUNN teilweise eine höhere Zeilenabdeckung als die Baseline, was auf die strukturierte Generierung zurückzuführen ist.

Ein klarer Sieger lässt sich nicht ausmachen. Während PREUNN2 konsistente Ergebnisse mit schneller Abdeckung liefert, erreicht die zufällige Baseline in Einzelfällen bessere Spitzenwerte. Eine kombinierte Strategie erscheint daher vielversprechend.

5.8 Zusammenfassung

Im Rahmen dieser Dissertation wurden zwei neuartige Ansätze zur protokollbasierten Rekonstruktion entwickelt. PREUNN und die darauf aufbauende Weiterentwicklung PREUNN2. Beide nutzen Deep-Learning-Architekturen, um automatisch die Struktur und Semantik von Netzwerkprotokollen wie HTTP v1.1 und FTP zu rekonstruieren und daraus realistische, kontextabhängige Testfälle zu generieren. Damit lösen sie das Problem, dass die Schritte nicht mehr manuell ausgeführt werden müssen und die Daten leichter für Fuzzing verwendet werden können. Somit ist sichergestellt, dass der Fuzzer gültige Testfälle verwenden kann und keine Zeit mit ungültigen Eingaben verschwendet, die direkt verworfen werden.

Die Ansätze kombinieren unterschiedliche Deep-Learning-Architekturen: Autoencoder für die Feature-Extraktion, LSTMs für das Reverse Engineering von Features und Zustandswahrnehmung, sowie Self-Organizing Maps für das Clustering. Durch die Integration von NLP-Techniken in PREUNN2 können zusätzlich semantisch korrekte und kontextabhängige Kommunikationssequenzen erzeugt werden. Der modulare Aufbau erlaubt auch die Nutzung moderner Architekturen wie BERT und eröffnet Perspektiven für den Einsatz von Reinforcement Learning, um automatisierte Fuzzer für beliebige nachrichtenbasierte Formate zu entwickeln.

Zur Evaluation wurde das ProFuzzBench-Framework erweitert, das mehrere Serverimplementierungen und Netzwerkprotokolle integriert. In den Fuzzing-Tests erzielten ML-generierte Testfälle eine höhere und stabilere Codeabdeckung als zufällig ausgewählte Sequenzen aus realen Netzwerkmitschnitten:

- **Express:** PREUNN2 erzielte durchweg die höchste und stabilste Codeabdeckung (>80 %) und identifizierte zuverlässig Abstürze und Hänger. PREUNN erzielte bessere Ergebnisse als die zufällige Baseline, war jedoch weniger konsistent als diese.
- **LightFTP:** PREUNN2 lieferte über alle Instanzen hinweg die stabilsten Resultate, mit durchschnittlich 48 % Zweigabdeckung und 67,6 % Zeilenabdeckung. Wenige Spitzenwerte wurden vereinzelt von der zufälligen Baseline übertroffen, insgesamt war PREUNN2 jedoch effizienter.

Die Analyse der generierten Sequenzen zeigt, dass 63 % der HTTP- und 100 % der FTP-Sequenzen als gültig erkannt wurden. Dies unterstreicht die Fähigkeit der Ansätze, realistische Protokollnachrichten zu modellieren. Begrenzungen ergeben sich vor allem durch Overfitting bei längeren Trainingsläufen, unbalancierte Trainingsdaten und eine begrenzte Cluster-Prägnanz.

5.9 Fazit

In diesem Kapitel wurde untersucht, inwiefern sich klassische Deep-Learning-Modelle wie AE und LSTMs für das PRE eignen. Die Ergebnisse zeigen, dass neuronale Netze Kommunikationsmuster erkennen, Protokollzustände modellieren und realistische Testfälle

generieren können. Insbesondere in Kombination verschiedener Architekturen konnten syntaktisch valide und semantisch konsistente Nachrichtenfolgen erzeugt werden. Dies stellt einen wichtigen Fortschritt für die gezielte Testfallgenerierung dar.

Allerdings zeigen die Experimente auch Grenzen: Klassische Deep-Learning-Modelle haben Schwierigkeiten, komplexe, hierarchisch strukturierte Daten über längere Sequenzen hinweg präzise zu erfassen. LLMs und Transformer-Architekturen bieten hier neue Möglichkeiten. Aufgrund ihrer Fähigkeit, Grammatik, Kontext und Abhängigkeiten in strukturierten Eingaben besser zu erfassen, eignen sie sich besonders für die Verarbeitung von Protokollen und Formaten, die sich durch eine klare, formale Struktur auszeichnen.

Daher wird im folgenden Kapitel untersucht, wie LLM-basierte Verfahren für die Generierung von Eingaben in Fuzzing-Szenarien eingesetzt werden können. Dies knüpft direkt an das Ziel dieser Dissertation an, automatisierte Verfahren zur effizienten Sicherheitsanalyse komplexer IoT-Systeme zu entwickeln.

6 Effizientes grammatikbasiertes Fuzzing mittels Large Language Models

Der Inhalt dieses Kapitels basiert auf einer gemeinsamen Veröffentlichung mit Ibrahim Mhiri, Akim Stark und Ingmar Baumgart. Teile der Ergebnisse wurden bereits in der unten genannten Publikation veröffentlicht. Es wird ein neuartiger Ansatz zur Kombination von LLMs mit grammatikbasiertem Fuzzing vorgestellt. Ziel ist es, die automatisierte Generierung zielgerichteter Eingaben effizienter zu gestalten und damit die Testabdeckung deutlich zu erhöhen.

- Ibrahim Mhiri, **Matthias Börsig**, Akim Stark und Ingmar Baumgart. „How to Train Your Llama – Efficient Grammar-Based Application Fuzzing Using Large Language Models“. In: Secure IT Systems: 29th Nordic Conference, NordSec 2024 Karlstad, Sweden, November 6–7, 2024 Proceedings. Hrsg. von Leonardo Horn Iwaya, Liina Kamm, Leonardo Martucci und Tobias Pulls. Bd. 15396. Lecture Notes in Computer Science. Karlstad,Sweden: Springer-Verlag, Jan. 2025, S. 239–257. isbn: 978-3-031-79006-5. DOI: 10.1007/978-3-031-79007-2_13 [Mhi+25].

6.1 Einleitung

Die Ergebnisse von PREUNN im letzten Kapitel zeigen, dass sich mithilfe von PRE realistische Eingaben für Netzwerkprotokolle erstellen lassen. Dadurch lassen sich die Grenzen klassischer, rein zufallsbasierter Fuzzing-Ansätze überwinden. Dennoch gibt es zahlreiche Szenarien, in denen hochstrukturierte Eingaben erforderlich sind und eine reine Protokollanalyse nicht ausreicht. Für diese Fälle hat sich grammatikbasiertes Fuzzing etabliert, das auf formalen Beschreibungen basiert und syntaktisch korrekte Testfälle generiert.

Jüngste Fortschritte im Bereich der LLMs eröffnen zudem neue Perspektiven. Modelle wie LLaMA2 [Tou+23] können komplexe Abhängigkeiten innerhalb strukturierter Daten erfassen und kontextbewusst neue Sequenzen generieren. Für das Fuzzing bedeutet dies einen Paradigmenwechsel: Anstelle manuell definierter Grammatiken können LLMs dynamisch Eingaben erzeugen, die sowohl syntaktisch gültig als auch semantisch konsistent sind. Dadurch lassen sich potenziell tiefere Zustandsräume erreichen und versteckte Schwachstellen identifizieren, die klassischen Methoden bislang verborgen geblieben sind.

Das Ziel dieses Kapitels besteht darin, das Potenzial von LLMs für die Generierung grammatikalisch korrekter Eingaben im Kontext des Fuzzings zu untersuchen. Aufbauend darauf lässt sich der Beitrag wie folgt zusammenfassen:

- **Adaption eines vortrainierten LLaMA-2-13B-Modells:** Ein vortrainiertes LLaMA-2-13B-Modell wurde mithilfe von Prompt-Tuning und Fine-Tuning an das XML-Format angepasst. Das Ziel bestand darin, Eingaben zu generieren, die sowohl syntaktisch als auch semantisch korrekt sind.
- **Integration von LLMs in AFL:** Das angepasste Modell wurde in den bekannten feedbackgesteuerten Fuzzer AFL integriert. Dadurch können die generierten Eingaben direkt in den Fuzzing-Prozess eingespeist werden, ohne dass manuelle Grammatikdefinitionen erforderlich sind.
- **Implementierung einer dynamischen Feedback-Schleife:** Es wurde ein kontinuierlicher Feedback-Mechanismus implementiert, der erfolgreiche Testfälle identifiziert und an das LLM zurückführt. Auf diese Weise kann das LLM während des Fuzzings weiter optimiert werden.
- **Evaluation und Ergebnisse:** Die Ansätze wurden anhand der XML-Parser `libxml2` und `TinyXML-2` evaluiert. Das LLM-basierte Fuzzing erzielte eine bis zu sechsmal höhere Codeabdeckung als ein reines AFL-Setup und übertraf einen klassischen grammatikbasierten Fuzzing-Ansatz (Nautilus in Verbindung mit AFL) um bis zu 50 %. Außerdem wurden drei neue Zeitüberschreitungen in `libxml2` entdeckt.

Die zentrale Herausforderung besteht darin, die Ausdrucksstärke generativer Modelle mit den spezifischen Anforderungen des Fuzzings in Einklang zu bringen. LLMs eröffnen zwar neue Freiheitsgrade bei der Sequenzgenerierung, gleichzeitig muss jedoch sichergestellt sein, dass die erzeugten Eingaben auch dem geforderten Format entsprechen. Die Evaluation zeigt, dass LLM-basierte Ansätze klassische grammatikbasierte Verfahren nicht nur ergänzen, sondern in vielen Fällen deutlich übertreffen können.

Dieses Kapitel trägt zum Gesamtziel der Dissertation bei, nämlich zur Effizienzsteigerung des Fuzzings durch die Erforschung KI-gestützter Methoden. Diese Methoden sollen die Automatisierung der Sicherheitsanalyse komplexer IoT-Systeme unterstützen und die Testabdeckung verbessern.

6.2 Stand der Technik

In diesem Abschnitt werden verwandte Arbeiten vorgestellt, um den Stand der Technik einzuordnen und die Unterschiede zum vorgestellten Ansatz herauszuarbeiten. Dieser entstand zeitgleich und unabhängig von mehreren anderen Projekten, die das Potenzial von LLMs für Fuzzing untersuchten, was die Aktualität und Vielfalt des Themas unterstreicht.

Hu et al. [HZY23] präsentieren *ChatFuzz* eine Erweiterung von Greybox-Fuzzern wie AFL++, die generative KI integriert. Das System nutzt das LLM ChatGPT, um XML-Eingaben zu generieren, die den Formatspezifikationen strukturierter Programme entsprechen. Fine-Tuning oder Prompt-Tuning werden jedoch nicht eingesetzt, was zu einer geringeren Testabdeckung führt. Der in diesem Kapitel vorgestellte Ansatz geht darüber hinaus, indem er verschiedene Tuning-Strategien gezielt untersucht und optimiert, um die Qualität der generierten Eingaben zu steigern.

Zhang et al. [Zha+24b] stellen *LLAMAFUZZ* vor und zeigen ebenfalls, dass LLMs in der Lage sind, strukturierte Eingaben für Fuzzing zu generieren. Ihr Fokus liegt auf dem Fine-Tuning von LLMs zur Mutation bestehender Seed-Eingaben, wobei AFL++ als Baseline dient. Im Gegensatz dazu fokussiert sich der in diesem Kapitel vorgestellte Ansatz nicht allein auf Mutationen, sondern trennt die initiale Eingabegenerierung durch das LLM klar von der nachgelagerten Mutation durch den Fuzzer. Zudem werden sowohl Prompt-Tuning als auch Fine-Tuning systematisch verglichen, um die effektivste Strategie zu ermitteln.

Xia et al. [Xia+24] beschreiben mit *Fuzz4All* ein generisches Fuzzing-Framework, das LLMs für unterschiedliche Zielprogramme nutzt. Es besteht aus einer Autoprompting-Phase und einer Fuzzing-Schleife, die Eingaben generiert. Das Ziel ist ein möglichst breiter Einsatz, etwa auch für das Testen von Compilern. Der in diesem Kapitel vorgestellte Ansatz zeichnet sich dadurch aus, dass er auf ein spezifisches Eingabeformat optimiert ist. Dadurch ist eine tiefere, an das Format angepasste Analyse möglich.

Zusammenfassend zeigt die Literatur, dass LLMs bereits auf unterschiedliche Weise in Fuzzing-Workflows eingesetzt wird. Der nachfolgend vorgestellte Ansatz erweitert den aktuellen Stand der Technik, indem er eine klare Trennung von Generierung und Mutation vornimmt, den Schwerpunkt auf die effiziente Erstellung strukturierter Eingabedaten legt und verschiedene Tuning-Methoden hinsichtlich ihrer Leistungsfähigkeit bewertet.

6.3 Entwurf

Der im Folgenden als How to Train Your Llama (HTTYL) bezeichnete Prototyp dient der systematischen Untersuchung unterschiedlicher Trainingsmethoden für LLMs sowie dem Vergleich ihrer Effektivität. Der Entwurf basiert auf einem spezialisierten Eingabegenerator für XML-Parser, dessen Ziel die Erzeugung semi-valider (syntaktisch gültiger, jedoch fehlerhafter) XML-Pakete unter Einsatz von ML-Techniken ist.

Als Fuzzing-Engine wurde AFL gewählt, da es eine kompakte Codebasis mit hoher Anpassungsfähigkeit verbindet und somit eine flexible Erweiterung ermöglicht. Im Gegensatz dazu integriert AFL++ zahlreiche Weiterentwicklungen, weist jedoch eine signifikant erhöhte Komplexität auf, wodurch die Implementierung eigener Modifikationen erschwert wird. Um den Fokus auf die Schnittstelle zwischen ML und Fuzzer zu legen, wurde daher AFL als Grundlage der experimentellen Umsetzung eingesetzt.

AFL steuert den Fuzzing-Prozess, übermittelt Eingaben an den XML-Parser, überwacht dessen Verhalten und protokolliert Abstürze, Zeitüberschreitungen sowie Anomalien. Seitens des ML wird ein LLM benötigt, das kontextfreie Grammatiken verarbeiten und komplexe Eingaben generieren kann. Hierfür wurde das frei verfügbare `LLaMA-2-13B-Modell` ausgewählt, das sich insbesondere durch die effiziente Generierung komplexer Strukturen auszeichnet.

Die Konzentration auf XML bietet mehrere Vorteile: Das menschenlesbare Format erlaubt eine unmittelbare Validierung der Ergebnisse, die strukturierte Natur erleichtert die Evaluation und Identifikation von Parser-Schwachstellen und die weite Verbreitung

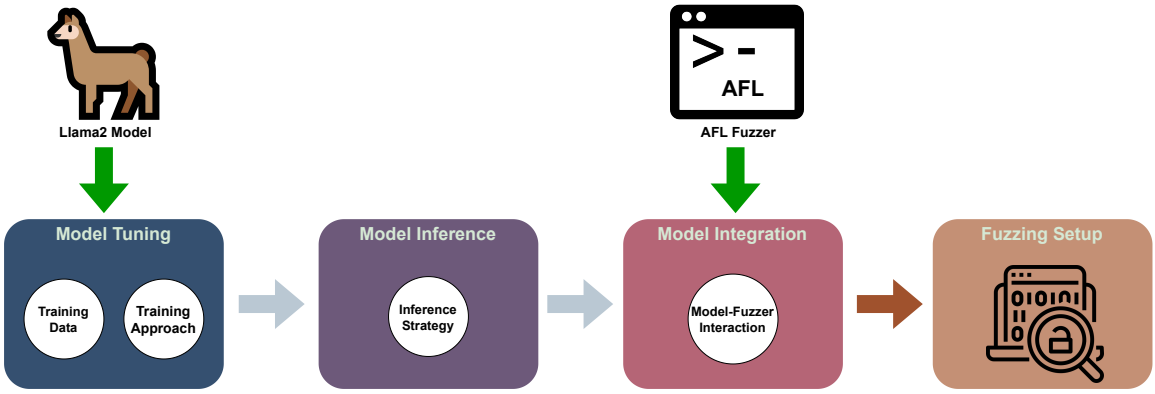

Abbildung 6.1: Übersicht über den vorgestellten Ansatz

gewährleistet vielfältige Testfälle. Zudem ermöglicht die formale Repräsentierbarkeit von XML-Grammatiken einen direkten Vergleich mit grammatikbasierten Fuzzern und schafft somit eine belastbare Benchmark zur Bewertung der Effektivität des Ansatzes. Die Abbildung 6.1 veranschaulicht das Konzept des Lösungsansatzes. Hierbei werden zwei Haupteingaben erfordert: ein LLM, im Beispiel Llama2, und das Fuzzing-Tool, in diesem Fall AFL. Die Methode besteht aus drei Phasen:

- **Modell-Tuning:** Zunächst wird die Fähigkeit des Modells verfeinert, die Eingabegrammatik des Zielprogramms zu verarbeiten, indem ein optimaler Datensatz und eine Trainingsmethode ausgewählt werden. Diese Phase resultiert in einem LLM, das auf die Verarbeitung von Eingabestrukturen abgestimmt ist, die für das Zielprogramm relevant sind, was seine Effizienz bei der Generierung sinnvoller Eingabevariationen erhöht.
- **Modell-Inferenz:** Nach dem Tuning wird diese Phase durchgeführt, um Testeingaben für den Fuzzing-Prozess zu erzeugen. Der Schwerpunkt liegt hier auf der Anwendung einer strategischen Inferenzmechanik, die die Erzeugung vielfältiger Testfälle ermöglicht.
- **Modell-Integration:** Dieser letzte Schritt stellt sicher, dass das Modell reibungslos mit dem Fuzzing-Tool zusammenarbeitet. Es wird ein einfacher Ansatz beschrieben, um das LLM direkt in den Fuzzing-Prozess einzubinden, mit dem Ziel einer nahtlosen Zusammenarbeit zwischen den beiden.

Dieser strukturierte Ansatz resultiert in einem umfassenden Fuzzing-Framework, das ein LLM enthält, das auf die Erzeugung gezielter Testeingaben abgestimmt ist, und eine AFL-Instanz, die auf die effektive Nutzung dieser Eingaben ausgerichtet ist.

6.3.1 Datensatz

Es wurde ein Datensatz zusammengestellt, der sowohl bösartige als auch harmlose XML-Dateien enthält. Insgesamt wurden 56 bösartigen XML-Dateien gefunden, die hauptsächlich XXE-Injection-Payloads enthalten und reale Anwendungsschwachstellen ausnutzen.

Sie wurden aus der Exploit-Datenbank[1] extrahiert. Für die harmlosen Dateien wurde das KIT-Motion-Language-Dataset[2] verwendet, aus dem zufällig 100 XML-Dateien ausgewählt wurden, um eine ausgewogene Mischung aus bösartigen und harmlosen Beispielen zu gewährleisten.

Obwohl der Trainingsdatensatz vergleichsweise klein ist, ist es herausfordernd, eine größere Vielfalt unterschiedlicher bösartiger XML-Dateien zu erhalten. Der geringe Anteil an harmlosen XML-Dateien sollte jedoch kein Problem darstellen, da Llama2 bereits auf großen Mengen öffentlich verfügbarer XML-Dateien aus dem Internet trainiert wurde. Dadurch verfügt das Modell über eine solide Grundlage für das Verständnis der XML-Struktur.

6.3.2 Trainingsansatz

Die Trainingsphase umfasst zunächst die Beschaffung von Daten, die den Zielen des Projekts entsprechen. Dazu wird das generative LLM eingesetzt, um AFL mit einer ausgewogenen Mischung aus harmlosen und bösartigen Beispielen zu versorgen. Somit wird eine gründliche Bewertung der Funktionalität des Zielprogramms ermöglicht. Nach der Vorbereitung des Datensatzes wird die optimale Methode zum Modell-Tuning ausgewählt. Da Fine-Tuning erhebliche Rechenressourcen erfordert, stellt Prompt-Tuning eine weniger ressourcenintensive Alternative dar, da hierbei nur ein Teil des Modells angepasst wird. Aufgrund dieser Überlegungen wird Prompt-Tuning priorisiert, während Fine-Tuning als robuste Vergleichsbasis dient.

6.3.3 Inferenzstrategie

Inferenzstrategien zur Erzeugung von Fuzzing-Beispielen fallen im Allgemeinen in zwei primäre Kategorien: Sampling-basierte Strategien, die den Eingaberaum mithilfe von Zufall erkunden, und deterministische Strategien, die einen vordefinierten Algorithmus oder eine Heuristik verwenden. Deterministischen Strategien, die für gegebene Token und Wahrscheinlichkeitspaare dieselben Sequenzen erzeugen, fehlt die notwendige Variabilität für eine effektive Fuzzing-Testung. Eine solche Vorhersehbarkeit, kombiniert mit Skalierungsproblemen im Zusammenhang mit Speicher und Ausführungszeit, macht deterministische Strategien für diese Anwendung unpraktisch. Der ausgewählte Ansatz ist daher die Top-k-Sampling-Methode, die zur Kategorie der Sampling-basierten Strategien gehört. Die Anpassungsfähigkeit der Top-k-Sampling-Methodeake macht sie ideal für die Erzeugung einer Vielzahl von Fuzzing-Eingaben, was eine umfassendere Bewertung der Schwachstellen des Zielprogramms ermöglicht. Diese Technik beinhaltet die Auswahl der Top-k-wahrscheinlichsten Token bei jedem Schritt, wodurch Zufall in den Prozess eingeführt wird, um eine vielfältige Menge generierter XML-Beispiele zu gewährleisten. Der

[1] `https://www.exploit-db.com/`

[2] `https://motion-annotation.humanoids.kit.edu/dataset/`

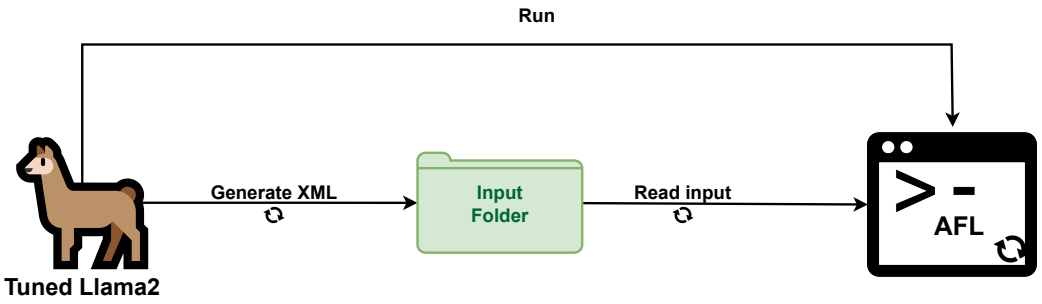

Abbildung 6.2: Integration des LLM in den Fuzzing-Test

Hauptvorteil dieser Methode besteht darin, dass sie ein Gleichgewicht zwischen der Erzeugung vielfältiger Eingaben und der Aufrechterhaltung eines effizienten und handhabbaren Auswahlprozesses aufrechterhält. Dieses Gleichgewicht ist für Fuzzing-Anwendungen von entscheidender Bedeutung, bei denen sowohl die Vielfalt der Eingaben als auch die Praktikabilität ihrer Erzeugung wichtig sind.

6.3.4 Modell-Integration

Nach der Vorbereitung des LLM für die Inferenz wird eine Integrationsstrategie beschrieben, die eine Einbindung des Modells in den Fuzzing-Test ermöglicht. Im Unterschied zu herkömmlichen Tools, die auf vordefinierten Grammatiken basieren, ist durch diese Integration ein direkter Zugriff auf generierte XML-Beispiele möglich, die in den Fuzzing-Prozess einfließen.

Hierzu wird AFL entsprechend angepasst, um eine nahtlose Übertragung der generierten Beispiele zu gewährleisten. Dies wird in Abschnitt 6.4 ausführlich beschrieben. Wie in Abbildung 6.2 dargestellt, erzeugt das Llama2-Modell eine vordefinierte Anzahl von XML-Beispielen. Diese werden direkt in ein Eingabeverzeichnis geschrieben, das von AFL für die Fuzzing-Tests verwendet wird. Der Vorgang erfolgt zyklisch, wobei während der gesamten Dauer des Fuzzing-Tests kontinuierlich neue Beispiele generiert und von AFL bewertet werden. Der Zyklus startet mit dem LLM, das AFL initiiert, um eine Fuzzing-Sitzung zu starten, die über einen definierten Zeitraum (z. B. 24 Stunden) läuft. Die für die LLM-Berechnungen benötigte Anfangszeit wird dabei nicht von der Fuzzing-Dauer abgezogen, da sie vorab berechnet werden kann. Nach Ablauf des Zeitraums wird die Beispielerzeugung beendet und AFL schließt den Vorgang ab. Dies kann entweder durch ein direktes Stopp-Signal des LLM oder durch einen vordefinierten Timeout-Mechanismus erfolgen, der zu Beginn der Tests festgelegt wurde.

6.3.5 Feedback-Schleife

Ein integrierender Bestandteil des Systems, wie in Abbildung 6.3 dargestellt, ist die Feedback-Schleife, die die Leistung des LLM über die Zeit verbessert. Während des Fuzzing-Tests

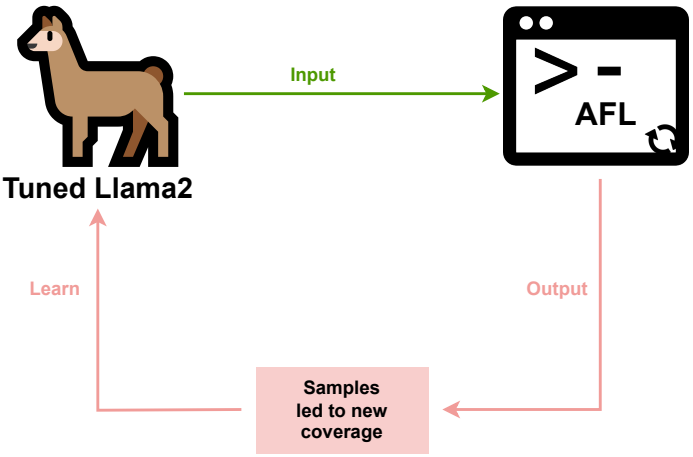

Abbildung 6.3: Llama2 lernt durch eine Feedback-Schleife

wird das LLM anhand der Rückmeldungen des Fuzzers angepasst. Die Feedback-Schleife analysiert die Ergebnisse von AFL, um XML-Beispiele zu identifizieren, die zur Entdeckung neuer Schwachstellen oder Pfade im Zielprogramm beitragen. Diese Beispiele werden erneut in den Trainingsprozess des LLM eingespeist, wodurch dessen Fähigkeit zur Generierung relevanter Testfälle kontinuierlich verbessert wird. Dieser iterative Zyklus ermöglicht eine fortlaufende Verfeinerung der Modellausgaben und eine Anpassung an die sich entwickelnde Landschaft potenzieller Schwachstellen in der Zielsoftware.

6.4 Implementierung

Es wurde ein PoC entwickelt, der die Synergien zwischen dem LLaMA-2-13B-Modell und AFL für ein erweitertes Fuzzing-Framework demonstriert. Der PoC zeigt die praktische Anwendung der Integration eines LLM mit einem führenden Fuzzing-Tool, um Software-Schwachstellen durch strukturierte XML-Eingaben aufzudecken.

6.4.1 Modelltraining und Integration mit AFL

Das LLM wird zunächst auf einem kuratierten Datensatz trainiert, um realistische XML-Eingaben für den Fuzzing-Prozess zu erzeugen. Sowohl Fine-Tuning als auch Prompt-Tuning kommen zum Einsatz, um ein Gleichgewicht zwischen Modellleistung und Rechenaufwand zu gewährleisten. Anschließend wird das trainierte Modell in AFL integriert, sodass generierte Testfälle automatisch in den Fuzzing-Zyklus eingespeist werden. Auf diese Weise kann das System kontinuierlich neue Eingaben erzeugen und direkt deren Effektivität bei der Aufdeckung von Schwachstellen evaluieren.

6.4.1.1 Fine-Tuning

Ein vollständiges Fine-Tuning des Llama2-Modells wurde durchgeführt, wobei das Modell aufgrund seines ausgewogenen Verhältnisses von Rechenaufwand und Leistung ausgewählt wurde. Der Fine-Tuning-Prozess erstreckte sich über drei Epochen bei einer Lernrate von 0,003, die empirisch bestimmt wurde, um ein Gleichgewicht zwischen einer schnellen Anpassung des Modells und dem Risiko von Overfitting zu gewährleisten. Das Training erfolgte anhand eines kuratierten Datensatzes, wie in Abschnitt 6.3.1 beschrieben.

6.4.1.2 Prompt-Tuning

Zusätzlich wurde ein Prompt-Tuning-Ansatz unter Verwendung der Transformers-Bibliothek[3] implementiert. Dieser Ansatz basiert auf demselben kuratierten Datensatz wie zuvor. Zur Evaluierung wurden zwei unterschiedliche Kontextlängen getestet: 3072 und 4096 Tokens. Während mit 3072 Tokens Eingaben bis zu dieser Länge verarbeitet werden können, ermöglicht die Einstellung von 4096 Tokens die Verarbeitung längerer Abfragen. Die größere Kontextlänge kann Ausgaben generieren, die potenziell detaillierter und konsistenter sind.

6.4.2 Kontinuierlicher Datenintegrationsmechanismus

Eine im Rahmen der Dissertation erstellte Erweiterung des AFL-Frameworks beinhaltet die Implementierung eines kontinuierlichen Datenintegrationsmechanismus. Dieser unterscheidet sich von traditionellen Fuzzing-Methoden. AFL kann somit neue Testfälle dynamisch während des Fuzzing-Prozesses anwenden, anstatt sich auf eine statische Eingabemenge zu verlassen.

Zu diesem Zweck wurde die Funktion `read_testcases()` von AFL so modifiziert, dass regelmäßig ein vordefiniertes Eingabeverzeichnis gescannt wird. Bereits verarbeitete Samples werden verfolgt und übersprungen, um Redundanzen zu vermeiden und die Effizienz zu steigern. Zusätzlich werden Testfälle, die neue eindeutige Pfade oder Programmabstürze erzeugen, sowohl im internen AFL Seed Pool als auch in einem vordefinierten Verzeichnis gespeichert. Dieses Verzeichnis dient als Repository für Beispiele, die später über den Feedback-Mechanismus zur Verbesserung des LLM verwendet werden.

6.4.3 Optimierungstechnologien

Zur Verbesserung der Skalierbarkeit und Effizienz kommen Optimierungstechnologien wie Accelerate [Gug+22], DeepSpeed [Ras+20] und Zero [Raj+21] zum Einsatz. Diese Technologien ermöglichen die schnelle Ausführung des LLM auf weit verbreiteter Hardware und steigern die Effizienz von Training und Nutzung großer neuronaler Netzwerke.

[3] `https://github.com/huggingface/transformers`

6.4.4 Dynamischer Feedback-Mechanismus

Ein dynamischer Feedback-Mechanismus überprüft die Ergebnisse jeder Fuzzing-Iteration. AFL verarbeitet die vom LLM generierten XML-Dateien und beginnt mit dem Fuzzing. Die Analyse bewertet die Effektivität verschiedener XML-Dateien, um diejenigen zu identifizieren, die neue Ausführungspfade oder Abstürze verursachen. Effektive Dateien werden an das LLM zurückgeführt, während ineffektive im AFL Seed Pool verbleiben, jedoch nicht für das Modelltraining genutzt werden.

Im nächsten Schritt wird das LLM auf den ausgewählten Samples prompt-getunt, um die Generierung neuer XML-Beispiele für nachfolgende Fuzzing-Tests zu optimieren. Dieser iterative Zyklus ermöglicht eine kontinuierliche Verbesserung der Testeffektivität. Gleichzeitig integriert AFL die neuen XML-Dateien in den Seed Pool und setzt den normalen Betrieb fort, wobei die Suche nach neuer Codeabdeckung aktiv weitergeführt wird.

6.5 Evaluation

Zur Bewertung des Ansatzes wurde der in Abschnitt 6.4 entwickelte PoC eingesetzt. Da die Rechenanforderungen für das Training und die Nutzung von LLMs sehr hoch sind, hatten die Entwicklungsumgebung und die Hardware einen entscheidenden Einfluss auf die Ergebnisse.

Verwendet wurde ein Server mit einer NVIDIA A100-Grafikkarte (80 GB VRAM), 720 GB RAM, 2,03 TB Speicherplatz und einem AMD EPYC 75F3-Prozessor. Jede Fuzzing-Sitzung wurde 24 Stunden lang durchgeführt. Dabei wurde der Zugriff auf GPU-Ressourcen ermöglicht und die Fuzzing-Testung auf einen einzelnen CPU-Kern beschränkt (Standard-Einstellung von AFL), um die Vergleichbarkeit zwischen den Läufen zu gewährleisten. In realen Szenarien stellen Ressourcen typischerweise keinen Engpass für Fuzzing-Tests dar. Das parallele Ausführen von AFL auf mehreren Kernen skaliert jedoch nicht linear, da Ergebnisse und Daten zwischen den einzelnen Instanzen geteilt werden müssen, was zusätzlichen Overhead verursacht.

Die Fuzzing-Einrichtung wurde gegen mehrere Zielprogramme getestet, darunter `libxml2` und `TinyXML-2`.

6.5.1 Bewertungsmetriken

Für die Bewertung wurden die von AFL bereitgestellten Metriken herangezogen:

- **Gesamtzahl der gefundenen Pfade:** Diese Metrik stellt die Gesamtzahl der eindeutigen Pfade dar, die während des Fuzzing-Tests innerhalb des Zielprogramms gefunden wurden.
- **Abstürze:** Dies sind eindeutige Testfälle, die zu fatalen Fehlern im getesteten Programm führen, wie z.B. SIGSEGV, SIGILL, SIGABRT usw.

- **Zeitüberschreitungen:** Zeitüberschreitungen sind eindeutige Testfälle, die dazu führen, dass die Rückmeldung des PUT zeitlich überschritten wird. Die Standard-Zeitbegrenzung von AFL vor einer Klassifizierung als Zeitüberschreitung beträgt eine Sekunde.

Zur Bewertung wurden zwei weit verbreitete XML-Parser `libxml2` und `TinyXML-2` herangezogen. Anstelle absichtlich eingeführter Fehler wurden reale Programme getestet, da künstliche Fehler die Subtilität echter Fehler nicht ausreichend abbilden. Wenn keine Fehler gefunden werden, was häufig der Fall ist, da Programme bereits gründlich getestet oder gepatcht wurden, dient die Codeabdeckung (Gesamtzahl der gefundenen Pfade) als zuverlässigster Indikator für die Tiefe der Programmuntersuchung.

6.5.2 Experimentelle Ansätze

Zur Evaluierung des Prototyps wurden mehrere experimentelle Ansätze getestet:

- **AFL-Fuzzing:** Dieser Ansatz verwendet die native Funktionalität des AFL-Fuzzing-Tools, wobei die Trainingsdaten für das LLM als Eingabe verwendet werden.
- **Vortrainiertes LLM-Fuzzing:** Diese Methode verwendet das unveränderte vortrainierte LLaMA-2-13B-Modell als Eingabegenerator für AFL
- **Fine-Tuned LLM-Fuzzing:** Diese Methode verwendet eine Fine-Tuned Version von Llama2, um Eingaben für AFL bereitzustellen.
- **Prompt-Tuned LLM-Fuzzing:** Dieser Ansatz verwendet ein Prompt-Tuned Llama2-Modell.
- **LLM-Fuzzing mit Feedback-Schleife:** Diese Methode verwendet ein Prompt-Tuned Llama2-Modell, um Eingabe-Samples für AFL bereitzustellen. Zusätzlich unterzieht sich dieses Modell während des Fuzzing-Tests einem Echtzeit-Prompt-Tuning mit Feedback vom Fuzzer. Vor allem Samples, die zur Entdeckung neuer Pfade führen, werden in das Prompt-Tuning mit einbezogen.
- **Nautilus[4] in Kombination mit AFL:** Nautilus 2.0, ein grammar-basierter Eingabegenerator, erzeugt syntaktisch korrekte XML-Daten, die anschließend von AFL ausgeführt werden. AFL misst dabei die Codeabdeckung und identifiziert Abstürze oder Hänger. Für die Experimente wurde das mit Nautilus bereitgestellte Beispiel einer XML-Struktur (`grammar_py_example.py`[5]) verwendet. Die Synchronisation erfolgt nur in eine Richtung: AFL kann Eingaben von Nautilus übernehmen, aber nicht umgekehrt.

Diese Ansätze erlauben den Vergleich der Effektivität und Effizienz unterschiedlicher Methoden innerhalb des Fuzzing-Frameworks.

[4] `https://github.com/nautilus-fuzz/nautilus`
[5] `https://github.com/nautilus-fuzz/nautilus/blob/mit-main/grammars/grammar_py_example.py`

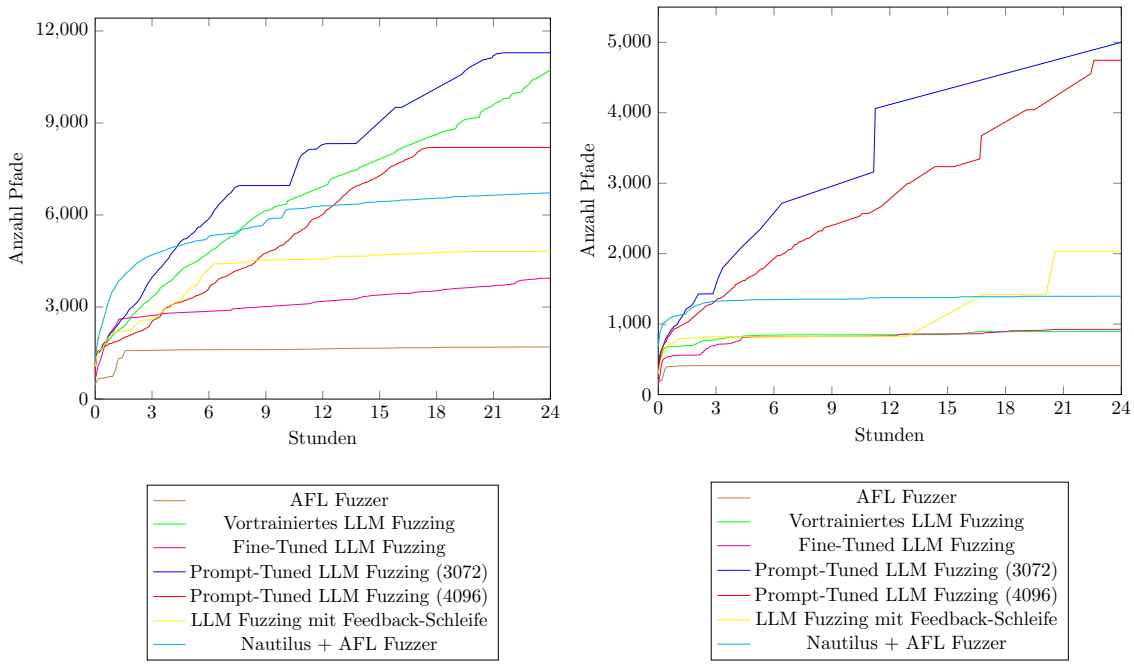

Abbildung 6.4: Gesamtzahl der gefundenen Pfade bei den verschiedenen Ansätze in `libxml2` (links) und `TinyXML-2` (rechts) über 24 Stunden

6.5.3 Experimentelle Ergebnisse

Abbildung 6.4 zeigt die Anzahl der vom Fuzzer gefundenen Pfade in `libxml2` und `TinyXML-2` über 24 Stunden. Die LLM-basierten Fuzzing-Methoden übertreffen den traditionellen AFL-Ansatz. Alle AFL-Varianten, die mit einem LLM integriert wurden, zeigten eine höhere Leistung als AFL allein, wobei Prompt-Tuned LLMs die beste Verbesserung erzielte.

Für `libxml2` wurden folgende Anzahlen an Pfaden gefunden: traditionelles AFL 1698, LLM-Fuzzing 10705, Prompt-Tuned LLM (mit 4096 Token Kontextlänge) 8203, Prompt-Tuned LLM (mit 3072 Token Kontextlänge) 11290, Fine-Tuned LLM 6719 und Nautilus + AFL 4826. Für `TinyXML-2` wurden folgende Ergebnisse erzielt: traditionelles AFL 411, LLM-Fuzzing 894, Prompt-Tuned LLM (mit 4096 Token Kontextlänge) 4745, Prompt-Tuned LLM (mit 3072 Token Kontextlänge) 5000, Fine-Tuned LLM 924 und Nautilus + AFL 1396.

Die vergleichsweise geringe Leistung des Fine-Tuned-Modells ist auf die nur drei Trainingsläufe zurückzuführen. Aufgrund der begrenzten Hardware-Ressourcen waren nicht mehr Trainingsläufe möglich. Prompt-Tuned-Modelle zeigten eine höhere Fähigkeit, neue Muster von XML-Dateien zu erlernen, insbesondere aus den bösartigen Beispieldateien. Dadurch verbesserten sich Abdeckung und Leistung. Eine Erhöhung der Kontextlänge kann die Leistung zwar weiter optimieren, führt jedoch unter Umständen zu einem Overhead, der die 24-Stunden-Performance beeinträchtigt, wie in Abbildung 6.4 gezeigt.

In den Tests zeigte die Kombination aus AFL und dem Prompt-Tuned LLM (3.072 Token Kontext) die beste Leistung. Das Feedback-Schleifen-Modell lag leicht darunter, da regelmäßige Trainingsphasen die Generierung von Samples unterbrachen. Die Unterschiede

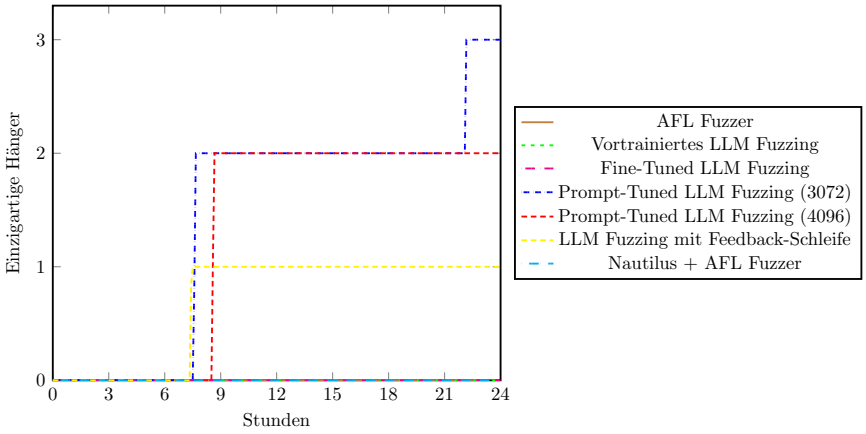

Abbildung 6.5: Gefundene Zeitüberschreitungen in `libxml2` für die verschiedenen Ansätze über 24 Stunden

Modell	Vortrainiertes Modell	Fine-Tuned Modell	Prompt-Tuned Modell	Feedback-Schleifen-Modell
Gesamtzahl der generierten Samples pro Lauf	6060 Samples	980 Samples	13980 Samples	2390 Samples
Generierungszeit eines Samples	14,04 Sekunden	89,49 Sekunden	5,99 Sekunden	-

Tabelle 6.1: Gesamtzahl der generierten Samples pro Fuzzing-Test

zwischen den Prompt-Tuned-LLMs mit 3.072 und 4.096 Token-Kontext lassen sich vermutlich auf den Overhead durch die größere Kontextlänge zurückführen, welcher die Performance über 24 Stunden beeinflusste.

Abbildung 6.5 zeigt die Anzahl der Zeitüberschreitungen, die während des 24-stündigen Fuzzing-Tests von `libxml2` auftraten. Die Prompt-Tuned-Fuzzing-Ansätze erzielten mit bis zu drei eindeutigen Zeitüberschreitungen die besten Ergebnisse. Das Feedback-Schleifen-Modell lag mit einer Zeitüberschreitung knapp dahinter. Traditionelles AFL-Fuzzing, vortrainiertes LLM-Fuzzing und das Fine-Tuned LLM-Modell führten dagegen während des gesamten Testzeitraums zu keinen Zeitüberschreitungen.

Für `TinyXML-2` wurden unabhängig vom eingesetzten Ansatz keine Zeitüberschreitungen festgestellt. Außerdem waren in beiden getesteten Programmen keine Abstürze zu beobachten.

Tabelle 6.1 zeigt die Gesamtzahl der generierten Samples pro Lauf sowie die für die Erstellung eines einzelnen Samples erforderliche Zeit der verschiedenen LLM-basierten Ansätze. Die Unterschiede in der Generierungszeit erklären die Abweichungen in der Gesamtzahl der Samples.

So generierte das vortrainierte Llama2-Modell 6.060 Samples innerhalb von 24 Stunden. Das Fine-Tuning-Modell erzeugte hingegen nur 980 Samples. Dies ist auf die erhöhte Komplexität und den zusätzlichen Aufwand durch das Fine-Tuning zurückzuführen. Mit 13.980 Samples erreichte das Prompt-Tuned-Modell (3072 Token Kontext) die höchste Anzahl und demonstrierte die Effizienz des Prompt-Tuning-Ansatzes. Dieser passt das Modell gezielt an einen Anwendungsfall an, ohne zusätzliche Komplexität einzuführen.

Top-k-Wert	5	25	50	150	250
Vortrainiertes LLM	13,12 Sek.	14,04 Sek.	14,34 Sek.	14,56 Sek.	14,68 Sek.
Fine-Tuned LLM	89,82 Sek.	89,49 Sek.	90,67 Sek.	89,52 Sek.	91,35 Sek.
Prompt-Tuned LLM	5,05 Sek.	5,99 Sek.	6,5 Sek.	6,97 Sek.	7,32 Sek.

Tabelle 6.2: Einfluss der Top-k-Variation auf die XML-Generierungszeit

Die Feedback-Schleifen-Variante erzielte eine geringere Anzahl an Samples, da das Modell während der Fuzzing-Tests regelmäßig Trainingsphasen durchlief, in denen die Generierung von Eingaben pausiert wurde.

6.5.4 Inferenzbewertung

Ein zentrales Merkmal der Inferenzbewertung ist die Laufzeit. Die erforderliche Zeit für jedes Modell, um eine XML-Datei zu generieren, wurde aufgezeichnet und in diesem Teil der Bewertung dargestellt. Die Tabelle 6.2 zeigt die für verschiedene Top-k-Werte erforderliche Zeit für jedes Modell, um ein XML-Beispiel in Sekunden zu generieren. Ein hoher Top-k-Wert kann die Generierungszeit von XML-Beispielen beeinflussen (mit leichten Variationen). Allerdings zeigt diese Tabelle auch ein interessantes Verhalten: die Generierungszeit von XML-Beispielen kann je nach Modelltyp (Fine-Tuned, Prompt-Tuned oder vortrainiert) erheblich variieren. Wie aus diesen Ergebnissen ersichtlich wird, ist Prompt-Tuned LLM das schnellste Modell, während das Fine-Tuned-Modell unabhängig von den Top-k-Werten die längste Zeit benötigt.

6.6 Einschränkungen und zukünftige Arbeiten

Es gibt mehrere Einschränkungen, die die Ergebnisse beeinflussen können:

- Neuere LLMs wie Llama3, Llama4 oder Code Llama könnten potenziell bessere Ergebnisse liefern, da sie auf umfangreicheren und aktuelleren Trainingsdaten basieren und über eine größere Anzahl an Parametern verfügen. Der Fokus dieser Dissertation lag jedoch auf der Machbarkeit und nicht auf der Optimierung der Leistung.
- Die Auswahl der verwendeten Sprachmodelle war durch die verfügbare Hardware begrenzt. Größere Modelle könnten eine höhere Testabdeckung und ein besseres Verständnis komplexer Eingaben ermöglichen.
- Das Trainingsdatenset bestand aus 56 bösartigen und 100 harmlosen XML-Beispielen. Eine Variation der Datentypen und des Verhältnisses von bösartigen zu harmlosen Beispielen könnte die Effektivität des Ansatzes steigern.
- Die gewählte Top-k-Sampling-Strategie minimiert die Sample-Generierungszeit, erfordert jedoch weitere Optimierung. Ein ausgewogenes Verhältnis von Gene-

rierungszeit und Sample-Qualität könnte durch Parameter-Tuning und alternative Inferenzmethoden erreicht werden.

- Die Feedback-Lernschleife beeinflusst die Anzahl der generierten Samples während der Fuzzing-Tests. Eine Optimierung über asynchrone Lern- und Generierungsprozesse könnte die Effizienz steigern.

Zusammenfassend verdeutlichen diese Ergebnisse das Potenzial der Integration von LLMs in Fuzzing-Methoden, weisen jedoch auch auf Bereiche hin, die für zukünftige Arbeiten weiter optimiert werden können.

Zukünftige Arbeiten eröffnen eine Vielzahl von Forschungsmöglichkeiten, die auf den Ergebnissen dieser Dissertation aufbauen. Mögliche Forschungsrichtungen umfassen:

- **Erforschung von KI-Alternativen:** Untersuchung der Machbarkeit alternativer KI-Modelle, wie z.B. GANs und Sequence to Sequence (Seq2Seq)-Modelle, zur Steigerung der Effizienz von Fuzzing-Tests. Zusätzlich könnte der Einsatz von LoRA die Fine-Tuning-Ergebnisse verbessern, da damit LLMs auch auf kleinen Datensätzen effektiv trainiert werden können, was in dieser Dissertation eine Einschränkung darstellte.

- **LLM-basiertes Mutation-basiertes Fuzzing**: Es könnte erforscht werden, wie sich LLMs nutzen lassen, um die Mutationsstrategien fortschrittlicher Fuzzer wie AFL zu ergänzen. Das Ziel besteht darin, das kontextuelle Verständnis von LLMs effektiv zu nutzen, um Mutationen zu steuern und somit die Testfallgenerierung und Schwachstellenentdeckung zu verbessern.

- **Domänen-spezifische Anpassung**: Die Anwendbarkeit des Ansatzes auf Eingaben jenseits von XML, insbesondere in Programmen mit domänenspezifischen Grammatiken, sollte untersucht werden. Dies erfordert eine Anpassung der Fine-Tuning-Phase unter Verwendung geeigneter Samples des Zielbereichs, etwa zum Lernen von PDF-Strukturen. Ziel könnte dabei die Erforschung geeigneter Formate und Methodologien für die Integration des Ansatzes in unterschiedliche Anwendungsdomänen sein.

Die Bewertung in Abschnitt 6.5 zeigt das Potenzial eines einfachen, vortrainierten Llama2-Modells, automatisierte Tests zu unterstützen und die Effizienz von Fuzzing-Tests für sicherheitskritische Anwendungen, wie beispielsweise XML-Parser, zu erhöhen.

Zukünftige Arbeiten könnten darüber hinaus die in Abschnitt 6.6 aufgeführten Einschränkungen adressieren. Die Nutzung aktueller LLM-Versionen, die Optimierung der Top-k-Sampling-Strategie, der parallele Einsatz zusätzlicher Hardware-Ressourcen sowie das Training auf größeren und diverseren Datensätzen könnten die Anwendbarkeit und Leistungsfähigkeit des vorgeschlagenen Ansatzes weiter verbessern.

6.7 Zusammenfassung

In dieser Dissertation wurde ein neuartiger Ansatz zur Integration von LLMs in Fuzzing-Workflows untersucht. Das Ziel bestand darin, die Effizienz und die Testabdeckung signifikant zu verbessern. Die Kombination eines Fuzzers mit einem LLM erwies sich als deutlich effektiver als der alleinige Einsatz klassischer Fuzzer und übertraf sogar grammatikbasierte Ansätze. Während traditionelle Fuzzer stark von Mutationen abhängig sind, generieren LLMs komplexe Eingaben auf Basis gelernter Muster und ermöglichen dadurch tiefere Einblicke in die Programmlogik.

Die Experimente konzentrierten sich auf XML-Dateien, da deren klar definierte Syntax eine systematische Testgenerierung erleichtert. Der in Abschnitt 6.3 vorgestellte Ansatz basiert auf der Evaluierung verschiedener Trainingsstrategien, darunter Fine-Tuning und Prompt-Tuning. Als leistungsfähigster Ansatz hat sich prompt-getuntes LLM-Fuzzing erwiesen, da die Grammatik implizit vom Sprachmodell erlernt wird und keine manuelle Spezifikation erforderlich ist. Dadurch ist die Generierung syntaktisch korrekter und semantisch aussagekräftiger Testfälle selbst für unbekannte oder komplexe Formate möglich. Die Anpassungsfähigkeit des Ansatzes erstreckt sich über XML hinaus auf Formate wie PDF, sodass vielfältige automatisierte Tests in unterschiedlichen Softwaresystemen möglich sind.

Der entwickelte Proof of Concept integriert das Llama2-Modell in das Fuzzing-Framework AFL. Tests mit den XML-Parsers `libxml2` und `TinyXML-2` zeigten eine deutlich höhere Code-Pfad-Abdeckung im Vergleich zu klassischen Methoden. Außerdem wurden drei bislang unbekannte Zeitüberschreitungen in `libxml2` entdeckt, was die Effektivität des Ansatzes bei der Aufdeckung neuer Schwachstellen bestätigt.

Durch die direkte Nutzung generativer Modelle lassen sich hochwertige Testfälle dynamisch erzeugen, wodurch sich die Effizienz und Genauigkeit klassischer Fuzzing-Methoden deutlich steigern lassen. Die Ergebnisse belegen, dass LLM die Entdeckung neuer Ausführungspfade beschleunigen, die Testabdeckung erweitern und bisher verborgene Fehler aufdecken können. Damit leistet diese Arbeit einen wichtigen Beitrag zur Weiterentwicklung automatisierter Sicherheitsanalysen und zeigt das Potenzial der Kombination klassischer Fuzzing-Techniken mit generativen KI-Methoden auf.

6.8 Fazit

Die Untersuchungen in diesem Kapitel haben gezeigt, dass sich LLMs für die Generierung hochstrukturierter Eingaben im Fuzzing einsetzen lassen. Durch die Anpassung eines vortrainierten LLaMA-2-13B-Modells auf XML-Daten und die Integration eines adaptiven Feedback-Mechanismus konnten syntaktisch korrekte und kontextbewusste Eingaben erzeugt werden.

Die größte Herausforderung bestand darin, die Balance zwischen der Flexibilität generativer Modelle und den Anforderungen des Fuzzings zu finden. Denn die erzeugten Sequenzen

mussten sowohl gültig als auch effizient für die Pfadexploration sein. Dies verdeutlichen die Ergebnisse: LLMs ergänzen klassische grammatikbasierte Ansätze sinnvoll und können in bestimmten Szenarien deren Reichweite übertreffen.

Aufbauend auf diesen Erkenntnissen wird im folgenden Kapitel ein theoretisches Konzept präsentiert, das die entwickelten Methoden zu einem einheitlichen Ansatz zusammenführt.

7 Ansatz für ein integriertes Fuzzing-Framework

In den vorangegangenen Kapiteln wurden vier eigenständige Ansätze entwickelt. Diese adressieren unterschiedliche Herausforderungen beim Fuzzing von IoT-Geräten und decken verschiedene Ebenen des Fuzzing-Prozesses ab.

- **ESP32 Binary Rewriting (EBR)** ermöglicht die gezielte Instrumentierung der Firmware direkt auf Binärebene, um beispielsweise Codeabdeckung oder Laufzeitschutzmechanismen zu erfassen (siehe Kapitel 3).
- **ESP32-QEMU-FUZZ (EQF)** stellt eine emulationsbasierte Testumgebung bereit, die den Zugriff auf interne Systemzustände erlaubt, die Auswertung beschleunigt und parallele Fuzzing-Durchläufe auf leistungsfähiger Hardware ermöglicht (siehe Kapitel 4).
- **Protocol Reverse Engineering using Neural Networks (PREUNN)** rekonstruiert unbekannte Netzwerkprotokolle automatisch und generiert syntaktisch valide Eingaben für Fuzzing-Tests (siehe Kapitel 5).
- **How to Train Your Llama (HTTYL)** nutzt Large Language Models, um variantenreiche und syntaktisch korrekte Testfälle für strukturierte Datenformate wie XML oder JSON zu erzeugen (siehe Kapitel 6).

Da diese Techniken komplementäre Aspekte des Fuzzing-Prozesses optimieren, bietet es sich an, ihre Interaktion in einem integrierten Ansatz zu betrachten. Ziel dieses Kapitels ist es daher, einen konzeptionellen Rahmen zu skizzieren, der die vorgestellten Verfahren modular kombiniert und flexibel an unterschiedliche Anwendungsszenarien anpasst. Zur besseren Verständlichkeit werden die einzelnen Ansätze im Weiteren einheitlich als *Module* bezeichnet. In dieser Dissertation wird keine praktische Implementierung vorgenommen. Stattdessen werden nur theoretisch die Machbarkeit und das Potenzial der Kombination aufgezeigt.

7.1 Konzeptionelle Integration der Module

Die vier entwickelten Module adressieren unterschiedliche Ebenen des Fuzzing-Prozesses und können flexibel miteinander kombiniert werden. Dabei steht die vollständige Modularität des Ansatzes im Vordergrund: Jedes Modul ist optional und kann je nach Anwendungsszenario individuell eingesetzt oder weggelassen werden.

7.1.1 Flexiblität der Module

Das zentrale Konzept dieses Ansatzes ist die Modularität. Jedes Modul bringt einen spezifischen Nutzen:

- **EBR:** Instrumentierung auf Binärebene, etwa zur Erfassung von Codeabdeckung oder zur Integration von Laufzeitschutzmechanismen.
- **EQF:** Emulationsumgebung mit Zugriff auf interne Systemzustände und Unterstützung paralleler Fuzzing-Instanzen.
- **PREUNN:** Rekonstruktion proprietärer Netzwerkprotokolle und Generierung syntaktisch valider Pakete.
- **HTTYL:** Grammatikbasiertes Fuzzing strukturierter Eingaben wie XML oder JSON mithilfe von LLMs.

Dank dieser Modularität können je nach Zielsystem, Datenformat und Testumfang maßgeschneiderte Fuzzing-Workflows definiert werden. Der Ansatz stellt somit keinen starren Framework-Entwurf dar, sondern eine flexible Sammlung kombinierbarer Bausteine.

7.1.2 PREUNN und HTTYL als parallele Module

PREUNN und HTTYL sind anwendungsspezifische Erweiterungen, die in der Regel parallel betrachtet werden sollten:

- **PREUNN** kann dann verwendet werden, wenn ein IoT-Gerät über proprietäre Netzwerkprotokolle angesprochen wird. Es ermöglicht die automatisierte Rekonstruktion der Protokollstruktur und generiert syntaktisch valide Netzwerkpakete für gezieltes Fuzzing.
- **HTTYL** kann verwendet werden, wenn das Zielsystem strukturierte Eingaben wie XML- oder JSON-Dokumente verarbeitet. Es erzeugt variantenreiche, syntaktisch korrekte Testfälle, die gezielt komplexe Parserlogik prüfen.

Ebenso denkbar ist ein kombiniertes Szenario, beispielsweise wenn ein IoT-Gerät ein proprietäres Netzwerkprotokoll nutzt, das eingebettete XML-Dokumente in bestimmten Feldern erwartet. In einem solchen Fall könnten PREUNN und HTTYL gemeinsam eingesetzt werden, um sowohl die Protokollstruktur als auch die eingebetteten Daten gezielt zu testen. In der Praxis ist es jedoch meist effizienter, diese Schritte nacheinander durchzuführen. Zunächst wird die Protokollebene vollständig abgedeckt und anschließend werden potenziell fehleranfällige Verarbeitungen strukturierter Eingaben gezielt geprüft.

7.1.3 Kombination der Module

Die flexible Kombinierbarkeit der Module ermöglicht die Realisierung verschiedener Fuzzing-Workflows. Ein Überblick ist in Abbildung 7.1 zu sehen. Besonders relevant sind zwei praxisnahe Integrationsszenarien:

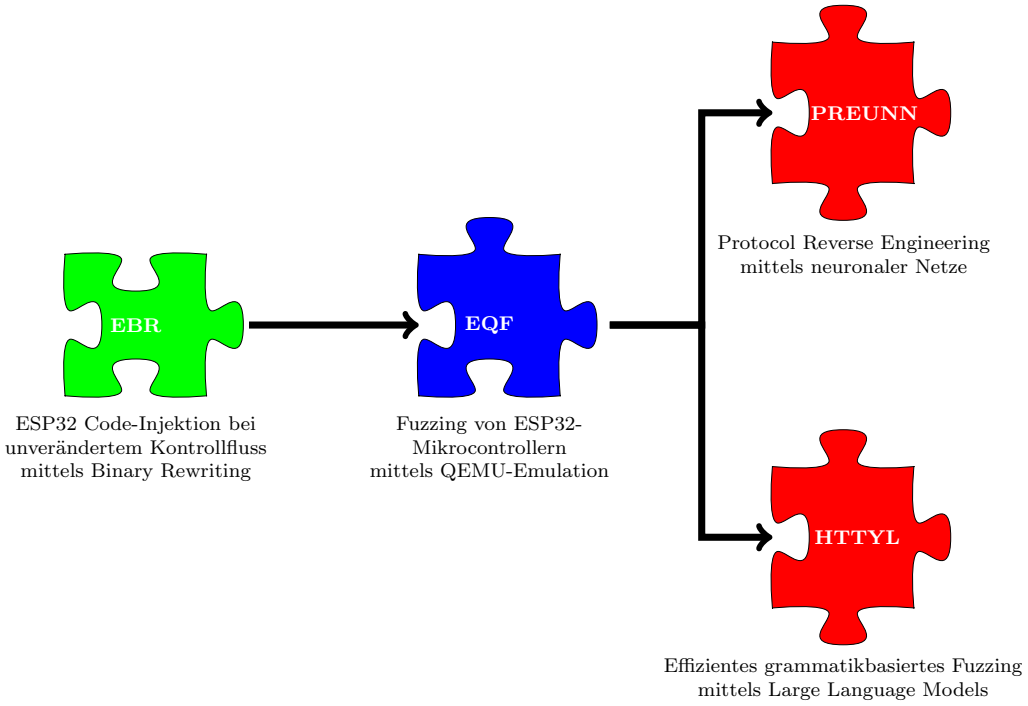

Abbildung 7.1: Modularer Aufbau des Fuzzing Frameworks

1. **EBR + EQF + PREUNN:** Dieses Setup ist geeignet, wenn das Zielgerät über ein unbekanntes oder proprietäres Netzwerkprotokoll kommuniziert. PREUNN erzeugt valide Netzwerkpakete, während EBR für gezielte Instrumentierung sorgt und EQF eine skalierbare Emulation auf leistungsfähiger Hardware ermöglicht.
2. **EBR + EQF + HTTYL:** Dieses Szenario ist geeignet, wenn die getestete Software strukturierte Eingaben wie XML oder JSON verarbeitet. HTTYL generiert syntaktisch korrekte Testfälle, EBR liefert tiefe Einblicke in die Codeabdeckung und EQF ermöglicht paralleles Fuzzing ohne physische Hardware.

In beiden Fällen können einzelne Module bei Bedarf weggelassen werden. So lassen sich beispielsweise EBR oder EQF auch unabhängig voneinander nutzen: EBR für instrumentiertes Fuzzing direkt auf realer Hardware oder EQF für emulationsbasiertes Fuzzing ohne Eingriff in die Firmware. PREUNN und HTTYL kommen nur dann zum Einsatz, wenn die Art der Eingaben dies erfordert.

7.2 Diskussion und Interpretation der Ergebnisse

Die vollständige Integration aller vier Module wurde im Rahmen dieser Dissertation nicht praktisch umgesetzt, da ihr Nutzen stark vom jeweiligen Anwendungsszenario abhängt und teilweise größere Anpassungen der einzelnen PoCs nötig wären (wie in Abschnitt 7.3

beschrieben). Die Machbarkeit und das Potenzial einer modularen Kombination wurden aufgezeigt, die umfassende Evaluation der konkreten Leistungsfähigkeit war jedoch nicht die Zielsetzung.

Die theoretische Analyse verdeutlicht, dass eine flexible Integration der entwickelten Module das Fuzzing von IoT-Geräten erheblich effizienter gestalten könnte. EBR ermöglicht eine präzise Instrumentierung der Firmware und liefert dadurch gezielte Laufzeitinformationen. EQF bietet eine skalierbare und performante Emulationsumgebung, die hohe Testdurchsätze erlaubt. Mit PREUNN lassen sich semantisch gültige Netzwerkpakete automatisiert generieren, während HTTYL strukturierte Eingaben wie XML-Dateien syntaktisch und semantisch korrekt erzeugt. Alle Module sind optional und können abhängig vom Anwendungsfall flexibel kombiniert oder weggelassen werden.

Die in dieser Dissertation präsentierten Ergebnisse und Kennzahlen basieren auf der Evaluierung einzelner Module. Eine experimentelle Validierung der vollständigen Systemintegration wurde bislang nicht durchgeführt. Dennoch lassen die Resultate eindeutig erkennen, dass die modulare Kombination der entwickelten Module das Potenzial hat, die Testabdeckung deutlich zu erhöhen, Fuzzing-Prozesse gezielter zu steuern und die Gesamteffizienz signifikant zu verbessern. Eine Kombination von EQF und HTTYL könnte für IoT-Geräte, die XML-Dateien verarbeiten, in einer virtuellen Umgebung im Vergleich zum herkömmlichen Blackbox-Fuzzing auf der ESP32-Plattform eine Effizienzsteigerung um den Faktor 50 ermöglichen (siehe Ergebnisse aus Abschnitt 4.5 und Abschnitt 6.5). Diese Verbesserung resultiert primär aus einer erhöhten Anzahl verarbeiteter Anfragen pro Sekunde sowie einer qualitativ hochwertigeren Eingabegenerierung. Durch zusätzliche Parallelisierung der Module ließe sich dieser Effekt voraussichtlich noch weiter verstärken.

Ein direkter Vergleich mit bestehenden Fuzzing-Frameworks erfolgt an dieser Stelle nicht, da keine existierende Arbeit alle vier betrachteten Module in vergleichbarer Tiefe integriert. Stattdessen werden im folgenden Kapitel (Kapitel 8) verwandte Arbeiten diskutiert, die einzelne Teilaspekte dieser Dissertation adressieren.

7.3 Limitationen

Trotz der vielversprechenden Ergebnisse des vorgestellten integrierten Ansatzes zur Sicherheitsanalyse von IoT-Geräten gibt es mehrere Einschränkungen, die die allgemeine Anwendbarkeit und Skalierbarkeit der Module begrenzen.

Die Untersuchungen wurden bewusst auf die ESP32-Plattform mit Xtensa-Architektur fokussiert, da alle entwickelten Patch-Methoden sowie das Binary-Rewriting-Framework speziell auf deren Befehlssatz, Speicherlayout und Toolchain zugeschnitten sind. Eine Übertragung auf andere Architekturen wäre zwar prinzipiell möglich, erfordert jedoch eine vollständige Anpassung der Instruktionsdekodierung, der Patch-Strategien und der Emulatorintegration.

Die einzelnen Komponenten wurden vorwiegend in Form von experimentellen Machbarkeitsstudien entwickelt. Die Verfahren sind komplex und müssen weiter ausgebaut werden,

um einen stabilen und praktischen Einsatz zu ermöglichen. So ist das ESP32 Binary Rewriting aktuell beispielsweise nur auf einen eingeschränkten Funktionsumfang begrenzt, da nur die am häufigsten verwendeten sowie die für das PoC notwendigen Instruktionen umgesetzt wurden. Dies wiederum schränkt die Möglichkeiten der Firmware-Instrumentierung ein. Für den produktiven Gebrauch ist eine umfassendere Unterstützung sämtlicher Firmware-Funktionen erforderlich.

Die Testbreite der angewandten Module ist ebenfalls begrenzt. PREUNN wurde ausschließlich an textbasierten Protokollen wie HTTPs und FTPs getestet. Ob es auf weit verbreitete binäre Protokolle wie MQTT übertragbar ist, bleibt offen. Ebenso wurde grammatikbasiertes Fuzzing mit LLMs nur mit XML-Daten geprüft. Dabei wurde die Annahme getroffen, dass sich die Methodik auf komplexere Formate wie PDF übertragen lässt. Diese Annahme könnte sich in der Praxis jedoch als nicht immer zutreffend erweisen, da unterschiedliche Dateiformate spezifische Anforderungen an die Modellierung stellen.

Außerdem bringen LLMs eigene Herausforderungen mit sich. So sind für das Training von LLM große Mengen hochwertiger Daten nötig, deren Qualität die Genauigkeit der generierten Testfälle und damit die Fuzzing-Effizienz direkt beeinflusst. Ein weiterer Punkt ist, dass sich die LLM-Forschung sehr schnell weiterentwickelt. In dieser Dissertation wurde Llama2 verwendet, das inzwischen um neuere Modelle ergänzt wurde. Fortschrittliche Trainingsmethoden wie Low-Rank Adaptation (LoRA) [Hu+22], die ein effizientes Training mit kleineren Datensätzen ermöglichen, wurden bisher ebenfalls noch nicht integriert. Sie bieten jedoch vielversprechende Ansätze für die Zukunft, um die Effizienz weiter zu steigern.

Schließlich ist auch die fehlende praktische Umsetzung des integrierten Ansatzes als Limitation dieser Dissertation zu verstehen, da die konkrete Implementierung stark vom jeweiligen Anwendungsszenario abhängt. Der Fokus lag auf der Darstellung der grundsätzlichen Machbarkeit und nicht auf einer umfassenden Evaluation der Leistungsfähigkeit, weshalb der theoretische Ansatz nicht praktisch evaluiert wurde.

8 Verwandte Arbeiten

Die Arbeiten, die bereits als direkte Grundlage für die vorgestellten Ansätze dienten oder methodisch eng verwandt sind, wurden in den jeweiligen Kapiteln behandelt. Beiträge, die später entstanden oder nur lose verwandt sind, werden hier gesammelt, um die wissenschaftliche Einordnung der Dissertation zu ermöglichen. In den letzten Jahren sind zahlreiche Arbeiten erschienen, die einzelne Aspekte dieser Dissertation berühren, etwa durch thematische Überschneidungen oder ähnliche Techniken, aber methodisch oft andere Wege verfolgen. Für die Einordnung dieser Dissertation wurden gezielt Beiträge berücksichtigt, die inhaltlich und methodisch anschlussfähige Weiterentwicklungen darstellen und so den Kontext und die Relevanz der vorliegenden Dissertation verdeutlichen.

Eine zentrale Inspirationsquelle für diese Dissertation ist die Untersuchung von Muench et al. [Mue+18], in der zentrale Herausforderungen beim Fuzzing eingebetteter Systeme identifiziert werden. Dazu zählen insbesondere das Fehlen vollständiger System-Emulatoren für die jeweilige Zielplattform und die Schwierigkeit, auftretende Fehler zuverlässig zu beobachten. Fuzzing in einem Emulator bietet den Vorteil einer transparenten Ausführung, wodurch sowohl die Fehlererkennung als auch die Erfassung der Codeabdeckung möglich wird. Auf diesen Beobachtungen basiert die Konzeption dieser Dissertation: So wurde das Binary Rewriting entwickelt, um gezielt Analyse- und Instrumentierungscode in Firmware einzufügen. Die QEMU-basierte Fuzzing-Emulation, wie sie in dieser Dissertation behandelt wurde, ermöglicht dagegen eine kontrollierte und messbare Ausführung auf der Zielarchitektur.

8.1 Binary Rewriting

Für das Binary Rewriting existiert eine Vielzahl von Tools, die für unterschiedliche Architekturen und Zielsetzungen entwickelt wurden. Einige Systeme, wie beispielsweise *Lancet* von Van Put et al. [Put+05], *Vulcan* von Srivastava, Edwards und Vo [SEV01] und *OM* von Wall und Srivastava [WS92] fokussieren sich auf Leistungsoptimierungen, etwa durch das Einfügen effizienterer Instruktionen oder die Umstrukturierung von Code. Andere wie *Zipr* von Hawkins et al. [Haw+17], *RevARM* von Kim et al. [Kim+17] oder *CFI CaRE* von Nyman et al. [Nym+17] zielen auf die Erhöhung der Sicherheit ab, beispielsweise durch das Hinzufügen von Schutzmechanismen oder Kontrollfluss-Integritätsprüfungen. Obwohl diese Ansätze unterschiedliche Ziele verfolgen, teilen sie mit dem hier vorgestellten Ansatz das grundlegende Prinzip, bestehenden Binärcode zu transformieren, ohne dessen wesentlichen Kontrollfluss zu verändern.

In der Literatur wird zwischen dynamischen und statischen Binary-Rewritern unterschieden. Dynamische Rewriter wie *Dynamo* von Bala, Duesterwald und Banerjia [BDB99], *STRATA* von Scott et al. [Sco+03] und *Pin* von Luk et al. [Luk+05] ändern den Code unmittelbar vor seiner Ausführung. Dies ermöglicht eine flexible Instrumentierung, erfordert jedoch eine kontinuierliche Anpassung zur Laufzeit und ist daher für Embedded-Plattformen wie den ESP32 ungeeignet.

Statische Rewriter hingegen arbeiten vollständig vor der Ausführung. Dabei lassen sich zwei Untergruppen unterscheiden: IR-basierte Tools wie *mctoll* von Yadavalli und Smith [YS19] und *revng* von Di Federico, Payer und Agosta [FPA17] übersetzen Binärcode in eine Zwischensprache wie LLVM IR, um Analysen und Optimierungen durchzuführen. Andererseits gibt es Disassembler wie ddisasm, Retrowrite und Uroboros, die den Rückbau in Assemblercode und den anschließenden Wiederaufbau zu einer modifizierten Binärdatei ermöglichen. Beiden Ansätzen ist gemein, dass die ursprüngliche Binärstruktur häufig nicht vollständig erhalten bleibt. Für sicherheitsrelevante Analysen, wie die in dieser Dissertation durchgeführten, ist jedoch eine möglichst hohe Strukturtreue entscheidend, um präzise und reproduzierbare Ergebnisse zu erzielen.

Umfassende Übersichtsarbeiten, etwa von Wenzel et al. [Wen+19] und Schulte et al. [SBF22], dokumentieren zwar ein reiches Ökosystem an Binary-Rewriting-Tools, doch wird darin deutlich, dass keiner der bestehenden Ansätze die Xtensa-Architektur unterstützt, die im ESP32 zum Einsatz kommt.

Der hier vorgestellte Ansatz schließt diese Lücke, indem er bestehende Binary-Rewriting-Methoden gezielt auf die Xtensa-Architektur überträgt und erweitert. Dabei bleibt der Kontrollfluss unverändert und die ursprüngliche Struktur der Binärdatei wird vollständig bewahrt. So entsteht einerseits eine klare Anschlussfähigkeit an bestehende Arbeiten und andererseits wird ein bislang ungelöstes Problem betrachtet.

8.2 Hardware Fuzzing von IoT-Geräten

Ein Ansatz, der direkt auf die Instrumentierung von Binärdateien abzielt, ist *QASan* von Fioraldi et al. [FDQ20]. Bei diesem Ansatz werden Binärdateien mit dem Quick Address Sanitizer (QASan) instrumentiert, einem Tool zur Laufzeitüberprüfung von Speicherfehlern wie Pufferüberlauf oder Use-After-Free. Dadurch wird ermöglicht, auch vorkompilierte Programme auf Speicherfehler zu testen, ohne dass der Quellcode vorliegt. Im Gegensatz dazu konzentriert sich diese Dissertation nicht auf reine Funktionen zum Schutz des Arbeitsspeichers, sondern auf die flexible Integration beliebigen Codes, um den Fuzzing-Prozess gezielt zu unterstützen.

Ein weiterer Ansatz, der Fuzzing auf unterschiedlichen Mikrocontrollern ermöglicht, ist *GDBFuzz* von Eisele et al. [Eis+23]. Hierbei werden Hardware-Debugschnittstellen wie GDB genutzt, um Codeabdeckung-Feedback aus uninstrumentiertem Firmware-Binärcode zu gewinnen. Während *GDBFuzz* auf reale Hardware und vorhandene Debug-Interfaces

angewiesen ist, erlaubt der in dieser Dissertation vorgestellte Ansatz die direkte Instrumentierung des Firmware-Codes. Dadurch entfällt die Abhängigkeit von externen Schnittstellen und eine flexiblere Analyse auf Mikrocontrollern wie dem ESP32 wird möglich.

Darüber hinaus geben verschiedene Übersichtsarbeiten einen umfassenden Überblick über bestehende Fuzzing-Techniken für IoT-Geräte. Maialen Eceiza-Olaizola et al. [EFI21] analysieren aktuelle Ansätze und identifizieren zentrale Herausforderungen beim Fuzzing eingebetteter Systeme. Touqir et al. [Tou+24] liefern eine systematische Übersicht über Fuzzing-Techniken in IoT-Umgebungen, bewerten deren Effektivität und zeigen bestehende Forschungslücken auf. Diese beispielhaft aufgeführten Arbeiten verdeutlichen die zunehmende Relevanz spezialisierter Fuzzing-Methoden für die Sicherheit vernetzter IoT-Geräte.

8.3 IoT Fuzzing mittels Emulation

Zum Zeitpunkt der in Kapitel 4 beschriebenen Arbeiten unterstützte die Standardimplementierung von ESP32-QEMU noch keine Emulation des integrierten WLAN-Subsystems. Inzwischen ist jedoch ein quelloffener Fork von QEMU[1] verfügbar, der durch Reverse Engineering der bislang undokumentierten WLAN-Register des ESP32 eine native Nachbildung dieses Subsystems ermöglicht. So ist es nun möglich, die ESP32-Firmware und die dazugehörigen Treiber in einer vollständig emulierten Umgebung auszuführen, ohne dass weitere Hardware-Peripherie emuliert werden muss. Diese Implementierung konzentriert sich allerdings ausschließlich auf die Emulation der ESP32-Firmware, während die vorliegende Dissertation den Schwerpunkt auf umfangreiche Fuzzing-Kampagnen in der emulierten Umgebung legt.

Einen verwandten, aber enger gefassten Ansatz als den in dieser Dissertation vorgestellten Emulationsansatz verfolgen Bogad und Huber [BH19] in ihrer Arbeit *Harzerroller*. Sie beschreiben, wie sie partielle Emulation einsetzen, um Firmware-Bilder des ESP8266 – dem Vorgänger des ESP32 – zu fuzzen. Dabei werden nur bestimmte Teile der Firmware in einer emulierten Umgebung ausgeführt, während andere auf der realen Hardware verbleiben. Dies ermöglicht die gezielte Untersuchung spezifischer Firmware-Bereiche, limitiert jedoch die Möglichkeiten zur vollständigen Automatisierung und umfassenden Erfassung der Codeabdeckung, wie sie in dieser Dissertation angestrebt werden.

Mehrere Übersichtsarbeiten zum Fuzzing mittels Emulation von IoT-Geräten liefern eine wichtige Grundlage zur Einordnung des in dieser Dissertation vorgestellten Ansatzes. Li et al. [LZZ18] liefern eine umfassende Übersicht zu Methoden der Schwachstellenentdeckung mit besonderem Fokus auf feedbackgesteuertes Fuzzing. Ihre Analyse zeigt die Wirksamkeit dieser Technik in verschiedensten Anwendungskontexten und unterstreicht damit die Relevanz einer präzisen Messung der Codeabdeckung für den Erfolg von Fuzzing-Kampagnen. Der in dieser Dissertation vorgestellte Ansatz greift diesen Befund auf, indem er Informationen zur Codeabdeckung nicht nur passiv auswertet, sondern sie aktiv in den Fuzzing-Workflow integriert, um die Testgenerierung gezielt zu steuern.

[1] `https://github.com/esp32-open-mac/qemu`

Yun et al. [Yun+22] geben in *Fuzzing of Embedded Systems: A Survey* einen breiten Überblick über den gesamten Fuzzing-Prozess eingebetteter Systeme. Sie betonen die Bedeutung von QEMU-Anpassungen für spezifische Hardware und gehen dabei auf Firmware-Dumping, Interface-Modellierung und Emulationstechniken ein. Darauf aufbauend identifizieren Eisele et al. [Eis+22] in *Embedded Fuzzing: A Review of Challenges, Tools, and Solutions* zentrale Herausforderungen beim Fuzzing von IoT-Geräten. Dazu zählen begrenzte Ressourcen, die Emulation spezialisierter Hardware und die Firmware-Instrumentierung. Die Autoren vergleichen verschiedene Emulationsformen sowie die Nutzung von Schnittstellen wie UART und JTAG. Beide Arbeiten liefern wertvolle Übersichten und Analysen, entwickeln jedoch keine neuen Fuzzing-Programme oder spezifische Implementierungen.

Das Buch *Fuzzing Against the Machine* von Nappa und Blázquez [NB23] beschreibt allgemeine Fuzzing- und Emulationskonzepte, insbesondere mit QEMU, und teilt damit die Grundidee dieser Dissertation. Es bleibt jedoch auf einer generischen Ebene und behandelt weder die ESP32-Plattform noch die Xtensa-Architektur, während der hier vorgestellte Ansatz gezielt auf diese spezialisiert ist.

Diese aufgeführten Übersichtsarbeiten zeigen deutlich, dass grundlegende Konzepte wie Emulation und Firmware-Verarbeitung zwar gut erforscht sind, es im Bereich der spezifischen Plattformanpassungen (z. B. Xtensa/ESP32) sowie in der Kombination statischer Instrumentierung mit Fuzzing in Emulationen jedoch noch erhebliche Forschungslücken gibt.

8.4 Fuzzing von Netzwerkprotokollen

Dem Erlernen und Fuzzing von Netzwerkprotokollen widmet sich ein eigener Forschungszweig, in dem zunehmend maschinelles Lernen und modellbasierte Verfahren zum Einsatz kommen.

Ansätze zur Klassifikation unbekannter Protokolle liefern wichtige Grundlagen für die Analyse von Netzwerkverkehr. Jung und Jeong [JJ20] trainieren ein Deep-Belief-Netzwerk auf statistisch gewonnenen Merkmalen, um Nachrichten mit hoher Genauigkeit zu klassifizieren. Lopez-Martin et al. [Lop+17] kombinieren Convolutional und Recurrent Neural Networks zur Kategorisierung von IoT-Verkehr, Michael et al. [Mic+17] nutzen neuronale Netze zur Erkennung unterschiedlicher Protokolle, und Li et al. [Li+18] setzen mit dem *Byte Segment Neural Network* (BSNN) Byte-Segmente ein, um Protokollstrukturen präzise zu identifizieren. Diese Arbeiten liefern wertvolle methodische Grundlagen für die Dissertation, fokussieren sich jedoch nicht auf die Generierung neuer Eingaben für das Fuzzing.

Für das Fuzzing selbst haben Pham et al. [PBR20] mit *AFLNET* den ersten Greybox-Fuzzer speziell für zustandsbehaftete Netzwerkprotokolle vorgestellt. Basierend auf aufgezeichnetem Client-Server-Verkehr erzeugt AFLNET Nachrichtensequenzen, die gezielt mutiert werden, um die Codeabdeckung und Zustandsabdeckung zu erhöhen. Durch das Lernen eines State-Machine-Modells erreicht AFLNET eine deutlich höhere Abdeckung als

klassische mutationsbasierte Fuzzer und demonstriert, dass modellbasierte Verfahren das Fuzzing komplexer Protokolle erheblich verbessern können.

PULSAR von Gascón et al. [Gas+15] ist ein modellbasiertes Fuzzing-Tool für zustandsbehaftete Netzwerkprotokolle. Es verwendet Markov-Modelle und Clustering, um Zustandsmaschinen abzuleiten, verzichtet jedoch auf den Einsatz neuronaler Netze. Während sich PULSAR auf die Testgenerierung auf Basis von Zustandsmodellen konzentriert, liegt der Fokus dieser Dissertation auf der automatisierten Protokollanalyse und der Generierung neuer Netzwerkpakete.

Die Forschungsergebnisse von PREUNN wurden mehrfach aufgegriffen und weiterentwickelt, um die automatisierte Protokollanalyse zu verbessern. *PREIUD* von Ning et al. [Nin+23] nutzt unüberwachtes Clustering und tiefe neuronale Netze, um Nachrichtentypen, Feldgrenzen und Abhängigkeiten zu erkennen, während *CNNPRE* von Garshasbi und Teimouri [GT23] Convolutional Neural Networks einsetzt, um aus Rohdaten von Netzwerkpaketen Muster und Feldgrenzen zu extrahieren. Beide Ansätze konzentrieren sich auf die Analyse und Segmentierung von Nachrichten, nicht jedoch auf die direkte Testfallerzeugung.

ProsegDL von Zhao et al. [Zha+22] erweitert dies, indem es Feldgrenzen automatisch aus Rohdaten lernt und unbekannte Protokollstrukturen rekonstruiert. *DL-ProS²* [Zha+24c] integriert U-Net, ein siamesisches Netzwerk und BiLSTM-CRF, um Feldgrenzen und semantische Informationen zu extrahieren. Ergänzt wird dies durch eine wissensbasierte Verkehrssimulation, die Protokollwissen aus öffentlichen Dokumenten, wie RFCs, einbezieht. Empirische Ergebnisse zeigen hohe Präzision und Recall bei der Analyse unbekannter Protokolle. *AEMK/EAEAP* von Nemati et al. [NMT24] nutzt Autoencoder-basiertes Clustering, um Nachrichtenmuster und semantische Beziehungen zwischen Nachrichten zu identifizieren. Alle diese Deep Learning-Ansätze fokussieren sich auf Analyse und Clustering, während die Testfallerzeugung nicht im Vordergrund steht.

Insgesamt zeigen diese Arbeiten, dass lern- und modellbasierte Verfahren die Analyse und das Fuzzing komplexer Netzwerkprotokolle erheblich verbessern können. Sie liefern vor allem methodische Konzepte und Übersichten, während die Entwicklung plattformspezifischer Fuzzing-Frameworks, wie sie in dieser Dissertation erfolgt, außerhalb ihres Fokus liegt.

8.5 Grammatik-basiertes Fuzzing

Grammatikbasierte Verfahren verfolgen das Ziel, strukturell valide Eingaben zu generieren, um eine höhere Code- und Zustandsabdeckung zu erzielen und somit verborgene Fehlerzustände aufzudecken.

Havrikov et al. [Hav+14] stellen mit *XMLMate* einen evolutionären Testgenerator vor, der auf XML-Schemas und vorhandenen Beispielen basiert. Durch die Anwendung genetischer Operatoren wie Mutation, Rekombination und Selektion werden strukturkonforme XML-Dokumente erzeugt, die sich gezielt für Robustheitstests von Parsern eignen.

Höschele et al. [HKZ17] schlagen mit *AUTOGRAM* einen Ansatz zur Grammatikinduktion vor: Beginnend mit wenigen Beispielen werden Datenflüsse analysiert und mittels Membership-Queries kontextfreie Grammatiken konstruiert, die sowohl formal korrekt als auch menschenlesbar sind und sich unmittelbar für Fuzzing einsetzen lassen.

Gopinath et al. [Gop+18] gehen mit *PYGMALION* noch einen Schritt weiter und zeigen, dass Grammatiken auch ohne Beispielinputs induziert werden können. Über systematisches Parserfuzzing werden dabei sukzessive gültige Eingaben (z. B. JSON, URLs) aufgebaut, die als Grundlage für weitere Tests dienen.

Pavese et al. [Pav+18] erweitern grammatikbasiertes Sampling um probabilistische Inversion: Neben häufigen Strukturen werden gezielt seltene, aber syntaktisch gültige Eingaben generiert, um ungewöhnliche Programmzweige zu erreichen und Robustheitsprobleme sichtbar zu machen.

Für strukturierte Binärformate zeigt Fioraldi et al. [FDC20] mit *WEIZZ*, dass Schlussfolgerungen über die Grammatik in Kombination mit Fuzzing valide und diversifizierte Eingaben erzeugen, was die Effektivität bei komplexen Formaten deutlich steigert.

Einen praxisnahen Ansatz verfolgen Dutra et al. [DGZ23] mit *FormatFuzzer*. Hierbei werden binäre Templates (z. B. aus dem *010 Editor*) in Parser und Generatoren übersetzt, wodurch komplex strukturierte Eingaben wie MP4- oder ZIP-Dateien erzeugt werden können. In Kombination mit AFL gelang so die Entdeckung bislang unbekannter Schwachstellen in FFmpeg und Timidity.

Einen anderen Ansatz verfolgt *GDBMiner* von Eisele et al. [Eis+25], das Eingabegrammatiken direkt aus Binärprogrammen extrahiert. Mithilfe des GNU-Debuggers (GDB) wird bytegenau analysiert, welche Eingabeteile von welchen Programmabschnitten verarbeitet werden, woraus präzise Grammatikregeln abgeleitet werden.

Amaya Zamudio et al. [ASZ25] untersuchen schließlich sprachbasierte Fuzzer, die formale Spezifikationen für die Testfallgenerierung nutzen. Der derzeit führende Ansatz, *ISLa*, setzt auf symbolisches Constraint-Solving und erzeugt damit hochpräzise, jedoch langsame Eingaben. Mit *FANDANGO* schlagen die Autoren eine suchbasierte Alternative auf Basis genetischer Algorithmen vor, die Constraints effizient erfüllt und dabei eine deutlich höhere Geschwindigkeit erreicht. Zudem erlaubt *FANDANGO* die Formulierung von Constraints direkt in Python, was eine größere Ausdrucksstärke und Flexibilität bietet.

Zusammenfassend legen die dargestellten Arbeiten die Grundlagen für effektives Fuzzing dar. Durch umfassende Kenntnisse der Eingabegrammatiken lässt sich die gezielte Erzeugung gültiger Testfälle systematisch realisieren. Unabhängig davon, ob dies mittels evolutionärer Verfahren, Parser-Fuzzing, Template-basierter Ansätze oder debuggergestützter Extraktion geschieht, führt eine grammatikorientierte Testfallgenerierung zu einer höheren Code- und Zustandsabdeckung. Insbesondere in IoT-Szenarien, in denen strukturierte Daten dominieren, erhöht grammatikbasiertes Fuzzing die Testeffizienz und steigert die Wahrscheinlichkeit, sicherheitsrelevante Schwachstellen aufzudecken.

8.6 Fuzzing mittels Machine Learning

Der Einsatz von ML im Fuzzing hat in den letzten Jahren erheblich an Bedeutung gewonnen. Das Ziel besteht darin, die Eingabegenerierung zu optimieren, um eine umfassendere Code- und Zustandsabdeckung sowie eine effizientere Schwachstellenentdeckung zu erreichen.

Einen der ersten konkreten Vorschläge machten Godefroid, Peleg und Singh [GPS17] mit *Learn&Fuzz*. Dabei werden Eingabeformate aus Beispieldaten gelernt und anschließend zur automatisierten Testfallgenerierung genutzt. Dieser Ansatz verdeutlicht, wie ML-Techniken eingesetzt werden können, um strukturierte Eingaben gezielt und systematisch zu erzeugen.

Spätere Arbeiten schlagen eine Brücke zwischen klassischen ML-Ansätzen und modernen LLM-Verfahren. So untersuchen Yang et al. [Yan+23], wie LLMs für das Whitebox-Compiler-Fuzzing eingesetzt werden können, und markieren damit eine Übergangsarbeit. Ein weiterer Schritt in diese Richtung ist *KernelGPT* von Yang et al. [YZZ23], das Kernel-Fuzzing durch die automatische Ableitung von Syzkaller-Spezifikationen erweitert. Syzkaller ist ein state-of-the-art feedbackgesteuerter Fuzzer für Betriebssystem-Kernel. Er deckt System-Call-APIs mittels formaler Spezifikationen ab und ermöglicht dadurch eine systematische Generierung von Testfällen.

Die Forschung zu LLM-basiertem Fuzzing entwickelt sich besonders dynamisch. Deng et al. [Den+23] schlagen mit *TitanFuzz* ein generatives Verfahren vor, das Saatprogramme für Deep-Learning-APIs erzeugt und Mutationen mithilfe eines Multi-Armed-Bandit-Algorithmus steuert. In einer Folgestudie erweitern Deng et al. [Den+24] diesen Ansatz zu *FuzzGPT*, das variierte Eingabeprogramme automatisch generiert und zahlreiche Bugs in PyTorch und TensorFlow aufdeckt. Liu et al. [LMC23] demonstrieren einen LLM-gestützten Ansatz zur automatisierten Ableitung von Fuzzing-Zielen, während Le Mieux et al. [Lem+23] mit *CodaMosa* zeigen, dass LLMs im Rahmen des Search-Based Software Testing (SBST) genutzt werden können, um aus bestehenden Testfällen neue abzuleiten und so die Testabdeckung zu erhöhen. Auch im Bereich des Protokollfuzzings finden LLMs Anwendung. Meng et al. [Men+24] stellen mit *ChatAFL* einen Fuzzer vor, der aus Beispieldaten Protokollregeln extrahiert, Eingabesequenzen vervollständigt und gezielt mutiert. Dies führt zu einer deutlich höheren Code- und Zustandsabdeckung und zur Entdeckung mehrerer bislang unbekannter Schwachstellen. Im Gegensatz zum Ansatz in der Dissertation, der sich auf XML-Parser konzentriert, liegt hier der Fokus auf Netzwerkprotokollen.

Parallel zu diesen Arbeiten ist eine Reihe von Übersichtsarbeiten entstanden, die den Forschungsstand systematisieren. Salem und Song [SS19] analysieren grammatikbasierte Techniken und betonen deren Bedeutung für die Erzeugung strukturierter Eingaben. Wang et al. [Wan+20a] bieten eine umfassende Klassifikation ML-basierter Fuzzing-Ansätze und zeigen deren Einsatz in verschiedenen Szenarien. Wu [Wu22] untersucht systematisch grammatikbasierte Verfahren und bewertet ihr Potenzial in Kombination mit ML. Neuere Meta-Arbeiten betrachten explizit den Einsatz von LLMs: Yu et al. [Jia+24] identifizieren fünf zentrale Herausforderungen beim Fuzzing mit LLMs, darunter die Abhängigkeit von Prompt-Sensitivität, die Sicherstellung der Gültigkeit und Vielfalt generierter Eingaben, die fehlende Standardisierung von Evaluationsmethoden, die Kostenfrage bei großem

Rechenaufwand sowie die Notwendigkeit robuster Benchmarks. Huang et al. [Hua+25] geben schließlich einen breiten Überblick über den aktuellen Stand: Von 14 untersuchten Arbeiten nutzen fünf LLMs für Prompt Engineering, zehn für Seed-Mutation und zwei für die Sammlung und Analyse von Ausgaben.

Insgesamt verdeutlichen diese Arbeiten, dass der Einsatz von Machine Learning (insbesondere LLMs) im Fuzzing ein stark wachsendes Forschungsfeld darstellt. Die Kombination von LLMs mit grammatikbasierten Methoden bietet dabei ein erhebliches Potenzial, um die Testeffizienz zu steigern und komplexe Schwachstellen systematisch aufzudecken.

8.7 Optimierung des Fuzzing-Prozesses

Ein effizienter Fuzzing-Prozess zeichnet sich nicht nur durch eine hohe Testabdeckung, sondern auch durch schnelle und ressourcenschonende Abläufe aus. Zahlreiche Arbeiten zeigen, wie gezielte Optimierungen die Performance deutlich steigern können.

Fioraldi et al. [Fio+20] demonstrieren dies mit *AFL++*, das durch inkrementelle Verbesserungen bei Mutation, Scheduler-Design und Codeinstrumentierung die Performance gegenüber dem Original-AFL erheblich erhöht. Dabei werden Erkenntnisse aus der aktuellen Fuzzing-Forschung direkt in die Implementierung übernommen.

Ein anderer Ansatz, der die Effektivität von Fuzzer-Ensembles zeigt, ist *EnFuzz* von Chen et al. [Che+19]. Durch ensemblebasierte Seed-Synchronisation werden mehrere komplementäre Fuzzer synergistisch eingesetzt, was sowohl die Codeabdeckung als auch die Entdeckung von Abstürzen verbessert.

Wang et al. [Wan+19] stellen mit *NeuFuzz* einen pfadbasierten Graybox-Fuzzer vor, der ein tiefes neuronales Netz zur gezielten Priorisierung von Seeds nutzt. Das Modell lernt aus vielen „vulnerablen“ und „sicheren“ Pfaden verborgene Merkmale und vergibt eine höhere „Mutation Energy“ an Seeds, die vermutlich fehleranfällige Pfade abdecken.

She et al. [She+19] verfolgen mittels *NEUZZ* einen datengetriebenen Ansatz: Durch *Program Smoothing* in Kombination mit neuronalen Surrogatmodellen werden gradientenbasierte Eingaben generiert, die eine schnellere Entdeckung von Bugs ermöglichen und gleichzeitig die Kantenabdeckung signifikant erhöhen. Im Gegensatz dazu adressieren Nagy und Hicks [NH19] mit *Full-Speed Fuzzing* primär den Laufzeit-Overhead. Es werden ausschließlich jene Testfälle instrumentiert, die tatsächlich eine Veränderung der Codeabdeckung bewirken. Dadurch wird die benötigte Rechenzeit deutlich reduziert.

Böhme et al. [BMC20] verfolgen mit *Entropic* einen informations-theoretischen Ansatz: Jeder generierte Testinput wird nach seinem erwarteten Informationsgehalt bewertet. Dadurch werden Seeds, die neue oder seltene Programmpfade abdecken, häufiger ausgewählt und mutiert. Auf diese Weise werden Bugs effizienter entdeckt und die Codeabdeckung beschleunigt. In LibFuzzer führte dies zu signifikanten Verbesserungen bei der Entdeckung neuer Programmbahnen und Fehler.

Jauernig et al. [Jau+23] setzen evolutionäre Strategien ein. Das Programm *DARWIN* passt Mutationswahrscheinlichkeiten adaptiv an und verbessert dadurch die Bug-Findung und die Testabdeckung.

Diese Arbeiten verdeutlichen, wie Prozessoptimierung im Fuzzing durch intelligente Mutationsstrategien, effizientes Tracing oder den Einsatz von Fuzzer-Ensembles erreicht werden kann. Der Fokus dieser Dissertation liegt hingegen auf der Entwicklung einer modularen Architektur, die bestehende Fuzzer als austauschbare Module integriert und so Fortschritte aus der aktuellen Forschung direkt nutzbar macht.

9 Zusammenfassung und Ausblick

In diesem Kapitel werden die zentralen Ergebnisse dieser Dissertation zusammengefasst und ein Ausblick auf zukünftige Entwicklungen im Bereich des Fuzzings ressourcenbeschränkter IoT-Geräte gegeben. Der Fokus liegt dabei sowohl auf den einzelnen Beiträgen der Dissertation als auch auf deren Zusammenspiel innerhalb eines übergreifenden Frameworks.

9.1 Zusammenfassung

Die Dissertation adressiert die zentrale Herausforderung, Fuzzing für ressourcenbeschränkte IoT-Geräte am Beispiel des ESP32 effizienter, flexibler und skalierbarer zu gestalten. Während bestehende Arbeiten meist nur isolierte Aspekte betrachten, wird hier ein integrativer Ansatz verfolgt. Es wurden vier eigenständige Verfahren (Module) entwickelt, implementiert und evaluiert, die gemeinsam die Grundlage für ein konzeptionelles Framework bilden. Die Komponenten sind modular ausgelegt und können je nach Anwendungsfall miteinander kombiniert werden. Auf diese Weise ergibt sich ein flexibles Architekturprinzip.

Das erste Modul, ESP32 Binary Rewriting (EBR), überträgt Konzepte, die bislang nur für Architekturen wie ARM und x86 verfügbar waren (beispielsweise *e9patch*), erstmals auf die Xtensa-Architektur des ESP32. Dieses Verfahren ermöglicht das Einfügen von beliebigem Code in bestehende Firmware-Binaries, insbesondere von Instrumentierungen, ohne deren ursprüngliche Funktion zu beeinträchtigen. Während der Programmausführung können somit Informationen zu Speicherzugriffen, Funktionsaufrufen und Kontrollflüssen erfasst und direkt an den Fuzzer übermittelt werden. Die Evaluierung belegt, dass die Instrumentierungen korrekt arbeiten und die Integrität der Firmware erhalten bleibt (siehe Kapitel 3).

Ein weiteres Modul ist ESP32-QEMU-FUZZ (EQF), das die vollständige Emulation von ESP32-Firmwares adressiert. Das Ziel bestand darin, verschiedene unvollständige Ansätze wie *ESP32-QEMU* und *QEMU-Fuzz* zu konsolidieren und eine funktionsfähige Umgebung bereitzustellen, die trotz fehlender Hardwarekomponenten wie WLAN eine automatisierte Testung ermöglicht. Das resultierende System erlaubt erstmals die vollständige Emulation beliebiger ESP32-Firmwares und zeigt eine erhebliche Effizienzsteigerung: Während hardwarebasiertes Fuzzing nur vier bis 40 Eingaben pro Sekunde ermöglicht, erreicht EQF bis zu 320 Eingaben pro Sekunde, was einer Beschleunigung um den Faktor 80 entspricht (siehe Kapitel 4).

Darüber hinaus wurde mit Protocol Reverse Engineering using Neural Networks (PREUNN) ein Modul entwickelt, das die manuell durchgeführte Aufgabe des Protocol Reverse Engineering proprietärer Netzwerkprotokolle automatisiert. Die Herausforderung hierbei lag in der Auswahl und Kombination geeigneter neuronaler Netze, um Protokollstrukturen ohne vorhandene Spezifikation zuverlässig zu rekonstruieren. Die Methode erlaubt erstmals die implizite Rekonstruktion von Protokollinformationen und die Generierung syntaktisch wie semantisch korrekter Netzwerkpakete. In der Evaluation erzielte PREUNN eine Erfolgsquote von 67,6 % für HTTP und 100 % für FTP, was die Testabdeckung im Vergleich zu zufälligen Ansätzen signifikant erhöht (siehe Kapitel 5).

Ein ergänzendes Modul ist How to Train Your Llama (HTTYL), welches den Einsatz von LLMs für grammatikbasiertes Fuzzing untersucht. Die zentrale Herausforderung bestand darin, LLMs so in bestehende Testabläufe einzubinden, dass sie gezielt syntaktisch und semantisch korrekte Eingaben erzeugen. Im Gegensatz zu prototypischen Lösungen wie *ChatFuzz* integriert HTTYL LLMs modular in den Fuzzing-Prozess und ermöglicht dadurch eine kontrollierte Zusammenarbeit zwischen Fuzzer und Modell. Die Evaluierung zeigt deutliche Verbesserungen: Eine um 50 % höhere Programmflussabdeckung gegenüber klassischen grammatikbasierten Fuzzern sowie eine bis zu sechsmal höhere Abdeckung im Vergleich zu Fuzzing ohne LLM (siehe Kapitel 6).

Schließlich werden die vier Verfahren in einem Ansatz für ein integriertes Fuzzing-Framework (AiFF) konzeptionell zusammengeführt. Diese modulare Plattform erlaubt flexible Kombinationen: So können EBR, EQF und PREUNN gemeinsam für protokollzentriertes Fuzzing eingesetzt werden, während sich EBR, EQF und HTTYL für strukturierte Dateiformate eignen. Durch den modularen Aufbau lassen sich die Komponenten an unterschiedliche Szenarien anpassen und unabhängig voneinander weiterentwickeln (siehe Kapitel 7).

Insgesamt liefert diese Dissertation ein Rahmenkonzept, das wesentliche Fortschritte im Fuzzing ressourcenbeschränkter IoT-Geräte erzielt. Die wichtigsten Ergebnisse sind:

- EBR: universelle Instrumentierungsfähigkeit für Xtensa-Firmware bei vollständiger Wahrung der Firmware-Integrität.
- EQF: deutliche Effizienzgewinne mit bis zu 320 Eingaben pro Sekunde (Faktor 80 gegenüber hardwarebasiertem Fuzzing) durch vollständige Emulation und integriertes Fuzzing.
- PREUNN: automatisierte Protokollrekonstruktion mit 67,6 % gültigen HTTP-Paketen und 100 % gültigen FTP-Paketen; signifikant höhere Testabdeckung als zufällige Ansätze.
- HTTYL: 50 % höhere Programmflussabdeckung als klassische grammatikbasierte Fuzzer und bis zu Faktor 6 höhere Abdeckung gegenüber Fuzzing ohne LLM.

Im Gegensatz zu bisherigen Arbeiten, die nur isolierte Teilaspekte wie Binary Rewriting, Emulation, Protokollanalyse oder Dateigenerierung behandeln, verfolgt diese Dissertation erstmals einen integrierten, modularen Ansatz dieser Aspekte.

9.2 Ausblick und zukünftige Arbeiten

Die Ergebnisse dieser Dissertation verdeutlichen das Potenzial einer kombinierten Nutzung von Binary Rewriting, Emulation, PRE und LLMs für das Fuzzing von IoT-Geräten. Aufbauend auf dem entwickelten Prototyp erscheint es naheliegend, die vorgestellten Module in einer einheitlichen Plattform zu integrieren. Eine solche modulare und reproduzierbare Architektur würde die systematische Analyse der Wechselwirkungen zwischen den einzelnen Verfahren ermöglichen und zugleich eine Grundlage für standardisierte Benchmarks und Metriken schaffen. Damit ließe sich die Effektivität verschiedener Modulkombinationen empirisch evaluieren und deren Einsatz in unterschiedlichen Anwendungsszenarien fundiert bewerten.

Im Zentrum der weiteren Forschung steht daher die praktische Umsetzung des AiFF-Konzepts als lauffähige Plattform. Diese sollte eine einheitliche Schnittstellenarchitektur, umfassendes Logging sowie Mechanismen zur Reproduzierbarkeit von Experimenten bieten. Ergänzend ist die Definition geeigneter Evaluationsmetriken notwendig, die sowohl strukturelle Abdeckung als auch Performance, Robustheit und Fehlerraten erfassen. Darauf aufbauende Fallstudien mit realistischen Testumgebungen könnten die Praxistauglichkeit der entwickelten Ansätze belegen.

Ein zweiter Schwerpunkt betrifft die Erweiterung des Plattform- und Architektursupports. Hierzu zählt insbesondere die Portierung von EBR und EQF auf zusätzliche IoT-relevante Prozessorarchitekturen sowie die vollständige Abdeckung des Xtensa-Befehlssatzes. Auf diese Weise ließe sich eine lückenlose Instrumentierung von Firmware realisieren und gleichzeitig die Übertragbarkeit der Ansätze auf heterogene Systeme gewährleisten. Darüber hinaus erscheint es sinnvoll, den Einfluss unterschiedlicher Netzwerktopologien und Laufzeitumgebungen auf die Wirksamkeit der Protokollrekonstruktion und des Fuzzings systematisch zu untersuchen.

Auch auf der Ebene der Eingabeverarbeitung ergeben sich vielfältige Perspektiven. Die Weiterentwicklung von PREUNN und dessen Anpassung an komplexe Protokolle soll es ermöglichen, mithilfe robuster Methoden zur Feature-Extraktion die Qualität automatisch rekonstruierter Protokollstrukturen weiter zu verbessern und deren Einsatz in dynamischen Netzwerkumgebungen effizient zu evaluieren. Parallel dazu sollte HTTYL um die Fähigkeit erweitert werden, strukturierte Datenformate wie PDF, SQL-Dumps oder proprietäre Binärdateien zu generieren und zu mutieren. Um die syntaktische und semantische Korrektheit der erzeugten Testfälle sicherzustellen, sind ergänzende Mechanismen zur Validierung erforderlich.

Ein weiteres zukunftsträchtiges Forschungsfeld bildet der Einsatz von LLMs für die Generierung und Modellierung strukturierter Eingaben. Aufbauend auf den in dieser Dissertation erzielten Ergebnissen könnten LLMs gezielt für spezifische Datenformate optimiert werden. Moderne Trainingsansätze wie LoRA versprechen dabei eine Steigerung der Testqualität bei gleichzeitig reduzierten Trainingskosten. Offene Fragen betreffen insbesondere Strategien zur Maximierung der Vielfalt fehlerverursachender Testfälle, Methoden zur Vermeidung von Überanpassung sowie Verfahren zur zuverlässigen Bewertung der semantischen Korrektheit generierter Eingaben.

Langfristig besteht die Perspektive, die entwickelten Konzepte in eine offene und modulare Fuzzing-Tool-Sammlung zu überführen. Diese würde sowohl die Wiederverwendbarkeit einzelner Komponenten als auch die Vergleichbarkeit zukünftiger Forschungsergebnisse unterstützen. Eine solche Plattform könnte als Referenzimplementierung dienen und zugleich die Basis für großangelegte empirische Studien bilden.

Begriffsdefinitionen

In dieser in deutscher Sprache verfassten Dissertation werden zahlreiche Fachbegriffe verwendet, die sich ursprünglich im Englischen etabliert haben. Bei der Übersetzung wurde darauf geachtet, Begriffe beizubehalten, für die es im Deutschen keine adäquaten Entsprechungen gibt oder deren deutsche Übersetzung im wissenschaftlichen Sprachgebrauch nicht üblich ist. Einige Begriffe wurden nicht übersetzt, weil sie auch im Deutschen weit verbreitet sind oder weil eine Übersetzung zu Bedeutungsunklarheiten führen könnte – insbesondere bei mehrdeutigen englischen Begriffen wie „security“ und „safety“. Da außerdem einige Fachbegriffe bereits in ihrer Ursprungssprache unterschiedliche Interpretationen zulassen, werden im Folgenden die für diese Dissertation gültigen Definitionen erläutert.

1D-Batch
: Eine Gruppe von eindimensionalen Eingabesequenzen, die gemeinsam durch ein neuronales Netzwerk verarbeitet werden.

1D-Batch-Normalisierung
: Die Anwendung der Batch-Normalisierung (siehe unten) speziell auf eindimensionale Daten, z. B. Sequenzen oder Zeitreihen, um die Trainingsstabilität und Konvergenz zu verbessern.

1D-Faltung
: Die Operation, bei der ein Filter (Kernel) über eindimensionale Daten verschoben wird, um lokale Merkmale zu extrahieren.

1D-Faltungsschichten
: Neurale Netzwerk-Layer, die eindimensionale Convolutionen auf sequenzielle Daten anwenden, um lokale Muster und Features zu extrahieren.

1D-Transponierte-Faltungsschichten
: Layer, die die Inverse der 1D-Faltung durchführen, um Daten zu vergrößern oder rekonstruktive Aufgaben wie bei Autoencodern zu unterstützen.

4-Gramm
: Eine Sequenz von vier aufeinanderfolgenden Elementen, wie Zeichen, Wörter oder Tokens, die in Text- oder Sequenzanalysen verwendet wird, um Muster oder Abhängigkeiten zu erfassen.

Adam-Optimizer
: Ein iterativer Optimierungsalgorithmus im maschinellen Lernen, der die Lernrate für jeden Parameter automatisch anpasst und vergangene Gradienteninformationen berücksichtigt.

AddressSanitizer (ASan)
Ein Laufzeitanalyse-Tool, das durch Compiler-Instrumentierung zusätzliche Speichergrenzenprüfungen und sogenannte *Stack Canaries* einfügt, um Speicherfehler wie Pufferüberläufe, Use-after-Free, Nullzeigerzugriffe und Speicherlecks (siehe unten) zuverlässig zu erkennen.

Augmentation
Eine Technik im maschinellen Lernen, bei der Trainingsdaten künstlich erweitert oder verändert werden, z. B. durch Rotation, Skalierung oder Hinzufügen von Rauschen, um die Modellrobustheit und Generalisierung zu verbessern.

Batch-Größe
Die Anzahl der Trainingsbeispiele, die in einem Durchgang (Iteration) eines Optimierungsalgorithmus verarbeitet werden.

Batch-Normalisierung
Ein Verfahren, das die Aktivierungen innerhalb eines Layers normalisiert, um die Trainingsstabilität zu erhöhen und die Konvergenz zu beschleunigen.

Binary Cross-Entropy (BCE)
Spezialform der Cross-Entropy (CE) Verlustfunktion (siehe unten) für binäre Klassifikationsaufgaben.

Binary Recovery
Verfahren zur teilweisen oder vollständigen Wiederherstellung von Programmcode aus Binärdateien, häufig im Kontext von Reverse Engineering oder Malware-Analyse.

Binary Rewriting
Ein Verfahren, bei dem bestehende Binärdateien analysiert und gezielt modifiziert werden, etwa durch Einfügen von Patches, Hooks oder Sprüngen, ohne das ursprüngliche Programmverhalten zu zerstören.

Categorical Cross-Entropy (CCE)
Spezialform der Cross-Entropy (CE) Verlustfunktion (siehe unten) für Mehrklassen-Klassifikationsaufgaben

Clustering
Ein Verfahren des maschinellen Lernens, bei dem Datenpunkte anhand ihrer Ähnlichkeit automatisch in Gruppen (Cluster) zusammengefasst werden, ohne dass vorher definierte Klassen bekannt sind.

Code Location Problem
Eigenname für das Problem, an welchen Stellen im Programmcode eigene Änderungen eingefügt werden können, ohne die Funktionalität zu beeinträchtigen.

Code-Injektion
Bezeichnet das Einschleusen von Code in ein Programm, meist mit dem Ziel, unerwünschte Operationen auszuführen.

Codeabdeckung
Ein Maß dafür, welcher Anteil des Programmcodes während eines Tests tatsächlich ausgeführt wird.

Compiler
Ein Programm, das Quellcode in ausführbaren Maschinencode übersetzt und dabei verschiedene Optimierungen vornimmt.

Cross-Entropy (CE) Verlustfunktion
Eine allgemeine Verlustfunktion (siehe unten) für Klassifikationsaufgaben, die die Abweichung zwischen vorhergesagten Wahrscheinlichkeitsverteilungen und den tatsächlichen Zielverteilungen anhand des negativen Logarithmus der korrekten Klasse misst.

Datensatz
Eine Sammlung von Samples (siehe unten), die für Training, Validierung oder Testen von maschinellen Lernmodellen verwendet wird.

Debugging-Schnittstelle
Hard- oder Software-Schnittstelle, die den Zugriff auf interne Zustände eines Systems ermöglicht, z. B. über JTAG (siehe unten), um die Fehleranalyse und Diagnose zu erleichtern.

Density-Based Spatial Clustering of Applications with Noise (DBSCAN)
Ein dichtebasiertes Clustering-Verfahren, das Cluster als zusammenhängende Regionen hoher Punktdichte identifiziert und Punkte in dünn besiedelten Bereichen als Ausreißer markiert.

Embedding
Eine Vektordarstellung von Daten, z. B. von Wörtern, Tokens oder Codefragmenten, in einem kontinuierlichen Merkmalsraum, um semantische Ähnlichkeiten messbar zu machen.

Emulation
Verfahren, bei dem Hardware oder Software eines Systems vollständig nachgebildet wird, um Programme in einer kontrollierten Umgebung auszuführen.

Epoche
Ein kompletter Durchlauf durch den gesamten Trainingsdatensatz während des Trainings eines maschinellen Lernmodells.

ESP32
Ein kostengünstiger, stromsparender Mikrocontroller mit integriertem WLAN- und Bluetooth-Support, der häufig in IoT-Geräten eingesetzt wird.

Feature Extraction
Der Prozess, bei dem ein Modell relevante Merkmale aus den Rohdaten identifiziert und extrahiert.

Feature Map
Die Ausgabedarstellung eines Layers in einem neuronalen Netzwerk, die die aus den

Eingabedaten extrahierten Merkmale für jeden Kanal abbildet und als Input für nachfolgende Layer dient.

Feature Reverse Engineering
Analyse bestehender Software oder Hardware, um ihre Funktionalitäten und Eigenschaften zu identifizieren und zu verstehen.

Feature
Ein einzelnes erkennbares Muster oder eine charakteristische Eigenschaft in den Eingabedaten, das von einem neuronalen Netzwerk extrahiert wird.

Feedbackgesteuertes Fuzzing
Eine Fuzzing-Technik, bei der die Generierung von Testeingaben durch Informationen über die bisher abgedeckten Programmstellen gesteuert wird.

Fine-Tuned LLM
Ein Large Language Model (LLM) (siehe unten), das durch gezieltes Nach-Training an spezifische Aufgaben oder Domänen angepasst wurde.

Fine-Tuning
Methode des maschinellen Lernens, bei der ein vortrainiertes Modell durch Training auf spezifischen Daten weiter spezialisiert wird.

Firmware
Software, die dauerhaft auf Hardwarekomponenten gespeichert ist und deren grundlegende Funktionalität steuert.

Flashen
Bezeichnet das (Über-)Schreiben bzw. Ersetzen der Firmware eines Geräts durch eine neue Version, meist mittels spezieller Tools oder Bootloader.

Fork
Eine Kopie eines Softwareprojekts oder Repositories, die unabhängig vom Original weiterentwickelt werden kann, häufig genutzt, um eigene Änderungen vorzunehmen oder Beiträge zurück zum Originalprojekt vorzuschlagen.

Framework
Eine strukturierte Sammlung von Bibliotheken, Konventionen und Tools, die die Entwicklung spezifischer Anwendungen oder Systeme erleichtert.

Fuzzing
Eine Testtechnik, bei der Programme automatisiert mit vielen ungültigen, zufälligen oder unerwarteten Eingaben getestet werden, um Schwachstellen aufzudecken. Der Begriff wird im Deutschen bereits verwendet.

Hamming-Distanz
Ein Maß zur Bestimmung der Unterschiedlichkeit zweier gleich langer Bit- oder Zeichenketten, definiert als die Anzahl der Positionen, an denen sich die Symbole unterscheiden.

Heap
Ein dynamischer Speicherbereich, aus dem Programme während der Laufzeit Speicherblöcke anfordern und wieder freigeben können.

Hierarchical Density-Based Spatial Clustering of Applications with Noise (HDBSCAN)
Eine Erweiterung von Density-Based Spatial Clustering of Applications with Noise (DBSCAN) (siehe oben), die hierarchisches Clustering mit dichtebasierter Clusterbildung kombiniert, um Cluster unterschiedlicher Dichte robuster zu erkennen und Ausreißer effizient zu identifizieren.

Hook
Eine Technik, bei der bestehender Code gezielt umgeleitet oder erweitert wird, indem vor, nach oder anstelle von Funktionsaufrufen zusätzlicher Code ausgeführt wird.

Hyperparameter
Einstellbare Parameter eines maschinellen Lernmodells, die nicht während des Trainings gelernt werden, sondern vorab festgelegt werden, wie Lernrate, Batch-Größe oder Anzahl der Layer.

Inferenz
Der Prozess, bei dem ein trainiertes Modell neue Eingabedaten verarbeitet, um Vorhersagen, Klassifikationen oder andere Ausgaben zu erzeugen.

Instrumentierung
Technik, bei der zusätzlicher Code in ein Programm eingefügt wird, um Laufzeitinformationen wie Codeabdeckung, Variablenwerte oder Kontrollfluss zu messen, ohne die ursprüngliche Programmlogik zu verändern.

Jump
Ein Sprungbefehl in einem Programm, der den Kontrollfluss an eine andere Stelle im Code überträgt.

JTAG
Ein Standard für die Debugging- und Testschnittstelle von Mikrocontrollern und integrierten Schaltkreisen, der Zugriff auf interne Register, Speicher und die Steuerung des Programmschritts ermöglicht.

Konvolutionelles Embedding
Eine dichte Vektorrepräsentation von Eingabedaten, die durch die Anwendung von Convolutional-Layern erzeugt wird und lokale Muster sowie Merkmale der Daten in einem niedrigdimensionalen Raum abbildet.

Large Language Model (LLM)
Maschinelle Lernmodelle, die auf der Verarbeitung und Generierung natürlicher Sprache in großem Maßstab basieren. Aufgrund fehlender etablierter deutscher Übersetzungen wird der englische Begriff verwendet.

Latente Repräsentation
Die komprimierte Darstellung der Eingabedaten, die ein Autoencoder nach der Encodierung gelernt hat, also die wichtigsten Merkmale oder Informationen, die das Modell zur Rekonstruktion der Daten benötigt.

Leaky ReLU
Eine Variante der Rectified Linear Unit (ReLU)-Aktivierungsfunktion (siehe unten), bei

der negative Eingaben nicht vollständig auf null gesetzt, sondern mit einem kleinen Faktor skaliert werden, um das Problem verschwindender Gradienten zu reduzieren.

LLaMA-2-13B-Modell
Ein LLM (siehe oben) der zweiten LLaMA-Generation von Meta mit 13 Milliarden Parametern.

Mean Squared Error (MSE)
Ein Maß für die mittlere quadratische Abweichung zwischen vorhergesagten und tatsächlichen Werten.

Multi-Hot-Vektor
Ein Vektor, bei dem mehrere Positionen den Wert 1 haben und alle anderen 0 sind. Häufig verwendet, um mehrere aktive Kategorien oder Merkmale gleichzeitig zu repräsentieren.

Negative Log-Likelihood (NLL)
Eine Verlustfunktion im maschinellen Lernen, die die Wahrscheinlichkeit der richtigen Zielwerte maximiert, indem sie den negativen Logarithmus der Modellwahrscheinlichkeiten berechnet.

Neighbor Eviction
Eine Patching-Taktik (siehe unten), bei der eine benachbarte Instruktion verschoben oder überschrieben wird, um Platz für zusätzlichen Patch-Code oder neue Sprungziele zu schaffen.

Network Trace
Aufzeichnung und Analyse von Netzwerkkommunikation, um Datenflüsse, Protokollnutzung oder Fehlverhalten von Anwendungen nachzuvollziehen.

Neuron
Eine grundlegende Verarbeitungseinheit in künstlichen neuronalen Netzen, die Eingaben gewichtet summiert, eine Aktivierungsfunktion anwendet und so eine Ausgabe erzeugt, die an andere Neuronen weitergegeben wird.

Nullzeiger
Ein Zeiger, der auf keine gültige Speicheradresse zeigt, üblicherweise mit dem Wert null initialisiert.

Nullzeigerzugriff (engl. Null Pointer Dereference)
Ein Laufzeitfehler, der entsteht, wenn ein Programm versucht, über einen Zeiger mit dem Wert Null auf Speicher zuzugreifen, was typischerweise zu einem Absturz führt.

One-Hot-Encoding
Eine Technik zur Darstellung kategorialer Daten als binäre Vektoren, bei denen genau eine Position den Wert 1 hat und alle anderen 0, um die Daten für maschinelle Lernmodelle nutzbar zu machen.

One-Hot-Vektor
Eine Vektordarstellung, bei der genau eine Komponente den Wert 1 hat und alle anderen 0, die häufig zur Kategorisierung diskreter Werte verwendet wird.

Opcode
Kurzform für „Operation Code“. Bezeichnet den Teil einer Maschineninstruktion, der angibt, welche Operation die CPU ausführen soll, z. B. Addition, Vergleich oder Sprung.

Overfitting
Ein Effekt beim maschinellen Lernen, bei dem ein Modell die Trainingsdaten zu genau lernt und dadurch Muster auswendig statt verallgemeinerbar erfasst, was zu schlechterer Leistung auf neuen, unbekannten Daten führt.

Parser
Ein Programm oder Modul, das strukturierte Daten (z. B. Quellcode, XML oder JSON) analysiert und in eine für die Weiterverarbeitung geeignete Form überführt.

Patching-Strategie
Übergeordneter Plan für das Einfügen, Ersetzen oder Entfernen von Programmcode in einem Binärprogramm, z. B. zur Laufzeitmanipulation oder Fehlerkorrektur.

Patching-Taktiken
Konkrete Methoden innerhalb einer Patching-Strategie (siehe oben), etwa Hooking (siehe oben), Überschreiben von Sprungzielen oder binäre Erweiterung von Funktionen.

Prompt-Tuned LLM
Ein LLM (siehe oben), das durch gezielte Optimierung seiner Antworten anhand vordefinierter Eingabeaufforderungen (Prompts, siehe oben) an spezifische Aufgaben angepasst wurde.

Prompt-Tuning
Trainingsmethode, bei der nur die Prompts (siehe oben) oder deren Einbettungen optimiert werden, ohne die zugrunde liegenden Modellparameter zu verändern.

Prompt
Text- oder Dateneingaben, die einem Modell wie einem LLM gegeben werden, um eine gewünschte Antwort, Vorhersage oder Handlung zu erzeugen.

Proof of Concept (PoC)
Ein einfacher Demonstrator oder Prototyp, mit dem die technische Machbarkeit einer Idee oder Angriffsmethode gezeigt wird.

Protokoll
Bezieht sich in dieser Dissertation auf Netzwerkprotokolle, also standardisierte Kommunikationsregeln zwischen Systemen [Pos81a].

Pufferüberlauf (engl. Buffer Overflow)
Ein Fehler, bei dem mehr Daten in einen Speicherpuffer geschrieben werden, als dieser aufnehmen kann, was zu Speicherüberschreibungen und potenziell zu sicherheitskritischen Schwachstellen führt.

Punned Jump
Eine Patching-Taktik (Siehe oben), bei der der Sprungbefehl Teile der nachfolgenden Instruktion überlagert, um zusätzliche Möglichkeiten für Sprungziele zu schaffen.

Reinforcement Learning
Ein Lernverfahren, bei dem ein Agent durch Interaktion mit einer Umgebung schrittweise lernt, optimale Aktionen auszuführen, indem er für erfolgreiches Verhalten Belohnungen und für Fehler Bestrafungen erhält.

ReLU
Eine Aktivierungsfunktion im maschinellen Lernen, definiert als $\mathrm{ReLU}(x) = \max(0, x)$, die negative Eingaben auf null setzt und positive unverändert lässt.

Retrieval-Augmented Generation (RAG)
Bezeichnet eine KI-Technik, die LLMs (siehe oben) mit externen Datenquellen kombiniert, um genauere, aktuellere und relevantere Antworten zu generieren.

Reverse-Engineering
Bezeichnet die Analyse und Rekonstruktion bestehender Systeme oder Software. Der Begriff ist im Deutschen gebräuchlich und wird nicht übersetzt.

Reverse-Order-Patching-Strategie
Eine Strategie, bei der Patches rückwärts im Code platziert werden (z. B. vom Ende zum Anfang), um Konflikte zu minimieren oder Platz effizient zu nutzen.

Rewriter
Programme oder Tools, die bestehende Binärdateien analysieren und gezielt modifizieren, etwa durch Einfügen von Sprüngen, Code oder Hooks.

Sample
Ein einzelnes Datenobjekt, das für Training, Test oder Analyse in einem maschinellen Lernmodell verwendet wird.

Seed Pool
Eine Sammlung von Seeds (siehe oben) für weiteres Fuzzing (siehe oben).

Seed
Eine initiale Eingabedatei oder Datenstruktur, die als Ausgangspunkt für die Generierung neuer Testeingaben beim Fuzzing (siehe oben) dient.

Sigmoid-Funktion
Eine Aktivierungsfunktion im maschinellen Lernen, definiert als $\sigma(x) = \frac{1}{1+e^{-x}}$, die Eingaben auf den Bereich (0, 1) abbildet.

Softplus
Eine glatte, differenzierbare Aktivierungsfunktion im maschinellen Lernen, definiert als $\mathrm{Softplus}(x) = \ln(1 + e^x)$, die eine stetige Approximation der ReLU-Funktion (siehe oben) darstellt.

Sparse Vector
Ein dünnbesetzter Vektor, der zu einem Großteil aus Nullen besteht und nur wenige relevante Werte enthält.

Speicherleck (engl. Memory Leak)
Ein Fehler, bei dem ein Programm reservierten Speicher nicht wieder freigibt, was im

Laufe der Zeit zu einem steigenden Speicherverbrauch und möglichen Leistungsproblemen oder Abstürzen führt.

Stack
Ein Speicherbereich, der für die Verwaltung von Funktionsaufrufen, lokalen Variablen und Rücksprungadressen genutzt wird. Daten werden nach dem LIFO-Prinzip (Last In, First Out) abgelegt.

Successor Eviction
Eine Patching-Taktik (siehe oben), bei der die aktuelle Instruktion und ihre direkt folgende Nachfolgerinstruktion an einen anderen Codebereich verschoben werden. Der Kontrollfluss wird angepasst, um Platz für zusätzlichen Patch-Code zu schaffen und die Programmlogik beizubehalten.

Synthetische Bugs
Künstlich eingefügte Fehler in Software, die gezielt zur Evaluierung und zum Benchmarking von Testverfahren wie Fuzzing (siehe oben) verwendet werden.

Tanh-Funktion
Eine Aktivierungsfunktion im maschinellen Lernen, definiert als $\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$, die Eingaben auf den Bereich (-1, 1) abbildet.

Time-to-Market (TTM)
Die Zeit, die ein Produkt vom Konzept bis zur Markteinführung benötigt. Dies führt zu einem Dilemma, da die Sicherheit vernachlässigt wird, um eine kürzere Time-to-Market (TTM) zu erreichen.

Token
Eine kleinste bedeutungstragende Einheit in Textdaten, die von Sprachmodellen verarbeitet wird, z. B. ein Wort, ein Satzzeichen oder ein Teilwort.

Tool
Eine Software oder ein Hilfsprogramm, das für eine spezifische Aufgabe eingesetzt wird, z. B. zur Analyse, Modifikation oder Überprüfung von Binärcode.

Top-k-Sampling
Eine Sampling-Strategie bei der Text- oder Sequenzgenerierung, bei der das Modell nur aus den k wahrscheinlichsten nächsten Token auswählt.

(Un)überwacht
Bezieht sich beim maschinelles Lernen auf das Training mit (un)beschrifteten Daten, um Muster mit bzw. ohne Vorgaben zu erlernen.

Use-after-Free
Ein Speicherfehler, bei dem auf einen bereits freigegebenen Speicherbereich zugegriffen wird, was zu undefiniertem Verhalten, Abstürzen oder der Ausführung von Schadcode führen kann.

Verlustfunktion
Eine Funktion, die den Unterschied zwischen den vorhergesagten Ausgaben eines

Modells und den tatsächlichen Zielwerten quantifiziert und somit als Grundlage für die Optimierung der Modellparameter dient.

Vitalitätsprüfung
Verfahren zur Laufzeitüberwachung. Mittels dieser Prüfungen wird sichergestellt, dass bestimmte Programmteile oder Kontrollflüsse erreichbar und aktiv sind oder korrekt ausgeführt werden können.

XML-Parser
Ein spezieller Parser (siehe oben), der XML-Datenstrukturen analysiert und in eine weiterverarbeitbare Form umwandelt.

Xtensa
Eine energieeffiziente, anpassbare Mikroprozessorarchitektur von Tensilica, häufig in IoT-Geräten wie dem ESP32 (siehe oben) eingesetzt.

XXE-Injection (XML External Entity Injection)
Eine Sicherheitslücke in XML-Verarbeitungsroutinen, bei der bösartige externe Entitäten in XML-Dokumenten eingebracht werden können, was zu einer Ausführung unerwünschter Aktionen führt.

Zeilenabdeckung
Eine Metrik der Codeabdeckung, die angibt, wie viele einzelne Programmzeilen mindestens einmal ausgeführt wurden.

Zweigabdeckung
Eine Metrik der Codeabdeckung, die überprüft, wie viele Verzweigungen (z. B. `if`-Bedingungen) im Programmcode durchlaufen wurden.

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

Eigene Arbeiten

[Bau+19] Ingmar Baumgart, **Matthias Börsig**, Niklas Goerke, Timon Hackenjos, Jochen Rill und Marek Wehmer. „Who Controls Your Energy? On the (In)Security of Residential Battery Energy Storage Systems“. In: *2019 IEEE International Conference on Communications, Control, and Computing Technologies for Smart Grids, SmartGridComm 2019, Beijing, China, October 21-23, 2019*. IEEE, Okt. 2019, S. 1–6. DOI: 10.1109/smartgridcomm.2019.8909749. URL: https://doi.org/10.1109/SmartGridComm.2019.8909749.

[Bör+20] **Matthias Börsig**, Sven Nitzsche, Max Eisele, Roland Gröll, Jürgen Becker und Ingmar Baumgart. „Fuzzing Framework for ESP32 Microcontrollers“. In: *2020 IEEE International Workshop on Information Forensics and Security (WIFS)*. IEEE, Dez. 2020, S. 1–6. DOI: 10.1109/wifs49906.2020.9360889. URL: https://ieeexplore.ieee.org/document/9360889.

[Kie+22] Valentin Kiechle, **Matthias Börsig**, Sven Nitzsche, Ingmar Baumgart und Jürgen Becker. „PREUNN: Protocol Reverse Engineering using Neural Networks“. In: *Proceedings of the 8th International Conference on Information Systems Security and Privacy - ICISSP*. **ICISSP 2022 Best Poster Award**. INSTICC. SciTePress, Feb. 2022, S. 345–356. ISBN: 978-989-758-553-1. DOI: 10.5220/0010813500003120. URL: https://www.scitepress.org/Link.aspx?doi=10.5220/0010813500003120.

[Kna+25] Leonard Knapp, Sven Nitzsche, **Matthias Börsig**, Alexandru Vasilache, Ingmar Baumgart und Juergen Becker. „Efficacy of Spiking Neural Networks for Intrusion Detection Systems“. In: *2025 International Conference on Cybersecurity and AI-Based Systems (Cyber-AI)*. IEEE Computer Society, Sep. 2025, S. 89–95. DOI: 10.1109/Cyber-AI66431.2025.11233776.

[Mhi+25] Ibrahim Mhiri, **Matthias Börsig**, Akim Stark und Ingmar Baumgart. „How to Train Your Llama – Efficient Grammar-Based Application Fuzzing Using Large Language Models“. In: *Secure IT Systems: 29th Nordic Conference, NordSec 2024 Karlstad, Sweden, November 6–7, 2024 Proceedings*. Hrsg. von Leonardo Horn Iwaya, Liina Kamm, Leonardo Martucci und Tobias Pulls. Bd. 15396. Lecture Notes in Computer Science. Karlstad, Sweden: Springer-Verlag, Jan. 2025, S. 239–257. ISBN: 978-3-031-79006-5. DOI: 10.1007/978-3-031-79007-2_13. URL: https://dx.doi.org/10.1007/978-3-031-79007-2_13.

[Pla+25] Benjamin Plach, **Matthias Börsig**, Maximilian Müller, Roland Gröll, Martin Dukek und Ingmar Baumgart. „Binary-Level Code Injection for Automated Tool Support on the ESP32 Platform“. In: *Secure IT Systems: 29th Nordic Conference, NordSec 2024 Karlstad, Sweden, November 6–7, 2024 Proceedings*. Hrsg.

von Leonardo Horn Iwaya, Liina Kamm, Leonardo Martucci und Tobias Pulls. Bd. 15396. Lecture Notes in Computer Science. Karlstad, Sweden: Springer-Verlag, Jan. 2025, S. 121–138. ISBN: 978-3-031-79006-5. DOI: 10.1007/978-3-031-79007-2_7. URL: https://dx.doi.org/10.1007/978-3-031-79007-2_7.

Weitere Literatur

[AAZ25] Atif Ali, Syed Adnan Ali und Nawal Zaheer. „The Role of ESP32 in Enabling Industry 4.0 and 5.0: A Comprehensive Narrative Review of Edge Intelligence, Human-Centric Automation, and Sustainable Innovation“. In: *Preprints* (Aug. 2025). DOI: 10.20944/preprints202508.0014.v1. URL: https://doi.org/10.20944/preprints202508.0014.v1.

[AC22] Anastasios Andronidis und Cristian Cadar. „SnapFuzz: high-throughput fuzzing of network applications“. In: *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2022. Virtual, South Korea: Association for Computing Machinery, 2022, S. 340–351. ISBN: 9781450393799. DOI: 10.1145/3533767.3534376. URL: https://doi.org/10.1145/3533767.3534376.

[AIB11] Andrea Arcuri, Muhammad Zohaib Iqbal und Lionel Briand. „Random testing: Theoretical results and practical implications“. In: *IEEE Transactions on Software Engineering* 38.2 (2011), S. 258–277.

[Ais25] Gudur Aishwarya. *Elements of Network protocol*. https://www.geeksforgeeks.org/computer-networks/elements-of-network-protocol/. Juli 2025.

[ANV11] João Antunes, Nuno Neves und Paulo Veríssimo. „Reverse Engineering of Protocols from Network Traces“. In: *Proceedings of the 2011 18th Working Conference on Reverse Engineering*. WCRE ’11. IEEE Computer Society, Okt. 2011, S. 169–178. ISBN: 9780769545820. DOI: 10.1109/WCRE.2011.28.

[ASZ25] José Antonio Amaya Zamudio, Marius Smytzek und Andreas Zeller. „FANDANGO: Evolving Language-Based Testing“. In: *Journal of the ACM (JACM)*. Bd. 2. ISSTA. New York, NY, USA: Association for Computing Machinery, Juni 2025. DOI: 10.1145/3728915. URL: https://doi.org/10.1145/3728915.

[Aun10] Benjamin Aunkofer. *Open System Interconnection – Referenzmodell*. https://www.der-wirtschaftsingenieur.de/index.php/open-system-interconnection-referenzmodell/. Jan. 2010.

[Ban+19] Andrew Banks, Ed Briggs, Ken Borgendale und Rahul Gupta. *MQTT Version 5.0*. Standard v5.0. OASIS Message Queuing Telemetry Transport (MQTT) TC, März 2019. URL: https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.html.

[BDB99] Vasanth Bala, Evelyn Duesterwald und Sanjeev Banerjia. *Transparent Dynamic Optimization: The Design and Implementation of Dynamo*. 1999. URL: https://homes.cs.washington.edu/~bodik/ucb/cs703-2002/papers/dynamo-full.pdf.

[BH19] Katharina Bogad und Manuel Huber. „Harzer Roller: Linker-Based Instrumentation for Enhanced Embedded Security Testing“. In: *Proceedings of the 3rd Reversing and Offensive-Oriented Trends Symposium*. ROOTS'19. Vienna, Austria: Association for Computing Machinery, 2019. ISBN: 9781450377751. DOI: 10.1145/3375894.3375897. URL: https://doi.org/10.1145/3375894.3375897.

[Bha22] Soumalya Bhattacharyya. *Understanding Fuzzing in Software Testing*. https://www.analyticssteps.com/blogs/understanding-fuzzing-software-testing. Analytics Steps. Nov. 2022.

[Bha25] Sakshi Bhakhra. *Difference Between Stateless and Stateful Protocol*. https://www.geeksforgeeks.org/computer-networks/difference-between-stateless-and-stateful-protocol/. Juli 2025.

[BKG20] Dor Bank, Noam Koenigstein und Raja Giryes. *Autoencoders*. März 2020. DOI: 10.48550/arXiv.2003.05991.

[Bla25] Blackduck, Inc. *The Heartbleed Bug*. https://www.heartbleed.com/. März 2025.

[BLP05] Fernando Bação, Victor Lobo und Marco Painho. „Self-organizing maps as substitutes for k-means clustering“. In: *International Conference on Computational Science*. Springer. 2005, S. 476–483.

[Blu23] Bluetooth SIG. *Bluetooth Core Specification*. Version 5.4. Bluetooth Special Interest Group. 2023. URL: https://www.bluetooth.com/specifications/bluetooth-core-specification/.

[BMC20] Marcel Böhme, Valentin J M Manès und Sang Kil Cha. „Boosting fuzzer efficiency: an information theoretic perspective“. In: *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. Virtual Event USA: ACM, Nov. 2020.

[BP23] David Belson und Lucas Pardue. *Examining HTTP/3 usage one year on*. https://blog.cloudflare.com/http3-usage-one-year-on/. Juni 2023.

[Bra+08] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler und François Yergeau. *Extensible Markup Language (XML) 1.0*. Fifth Edition. W3C. Nov. 2008. URL: https://www.w3.org/TR/xml/.

[Bro18] Chris Brook. *What is Deep Packet Inspection? How It Works, Use Cases for DPI, and More*. https://digitalguardian.com/blog/what-deep-packet-inspection-how-it-works-use-cases-dpi-and-more. Dez. 2018.

[BSI25] BSI. *Botnetze – Auswirkungen und Schutzmaßnahmen*. https://www.bsi.bund.de/DE/Themen/Verbraucherinnen-und-Verbraucher/Cyber-Sicherheitslage/Methoden-der-Cyber-Kriminalitaet/Botnetze/botnetze_node.html. 2025.

[BSM22] Marcel Böhme, László Szekeres und Jonathan Metzman. „On the reliability of coverage-based fuzzer benchmarking“. In: *Proceedings of the 44th International Conference on Software Engineering*. ICSE '22. Pittsburgh, Pennsylvania: Association for Computing Machinery, 2022, S. 1621–1633. ISBN: 9781450392211. DOI: 10.1145/3510003.3510230. URL: https://doi.org/10.1145/3510003.3510230.

[Bun+21] Joshua Bundt, Andrew Fasano, Brendan Dolan-Gavitt, William Robertson und Tim Leek. „Evaluating Synthetic Bugs“. In: *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*. ASIA CCS '21. Virtual Event, Hong Kong: Association for Computing Machinery, 2021, S. 716–730. ISBN: 9781450382878. DOI: 10.1145/3433210.3453096. URL: https://doi.org/10.1145/3433210.3453096.

[Cad22] Cadence Design Systems, Inc. *Xtensa®Instruction Set Architecture (ISA) Summary*. Modification: 737871. https://www.cadence.com/content/dam/cadence-www/global/en_US/documents/tools/silicon-solutions/compute-ip/isa-summary.pdf. Apr. 2022.

[Cad24] Cadence Design Systems, Inc. *Xtensa LX7 Processor*. https://www.cadence.com/en_US/home/resources/product-briefs/xtensa-lx7-processor-pb.html. 2024.

[Cad25] Cadence Design Systems, Inc. *Xtensa LX6 Customizable DPU*. https://www.electronicspecifier.com/wp-content/uploads/2025/07/Cadence-Xtensa-LX6-datasheet.pdf. 2025.

[Cha+17] Buddhika Chamith, Bo Joel Svensson, Luke Dalessandro und Ryan R. Newton. „Instruction punning: lightweight instrumentation for x86-64“. In: *SIGPLAN Not.* 52.6 (Juni 2017), S. 320–332. ISSN: 0362-1340. URL: https://doi.org/10.1145/3140587.3062344.

[Che+19] Yuanliang Chen, Yu Jiang, Fuchen Ma, Jie Liang, Mingzhe Wang, Chijin Zhou, Xun Jiao und Zhuo Su. „EnFuzz: Ensemble Fuzzing with Seed Synchronization among Diverse Fuzzers“. In: *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, S. 1967–1983. ISBN: 978-1-939133-06-9. URL: https://www.usenix.org/conference/usenixsecurity19/presentation/chen-yuanliang.

[Che18] Guillaume Chevalier. *LARNN: Linear Attention Recurrent Neural Network*. 2018. arXiv: 1808.05578 [cs.LG]. URL: https://arxiv.org/abs/1808.05578.

[Cho13] Fred Chow. „Intermediate representation“. In: *Communications of the ACM* 56.12 (Dez. 2013), S. 57–62. ISSN: 1557-7317. DOI: 10.1145/2534706.2534720. URL: http://dx.doi.org/10.1145/2534706.2534720.

[Cir25] Circuitlabs. *Chapter 8: Understanding ESP32 Boot Process*. https://circuitlabs.net/understanding-esp32-boot-process/. Mai 2025.

[CKW07] Weidong Cui, Jayanthkumar Kannan und Helen J Wang. „Discoverer: Automatic Protocol Reverse Engineering from Network Traces.“ In: *USENIX Security Symposium*. 2007, S. 1–14.

[Cla09] Justin Clarke. *SQL injection attacks and defense*. Elsevier, 2009. DOI: 10.1016/B978-1-59749-424-3.X0001-1.

[Cle+20] Abraham A Clements, Eric Gustafson, Tobias Scharnowski, Paul Grosen, David Fritz, Christopher Kruegel, Giovanni Vigna, Saurabh Bagchi und Mathias Payer. „HALucinator: Firmware Re-hosting Through Abstraction Layer Emulation“. In: *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, S. 1201–1218. ISBN: 978-1-939133-17-5. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/clements.

[CMS13] Ricardo J. G. B. Campello, Davoud Moulavi und Joerg Sander. „Density-Based Clustering Based on Hierarchical Density Estimates“. In: *Advances in Knowledge Discovery and Data Mining*. Hrsg. von Jian Pei, Vincent S. Tseng, Longbing Cao, Hiroshi Motoda und Guandong Xu. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, S. 160–172. ISBN: 978-3-642-37456-2.

[Com+09] Paolo Milani Comparetti, Gilbert Wondracek, Christopher Kruegel und Engin Kirda. „Prospex: Protocol specification extraction“. In: *2009 30th IEEE Symposium on Security and Privacy*. IEEE. 2009, S. 110–125.

[Con23] Connectivity Standards Alliance. *Zigbee Specification*. Connectivity Standards Alliance. 2023. URL: https://csa-iot.org/wp-content/uploads/2023/04/05-3474-23-csg-zigbee-specification-compressed.pdf.

[Cor19] Mitre Corporation. *2019 CWE Top 25 Most Dangerous Software Errors*. https://cwe.mitre.org/top25/archive/2019/2019_cwe_top25.html. 2019.

[DB25] Sena Dikici und Turgay Tugay Bilgin. „Advancements in automated program repair: a comprehensive review“. In: *Knowledge and Information Systems* 67.6 (2025), S. 4737–4783. ISSN: 0219-3116. DOI: 10.1007/s10115-025-02383-9.

[DEM94] Peter Deutsch, Alan Emtage und April Marine. *How to Use Anonymous FTP*. Techn. Ber. 1635. Internet Engineering Task Force (IETF), März 1994. 13 S. DOI: 10.17487/RFC1635. URL: https://www.rfc-editor.org/info/rfc1635.

[Den+23] Yinlin Deng, Chunqiu Steven Xia, Haoran Peng, Chenyuan Yang und Lingming Zhang. *Large Language Models are Zero-Shot Fuzzers: Fuzzing Deep-Learning Libraries via Large Language Models*. 2023. arXiv: 2212.14834 [cs.SE]. URL: https://doi.org/10.48550/arXiv.2212.14834.

[Den+24] Y. Deng, C. Xia, C. Yang, S. Zhang, S. Yang und L. Zhang. „Large Language Models are Edge-Case Generators: Crafting Unusual Programs for Fuzzing Deep Learning Libraries“. In: *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. ICSE '24. Lisbon, Portugal: IEEE Computer Society, Apr. 2024, S. 830–842. URL: https://doi.org/10.1145/3597503.3623343.

[Dev+19] Jacob Devlin, Ming-Wei Chang, Kenton Lee und Kristina Toutanova. *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*. 2019. arXiv: 1810.04805 [cs.CL]. URL: https://arxiv.org/abs/1810.04805.

[DGR20] Gregory J. Duck, Xiang Gao und Abhik Roychoudhury. „Binary rewriting without control flow recovery“. In: *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*. Hrsg. von Alastair F. Donaldson und Emina Torlak. ACM, 2020, S. 151–163. URL: https://doi.org/10.1145/3385412.3385972.

[DGZ23] Rafael Dutra, Rahul Gopinath und Andreas Zeller. „FormatFuzzer: Effective Fuzzing of Binary File Formats“. In: *ACM Trans. Softw. Eng. Methodol.* 33.2 (Dez. 2023). ISSN: 1049-331X. DOI: 10.1145/3628157. URL: https://doi.org/10.1145/3628157.

[DHS11] John Duchi, Elad Hazan und Yoram Singer. „Adaptive Subgradient Methods for Online Learning and Stochastic Optimization“. In: *Journal of Machine Learning Research* 12.61 (2011), S. 2121–2159. URL: http://jmlr.org/papers/v12/duchi11a.html.

[EFI21] Maialen Eceiza-Olaizola, Jose Luis Flores und Mikel Iturbe. „Fuzzing the Internet of Things: A Review on the Techniques and Challenges for Efficient Vulnerability Discovery in Embedded Systems“. In: *IEEE Internet of Things Journal* PP (Feb. 2021), S. 1–1. DOI: 10.1109/JIOT.2021.3056179.

[Eis+22] Max Eisele, Marcello Maugeri, Rachna Shriwas, Christopher Huth und Giampaolo Bella. „Embedded fuzzing: a review of challenges, tools, and solutions“. In: *Cybersecurity* 5 (Sep. 2022). DOI: 10.1186/s42400-022-00123-y.

[Eis+23] Max Eisele, Daniel Ebert, Christopher Huth und Andreas Zeller. „Fuzzing Embedded Systems using Debug Interfaces“. In: *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2023. Seattle, WA, USA: Association for Computing Machinery, 2023, S. 1031–1042. DOI: 10.1145/3597926.3598115. URL: https://doi.org/10.1145/3597926.3598115.

[Eis+25] Max Eisele, Johannes Hägele, Christopher Huth und Andreas Zeller. „GDBMiner: Mining Precise Input Grammars on (Almost) Any System“. In: *Leibniz Transactions on Embedded Systems* 10.1 (2025), 1:1–1:26. ISSN: 2199-2002. DOI: 10.4230/LITES.10.1.1. URL: https://drops.dagstuhl.de/entities/document/10.4230/LITES.10.1.1.

[Eli22] Michael Eling. *Der ESP32: Ein leistungsstarker Mikrocontroller für IoT-Anwendungen*. https://techgeeks.de/der-esp32-ein-leistungsstarker-mikrocontroller-fuer-iot-anwendungen/. Dez. 2022.

[Esp18] Espressif. *Espressif Achieves the 100-Million Target for IoT Chip Shipments*. https://www.espressif.com/en/news/Espressif_Achieves_the_Hundredmillion_Target_for_IoT_Chip_Shipments. 2018.

[Esp19] Espressif. *QEMU fork with ESP32 support*. https://github.com/espressif/qemu. 2019.

[Esp23] Espressif Systems. *ESP8266EX Datasheet*. Version 7.0. https://www.espressif.com/sites/default/files/documentation/0a-esp8266ex_datasheet_en.pdf. Juni 2023.

[Esp25a] Espressif Systems. *ESP32 Series Datasheet*. Version 5.0. https://www.espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf. Aug. 2025.

[Esp25b] Espressif Systems. *ESP32-C3 Series*. Version 2.2. https://www.espressif.com/sites/default/files/documentation/esp32-c3_datasheet_en.pdf. 2025.

[Esp25c] Espressif Systems. *ESP32-WROOM-32 Datasheet*. Version 3.6. https://www.espressif.com/sites/default/files/documentation/esp32-wroom-32_datasheet_en.pdf. Aug. 2025.

[Est+96] Martin Ester, Hans-Peter Kriegel, Jörg Sander und Xiaowei Xu. „A density-based algorithm for discovering clusters in large spatial databases with noise“. In: *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*. KDD'96. Portland, Oregon: AAAI Press, 1996, S. 226–231.

[Exp25] Express.js Team. *Express 5.x - API Reference*. https://expressjs.com/en/api.html. 2025.

[FC17] Rong Fan und Yaoyao Chang. „Machine learning for black-box fuzzing of network protocols“. In: *International Conference on Information and Communications Security*. Springer. 2017, S. 621–632.

[FDC20] Andrea Fioraldi, Daniele Cono D'Elia und Emilio Coppa. „WEIZZ: Automatic Grey-Box Fuzzing for Structured Binary Formats“. In: *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2020. Virtual Event, USA: Association for Computing Machinery, 2020, S. 1–13. ISBN: 9781450380089. DOI: 10.1145/3395363.3397372. URL: https://doi.org/10.1145/3395363.3397372.

[FDQ20] Andrea Fioraldi, Daniele Cono D'Elia und Leonardo Querzoni. „Fuzzing Binaries for Memory Safety Errors with QASan“. In: *2020 IEEE Secure Development (SecDev)*. 2020, S. 23–30. DOI: 10.1109/SecDev45635.2020.00019.

[Fie00] Roy Thomas Fielding. „Architectural Styles and the Design of Network-based Software Architectures“. AAI9980887. Diss. University of California, Irvine, 2000. ISBN: 0599871180. URL: https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm.

[Fio+20] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt und Marc Heuse. „AFL++ : Combining Incremental Steps of Fuzzing Research“. In: *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association, Aug. 2020. URL: https://www.usenix.org/conference/woot20/presentation/fioraldi.

[For20] Fortune Business Insights. *Internet of Things (IoT) Market Size, Share and Industry Analysis By Platform (Device Management, Application Management, Network Management), By Software & Services (Software Solution, Services), By End-Use Industry (BFSI, Retail, Governments, Healthcare, Others) And Regional Forecast, 2020-2027*. 2020.

[FPA17] Alessandro Di Federico, Mathias Payer und Giovanni Agosta. „rev.ng: a unified binary analysis framework to recover CFGs and function boundaries“. In: *Proceedings of the 26th International Conference on Compiler Construction, Austin, TX, USA, February 5-6, 2017*. Hrsg. von Peng Wu und Sebastian Hack. ACM, 2017, S. 131–141. URL: http://dl.acm.org/citation.cfm?id=3033028.

[FS00] Wenfei Fan und Jérôme Siméon. „Integrity constraints for XML“. In: *Proceedings of the Nineteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*. PODS '00. Dallas, Texas, USA: Association for Computing Machinery, 2000, S. 23–34. ISBN: 158113214X. DOI: 10.1145/335168.335172. URL: https://doi.org/10.1145/335168.335172.

[GAJ24] Matthew G. Gaber, Mohiuddin Ahmed und Helge Janicke. „Malware Detection with Artificial Intelligence: A Systematic Literature Review“. In: *ACM Comput. Surv.* 56.6 (Jan. 2024). ISSN: 0360-0300. DOI: 10.1145/3638552. URL: https://doi.org/10.1145/3638552.

[Gar08] Simson Garfinkel. *Nitroba University Harassment Scenario*. Dataset: https://digitalcorpora.org/corpora/scenarios/nitroba-university-harassment-scenario. Digital Corpora, Nov. 2008.

[Gas+15] Hugo Gascón, Christian Wressnegger, Fabian Yamaguchi, Daniel Arp und Konrad Rieck. „Pulsar: Stateful Black-Box Fuzzing of Proprietary Network Protocols“. In: *International Conference on Security and Privacy in Communication Systems*. Springer. Springer International Publishing, 2015, S. 330–347. ISBN: 978-3-319-28865-9.

[GGG22] Rahul Gopinath, Philipp Görz und Alex Groce. *Mutation Analysis: Answering the Fuzzing Challenge*. 2022. arXiv: 2201.11303 [cs.SE]. URL: https://arxiv.org/abs/2201.11303.

[GKL08] Patrice Godefroid, Adam Kiezun und Michael Y. Levin. „Grammar-based whitebox fuzzing“. In: *SIGPLAN Not.* 43.6 (Juni 2008), S. 206–215. ISSN: 0362-1340. DOI: 10.1145/1379022.1375607. URL: https://doi.org/10.1145/1379022.1375607.

[GLM08] Patrice Godefroid, Michael Y. Levin und David Molnar. „Automated Whitebox Fuzz Testing“. In: *Proceedings of the Network and Distributed System Security Symposium, NDSS 2008*. Bd. 8. The Internet Society, Nov. 2008, S. 151–166. URL: https://www.microsoft.com/en-us/research/publication/automated-whitebox-fuzz-testing/.

[Goo+14] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville und Yoshua Bengio. „Generative adversarial nets“. In: *Advances in neural information processing systems*. 2014, S. 2672–2680.

[Goo+19] Young-Hoon Goo, Kyu-Seok Shim, Min-Seob Lee und Myung-Sup Kim. *HTTP and DNS traffic traces for experimenting of protocol reverse engineering methods*. http://dx.doi.org/10.21227/tpqf-fe98. 2019. DOI: 10.21227/tpqf-fe98.

[Gop+18] Rahul Gopinath, Björn Mathis, Mathias Höschele, Alexander Kampmann und Andreas Zeller. *Sample-Free Learning of Input Grammars for Comprehensive Software Fuzzing*. 2018. arXiv: 1810.08289 [cs.SE]. URL: https://arxiv.org/abs/1810.08289.

[GPS17] Patrice Godefroid, Hila Peleg und Rishabh Singh. „Learn&Fuzz: Machine learning for input fuzzing“. In: *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Los Alamitos, CA, USA: IEEE Computer Society, 2017, S. 50–59. URL: https://doi.org/10.1109/ASE.2017.8115618.

[GT23] Javad Garshasbi und Mehdi Teimouri. „CNNPRE: A CNN-Based Protocol Reverse Engineering Method“. In: *IEEE Access* 11 (2023), S. 116255–116268. DOI: 10.1109/ACCESS.2023.3325391.

[Gug+22] Sylvain Gugger, Lysandre Debut, Thomas Wolf, Philipp Schmid, Zachary Mueller, Sourab Mangrulkar, Marc Sun und Benjamin Bossan. *Accelerate: Training and inference at scale made simple, efficient and adaptable*. https://github.com/huggingface/accelerate. 2022.

[Gui+20] Zhijie Gui, Hui Shu, Fei Kang und Xiaobing Xiong. „FIRMCORN: Vulnerability-Oriented Fuzzing of IoT Firmware via Optimized Virtual Execution“. In: *IEEE Access* 8 (2020), S. 29826–29841.

[Gup19] Aditya Gupta. *The IoT Hacker's Handbook: A Practical Guide to Hacking the Internet of Things*. en. 1. Aufl. APress, Apr. 2019, S. 340. ISBN: 1484242998. DOI: 10.1007/978-1-4842-4300-8.

[Has+19] Vikas Hassija, Vinay Chamola, Vikas Saxena, Divyansh Jain, Pranav Goyal und Biplab Sikdar. „A Survey on IoT Security: Application Areas, Security Threats, and Solution Architectures“. In: *IEEE Access* PP (Juni 2019), S. 1–1. DOI: 10.1109/ACCESS.2019.2924045.

[Hav+14] Nikolas Havrikov, Matthias Höschele, Juan Pablo Galeotti und Andreas Zeller. „XMLMate: evolutionary XML test generation“. In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. FSE 2014. Hong Kong, China: Association for Computing Machinery, 2014, S. 719–722. ISBN: 9781450330565. DOI: 10.1145/2635868.2661666. URL: https://doi.org/10.1145/2635868.2661666.

[Haw+17] William H. Hawkins, Jason D. Hiser, Michele Co, Anh Nguyen-Tuong und Jack W. Davidson. „Zipr: Efficient Static Binary Rewriting for Security“. In: *47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2017, Denver, CO, USA, June 26-29, 2017*. IEEE Computer Society, 2017, S. 559–566. URL: https://doi.org/10.1109/DSN.2017.27.

[HKZ17] Matthias Höschele, Alexander Kampmann und Andreas Zeller. *Active Learning of Input Grammars*. 2017. arXiv: 1708.08731 [cs.PL]. URL: https://arxiv.org/abs/1708.08731.

[HMU11] John E. Hopcroft, Rajeev Motwani und Jeffrey D. Ullman. *Einführung in die Automatentheorie, Formale Sprachen und Komplexitätstheorie*. de. 3., aktualisierte Auflage. Pearson Studium. Harlow, England: Pearson Deutschland, Feb. 2011, S. 256. ISBN: 9783868940824. URL: https://elibrary.pearson.de/book/99.150005/9783863265090.

[HN17] Jesse Hertz und Tim Newsham. *Project Triforce: Run AFL On Everything*. Whitepaper. https://www.nccgroup.com/research-blog/whitepaper-project-triforce-run-afl-on-everything-2017/. NCC Group, Apr. 2017.

[HS06] Geoffrey E. Hinton und R. Salakhutdinov. „Reducing the Dimensionality of Data with Neural Networks“. In: *Science* 313 (2006), S. 504–507.

[HS97] Sepp Hochreiter und Jürgen Schmidhuber. „Long short-term memory“. In: *Neural computation* 9.8 (1997), S. 1735–1780.

[HSS12] Geoffrey Hinton, Nitish Srivastava und Kevin Swersky. *Lecture 6e – RMSProp: Divide the gradient by a running average of its recent magnitude*. http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf. University of Toronto. 2012.

[HSW+89] Kurt Hornik, Maxwell Stinchcombe, Halbert White u. a. „Multilayer feedforward networks are universal approximators.“ In: *Neural networks* 2.5 (1989), S. 359–366.

[Hu+18] Zhicheng Hu, Jianqi Shi, YanHong Huang, Jiawen Xiong und Xiangxing Bu. „GANFuzz: a GAN-based industrial network protocol fuzzing framework“. In: *Proceedings of the 15th ACM International Conference on Computing Frontiers*. 2018, S. 138–145.

[Hu+22] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang und Weizhu Chen. „LoRA: Low-Rank Adaptation of Large Language Models“. In: *ICLR* 1.2 (2022), S. 3. URL: https://arxiv.org/abs/2106.09685.

[Hua+25] Linghan Huang, Peizhou Zhao, Huaming Chen und Lei Ma. *On the Challenges of Fuzzing Techniques via Large Language Models*. 2025. arXiv: 2402.00350 [cs.SE]. URL: https://arxiv.org/abs/2402.00350.

[HZY23] Jie Hu, Qian Zhang und Heng Yin. *Augmenting Greybox Fuzzing with Generative AI*. 2023. arXiv: 2306.06782 [cs.CR]. URL: https://doi.org/10.48550/arXiv.2306.06782.

[IHR06] Internet Architecture Board, Mark J. Handley und Eric Rescorla. *Internet Denial-of-Service Considerations*. Techn. Ber. 4732. Internet Engineering Task Force (IETF), Dez. 2006. 38 S. DOI: 10.17487/RFC4732. URL: https://www.rfc-editor.org/info/rfc4732.

[Jau+23] Patrick Jauernig, Domagoj Jakobovic, Stjepan Picek, Emmanuel Stapf und Ahmad-Reza Sadeghi. „DARWIN: Survival of the fittest fuzzing mutators". In: *Proceedings 2023 Network and Distributed System Security Symposium*. San Diego, CA, USA: Internet Society, 2023.

[Jia+24] Yu Jiang, Jie Liang, Fuchen Ma, Yuanliang Chen, Chijin Zhou, Yuheng Shen, Zhiyong Wu, Jingzhou Fu, Mingzhe Wang, ShanShan Li und Quan Zhang. *When Fuzzing Meets LLMs: Challenges and Opportunities*. 2024. arXiv: 2404.16297 [cs.SE]. URL: https://arxiv.org/abs/2404.16297.

[JJ20] YoungGiu Jung und Chang-Min Jeong. „Deep neural network-based automatic unknown protocol classification system using histogram feature". In: *The Journal of Supercomputing* 76.7 (2020), S. 5425–5441.

[KB17] Diederik P. Kingma und Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2017. arXiv: 1412.6980 [cs.LG]. URL: https://arxiv.org/abs/1412.6980.

[Kim+17] Taegyu Kim, Chung Hwan Kim, Hongjun Choi, Yonghwi Kwon, Brendan Saltaformaggio, Xiangyu Zhang und Dongyan Xu. „RevARM: A Platform-Agnostic ARM Binary Rewriter for Security Applications". In: *Proceedings of the 33rd Annual Computer Security Applications Conference, Orlando, FL, USA, December 4-8, 2017*. ACM, 2017, S. 412–424. URL: https://doi.org/10.1145/3134600.3134627.

[Kin21] Jörg Kindermann. *Generative Adversarial Networks (GANs) für maschinelle Übersetzung*. https://lamarr-institute.org/de/blog/generative-neuronale-modelle-gan/. Juni 2021.

[Koh82] Teuvo Kohonen. „Self-organized formation of topologically correct feature maps". In: *Biological cybernetics* 43.1 (1982), S. 59–69.

[Kre16] Brian Krebs. *Source Code for IoT Botnet 'Mirai' Released*. https://krebsonsecurity.com/2016/10/source-code-for-iot-botnet-mirai-released/. Okt. 2016.

[KSH12] Alex Krizhevsky, Ilya Sutskever und Geoffrey E Hinton. „ImageNet Classification with Deep Convolutional Neural Networks“. In: *Advances in Neural Information Processing Systems 25*. Hrsg. von F. Pereira, C. J. C. Burges, L. Bottou und K. Q. Weinberger. Curran Associates, Inc., 2012, S. 1097–1105. URL: http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf.

[KY24] Low Choon Keat und Tan Xuan Ying. „Artificial Intelligence-Based Email Spam Filtering“. In: *Journal of Advanced Research in Artificial Intelligence & It's Applications* 2.2 (Dez. 2024). DOI: 10.5281/zenodo.14264139. URL: https://doi.org/10.5281/zenodo.14264139.

[LAC21] Brian Lester, Rami Al-Rfou und Noah Constant. *The Power of Scale for Parameter-Efficient Prompt Tuning*. 2021. arXiv: 2104.08691 [cs.CL]. URL: https://doi.org/10.48550/arXiv.2104.08691.

[Lan99] Kevin J Lang. „Faster algorithms for finding minimal consistent DFAs“. In: *NEC Research Institute, Tech. Rep* (1999).

[Lem+23] Caroline Lemieux, Jeevana Priya Inala, Shuvendu K. Lahiri und Siddhartha Sen. „CodaMosa: Escaping Coverage Plateaus in Test Generation with Pre-Trained Large Language Models“. In: *Proceedings of the 45th International Conference on Software Engineering*. ICSE '23. Melbourne, Victoria, Australia: IEEE Press, 2023, S. 919–931. ISBN: 9781665457019. URL: https://doi.org/10.1109/ICSE48619.2023.00085.

[Li+18] R. Li, X. Xiao, S. Ni, H. Zheng und S. Xia. „Byte Segment Neural Network for Network Traffic Classification“. In: *2018 IEEE/ACM 26th International Symposium on Quality of Service (IWQoS)*. 2018, S. 1–10. DOI: 10.1109/IWQoS.2018.8624128.

[Lia+18] Hongliang Liang, Xiaoxiao Pei, Xiaodong Jia, Wuwei Shen und Jian Zhang. „Fuzzing: State of the Art“. In: *IEEE Transactions on Reliability* 67.3 (2018), S. 1199–1218. DOI: 10.1109/TR.2018.2834476.

[LMC23] Dongge Liu, Jonathan Metzman und Oliver Chang. *AI-Powered Fuzzing: Breaking the Bug Hunting Barrier*. https://security.googleblog.com/2023/08/ai-powered-fuzzing-breaking-bug-hunting.html. Google Open Source Security Team. Aug. 2023.

[Lop+17] Manuel Lopez-Martin, Belen Carro, Antonio Sanchez-Esguevillas und Jaime Lloret. „Network traffic classifier with convolutional and recurrent neural networks for Internet of Things“. In: *IEEE Access* 5 (2017), S. 18042–18050.

[Luk+05] Chi-Keung Luk, Robert S. Cohn, Robert Muth, Harish Patil, Artur Klauser, P. Geoffrey Lowney, Steven Wallace, Vijay Janapa Reddi und Kim M. Hazelwood. „Pin: building customized program analysis tools with dynamic instrumentation“. In: *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*. Hrsg. von Vivek Sarkar und Mary W. Hall. ACM, 2005, S. 190–200. URL: https://doi.org/10.1145/1065010.1065034.

[LZZ18] Jun Li, Bodong Zhao und Chao Zhang. „Fuzzing: a survey“. In: *Cybersecurity* 1.1 (2018), S. 6.

[Men+24] Ruijie Meng, Martin Mirchev, Marcel Böhme und Abhik Roychoudhury. „Large Language Model guided Protocol Fuzzing“. In: *Proceedings of the 31st Annual Network and Distributed System Security Symposium (NDSS)*. 2024.

[Mic+17] A. Michael, E. Valla, Natinael Solomon Neggatu und A. Moore. *Network traffic classification via neural networks*. Techn. Ber. UCAM-CL-TR-912. University of Cambridge, Computer Laboratory, Sep. 2017. URL: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-912.pdf.

[Mil21] Saša Miličević. *ESP32 Based Devices*. https://templates.blakadder.com/esp32.html. Jan. 2021.

[MRR12] Michael McCool, Arch D. Robison und James Reinders. *Structured Parallel Programming: Patterns for Efficient Computation*. Elsevier/Morgan Kaufmann, Juni 2012, S. 1–432. ISBN: 0124159931.

[MS12] Christoph Meinel und Harald Sack. „Die Grundlage des Internets: TCP/IP-Referenzmodell“. In: *Internetworking: Technische Grundlagen und Anwendungen*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, S. 31–67. ISBN: 978-3-540-92940-6. DOI: 10.1007/978-3-540-92940-6_2. URL: https://doi.org/10.1007/978-3-540-92940-6_2.

[Mue+18] Marius Muench, Jan Stijohann, Frank Kargl, Aurélien Francillon und Davide Balzarotti. „What You Corrupt Is Not What You Crash: Challenges in Fuzzing Embedded Devices.“ In: *NDSS*. 2018.

[Nat22] Roberto Natella. „StateAFL: Greybox Fuzzing for Stateful Network Servers“. In: *Empirical Software Engineering* 27.7 (2022), S. 191. ISSN: 1573-7616. DOI: 10.1007/s10664-022-10233-3. URL: https://doi.org/10.1007/s10664-022-10233-3.

[NB23] Antonio Nappa und Eduardo Blázquez. *Fuzzing Against the Machine*. en. Birmingham, England: Packt Publishing, Mai 2023.

[NH19] Stefan Nagy und Matthew Hicks. „Full-Speed Fuzzing: Reducing Fuzzing Overhead through Coverage-Guided Tracing“. In: *IEEE Symposium on Security and Privacy (Oakland)*. Mai 2019, S. 787–802. DOI: 10.1109/SP.2019.00069.

[Nie+99] Henrik Nielsen, Jeffrey Mogul, Larry M Masinter, Roy T. Fielding, Jim Gettys, Paul J. Leach und Tim Berners-Lee. *Hypertext Transfer Protocol – HTTP/1.1*. Techn. Ber. 2616. Internet Engineering Task Force (IETF), Juni 1999. 176 S. DOI: 10.17487/RFC2616. URL: https://www.rfc-editor.org/info/rfc2616.

[Nin+23] Bowei Ning, Xuejun Zong, Kan He und Lian Lian. „PREIUD: An Industrial Control Protocols Reverse Engineering Tool Based on Unsupervised Learning and Deep Neural Network Methods“. In: *Symmetry* 15.3 (2023). ISSN: 2073-8994. DOI: 10.3390/sym15030706. URL: https://www.mdpi.com/2073-8994/15/3/706.

[NMT24] Mohaddese Nemati, Shiva Mahmoudzadeh und Mehdi Teimouri. „Enhanced Autoencoder-Based Clustering for Message Analysis in Binary Protocols“. In: *2024 14th International Conference on Computer and Knowledge Engineering (ICCKE)*. 2024, S. 302–307. DOI: 10.1109/ICCKE65377.2024.10874790.

[NP21] Roberto Natella und Van-Thuan Pham. „ProFuzzBench: a benchmark for stateful protocol fuzzing“. In: *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2021. Virtual, Denmark: Association for Computing Machinery, 2021, S. 662–665. ISBN: 9781450384599. DOI: 10.1145/3460319.3469077. URL: https://doi.org/10.1145/3460319.3469077.

[Nym+17] Thomas Nyman, Jan-Erik Ekberg, Lucas Davi und N. Asokan. „CFI CaRE: Hardware-supported Call and Return Enforcement for Commercial Microcontrollers“. In: *CoRR* abs/1706.05715 (2017). arXiv: 1706.05715. URL: http://arxiv.org/abs/1706.05715.

[Pat24] Harsh Pathak. *Parameter-efficient fine-tuning (PEFT) and how it's different from fine-tuning*. https://medium.com/@harshnpathak/parameter-efficient-fine-tuning-peft-and-how-its-different-from-fine-tuning-3f6b95c73bac. Juli 2024.

[Pav+18] Esteban Pavese, Ezekiel Soremekun, Nikolas Havrikov, Lars Grunske und Andreas Zeller. *Inputs from Hell: Generating Uncommon Inputs from Common Samples*. 2018. arXiv: 1812.07525 [cs.SE]. URL: https://arxiv.org/abs/1812.07525.

[PBR20] Van-Thuan Pham, Marcel Böhme und Abhik Roychoudhury. „AFLNET: A Greybox Fuzzer for Network Protocols“. In: *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. 2020, S. 460–465. DOI: 10.1109/ICST46399.2020.00062.

[Pla20] Patrick von Platen. *How to generate text: using different decoding methods for language generation with Transformers*. März 2020. URL: https://huggingface.co/blog/how-to-generate.

[Pos80] Jon Postel. *User Datagram Protocol*. Techn. Ber. 768. Internet Engineering Task Force (IETF), Aug. 1980. 3 S. DOI: 10.17487/RFC0768. URL: https://www.rfc-editor.org/info/rfc768.

[Pos81a] Jon Postel. *RFC 791: Internet Protocol*. Request for Comments 791. Internet Engineering Task Force (IETF), Sep. 1981. DOI: doi.org/10.17487/RFC0791. URL: https://www.rfc-editor.org/rfc/rfc791.

[Pos81b] Jon Postel. *Transmission Control Protocol*. Techn. Ber. 793. Internet Engineering Task Force (IETF), Sep. 1981. 91 S. DOI: 10.17487/RFC0793. URL: https://www.rfc-editor.org/info/rfc793.

[PP03] Ruoming Pang und Vern Paxson. *Lawrence Berkeley National Laboratory - FTP - Packet Trace*. Dataset: https://ee.lbl.gov/anonymized-traces.html. Lawrence Berkeley National Laboratory, Jan. 2003.

[PR85] Jon Postel und Joyce Kathleen Reynolds. *File Transfer Protocol*. Techn. Ber. 959. Internet Engineering Task Force (IETF), Okt. 1985. 69 S. DOI: 10.17487/RFC0959. URL: https://www.rfc-editor.org/info/rfc959.

[Put+05] Ludo Van Put, Bjorn De Sutter, Matias Madou, Bruno De Bus, Dominique Chanet, Kristof Smits und Koen De Bosschere. „LANCET: a nifty code editing tool". In: *Proceedings of the 2005 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering, PASTE'05, Lisbon, Portugal, September 5-6, 2005*. Hrsg. von Michael D. Ernst und Thomas P. Jensen. ACM, 2005, S. 75–81. URL: https://doi.org/10.1145/1108792.1108812.

[QV15] Nguyen Anh Quynh und Dang Hoang Vu. *Unicorn-The ultimate CPU emulator*. https://www.unicorn-engine.org/. 2015.

[Rad+19] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei und Ilya Sutskever. „Language models are unsupervised multitask learners". In: *OpenAI Blog* 1.8 (2019), S. 9. URL: https://cdn.openai.com/better-language-models/language_models_are_unsupervised_multitask_learners.pdf.

[Raf+20] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li und Peter J. Liu. „Exploring the limits of transfer learning with a unified text-to-text transformer". In: *J. Mach. Learn. Res.* 21.1 (Jan. 2020). ISSN: 1532-4435.

[Raj+21] Samyam Rajbhandari, Olatunji Ruwase, Jeff Rasley, Shaden Smith und Yuxiong He. „ZeRO-infinity: breaking the GPU memory wall for extreme scale deep learning". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '21. St. Louis, Missouri: Association for Computing Machinery, 2021. ISBN: 9781450384421. URL: https://doi.org/10.1145/3458817.3476205.

[Ras+20] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase und Yuxiong He. „DeepSpeed: System Optimizations Enable Training Deep Learning Models with Over 100 Billion Parameters". In: *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. KDD '20. Virtual Event, CA, USA: Association for Computing Machinery, 2020, S. 3505–3506. ISBN: 9781450379984. URL: https://doi.org/10.1145/3394486.3406703.

[Ras24] Raspberry Pi Ltd. *Raspberry Pi Pico Datasheet*. build-version: eec2b0c-clean. https://datasheets.raspberrypi.com/pico/pico-datasheet.pdf. Okt. 2024.

[Rau23] Nico Rausch. „Evaluation eines Machine-learning-basierten Ansatzes zum Protocol Reverse Engineering für effizientes Fuzzing von Netzwerkanwendungen". Betreuer: **Matthias Börsig** und Martin Dukek, Erstgutachter: PD Dr.-Ing. Ingmar Baumgart, Zweitgutachter: Prof. Dr. Ralf H. Reussner. Masterarbeit. Postfach 6980, 76128 Karlsruhe: Karlsruher Institut für Technologie, Sep. 2023.

[Red+21] Monalika Padma Reddy, Sheba Selvam, Meghana Achandar und Ashwitha NA. „Human Activity Recognition using 3D CNN“. In: *Turkish Online Journal of Qualitative Inquiry (TOJQI)* 12.7 (2021), S. 12898–12908. DOI: 10.13140/RG.2.2.20520.49923. URL: https://tojqi.net/index.php/journal/article/view/6572/4681.

[RMC15] Alec Radford, Luke Metz und Soumith Chintala. „Unsupervised representation learning with deep convolutional generative adversarial networks“. In: *arXiv preprint arXiv:1511.06434* (2015).

[Rud17] Sebastian Ruder. *An overview of gradient descent optimization algorithms*. 2017. arXiv: 1609.04747 [cs.LG]. URL: https://arxiv.org/abs/1609.04747.

[Sai11] Peter Saint-Andre. *Extensible Messaging and Presence Protocol (XMPP): Core*. Techn. Ber. 6120. Internet Engineering Task Force (IETF), März 2011. 211 S. DOI: 10.17487/RFC6120. URL: https://www.rfc-editor.org/info/rfc6120.

[SBF22] Eric Schulte, Michael D. Brown und Vlad Folts. „A Broad Comparative Evaluation of x86-64 Binary Rewriters“. In: *CSET 2022: Cyber Security Experimentation and Test Workshop, Virtual Event, 8 August 2022*. ACM, 2022, S. 129–144. URL: https://doi.org/10.1145/3546096.3546112.

[SBV23] Andrei Simion, Calin Bira und Valentin-Gabriel Voiculescu. „Embedded platform characterization for interface throughput and computing power in common 8/16/32-bit platforms“. In: *Advanced Topics in Optoelectronics, Microelectronics, and Nanotechnologies XI*. Hrsg. von Marian Vladescu, Razvan D. Tamas und Ionica Cristea. Bd. 12493. International Society for Optics und Photonics. SPIE, 2023, S. 1249323. DOI: 10.1117/12.2643278. URL: https://doi.org/10.1117/12.2643278.

[Sch08] Henning Schulzrinne. *Textual vs. Binary Protocols*. https://www.cs.columbia.edu/sip/textual-binary.html. Jan. 2008.

[Sco+03] Kevin Scott, Naveen Kumar, S. Velusamy, Bruce R. Childers, Jack W. Davidson und Mary Lou Soffa. „Retargetable and Reconfigurable Software Dynamic Translation“. In: *1st IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2003), 23-26 March 2003, San Francisco, CA, USA*. Hrsg. von Richard Johnson, Tom Conte und Wen-mei W. Hwu. IEEE Computer Society, 2003, S. 36–47. URL: https://doi.org/10.1109/CGO.2003.1191531.

[Sel+17] Ramprasaath R Selvaraju, Michael Cogswell, Abhishek Das, Ramakrishna Vedantam, Devi Parikh und Dhruv Batra. „Grad-CAM: Visual Explanations from Deep Networks via Gradient-based Localization“. In: *Proceedings of the IEEE international conference on computer vision*. 2017, S. 618–626.

[SEV01] Amitabh Srivastava, Andrew Edwards und Hoi Vo. *Vulcan: Binary Transformation In A Distributed Environment*. 2001. URL: https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/tr-2001-50.pdf.

[SGA07] Michael Sutton, Adam Greene und Pedram Amini. *Fuzzing: Brute Force Vulnerability Discovery*. Boston, MA: Addison-Wesley Educational, Juni 2007, S. 576. ISBN: 9780321446114.

[Sha+19] Iman Sharafaldin, Arash Habibi Lashkari, Saqib Hakak und Ali A Ghorbani. „Developing Realistic Distributed Denial of Service (DDoS) Attack Dataset and Taxonomy“. In: *2019 International Carnahan Conference on Security Technology (ICCST)*. IEEE. 2019, S. 1–8.

[Sha05] Yakov Shafranovich. *Common Format and MIME Type for Comma-Separated Values (CSV) Files*. Techn. Ber. 4180. Internet Engineering Task Force (IETF), Okt. 2005. 8 S. DOI: 10.17487/RFC4180. URL: https://www.rfc-editor.org/info/rfc4180.

[Sha25] Sanjeev Sharma. *Tanh vs. Sigmoid vs. ReLU*. https://www.geeksforgeeks.org/deep-learning/tanh-vs-sigmoid-vs-relu/. Juli 2025.

[SHB14] Zach Shelby, Klaus Hartke und Carsten Bormann. *The Constrained Application Protocol (CoAP)*. Techn. Ber. 7252. Internet Engineering Task Force (IETF), Juni 2014. 112 S. DOI: 10.17487/RFC7252. URL: https://www.rfc-editor.org/info/rfc7252.

[She+19] Dongdong She, Kexin Pei, Dave Epstein, Junfeng Yang, Baishakhi Ray und Suman Jana. „NEUZZ: Efficient Fuzzing with Neural Program Smoothing“. In: *2019 IEEE Symposium on Security and Privacy (SP)*. 2019, S. 803–817. DOI: 10.1109/SP.2019.00052.

[Shi+12] Ali Shiravi, Hadi Shiravi, Mahbod Tavallaee und Ali A Ghorbani. „Toward developing a systematic approach to generate benchmark datasets for intrusion detection“. In: *computers & security* 31.3 (2012), S. 357–374.

[SM07] Karen Scarfone und Peter Mell. *Guide to Intrusion Detection and Prevention Systems (IDPS)*. Techn. Ber. Special Publication 800-94. Gaithersburg, MD: National Institute of Standards und Technology (NIST), Feb. 2007. URL: https://csrc.nist.gov/publications/detail/sp/800-94/final.

[Spä+16] Christopher Späth, Christian Mainka, Vladislav Mladenov und Jörg Schwenk. „SoK: XML Parser Vulnerabilities“. In: *10th USENIX Workshop on Offensive Technologies (WOOT 16)*. Austin, TX: USENIX Association, Aug. 2016. URL: https://www.usenix.org/conference/woot16/workshop-program/presentation/spath.

[Sri+14] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever und Ruslan Salakhutdinov. „Dropout: A Simple Way to Prevent Neural Networks from Overfitting“. In: *Journal of Machine Learning Research* 15.56 (2014), S. 1929–1958. URL: http://jmlr.org/papers/v15/srivastava14a.html.

[SS19] Hamad Ali Al Salem und Jia Song. „A Review on Grammar-Based Fuzzing Techniques“. English. In: *International Journal of Computer Science and Security (IJCSS)* 13.3 (Juni 2019). Hrsg. von Editor, S. 114–123. ISSN: 1985-1553. URL: http://www.cscjournals.org/library/manuscriptinfo.php?mc=IJCSS-1481.

[STM25] STMicroelectronics. *STM32F401xD STM32F401xE Datasheet*. DS10086 Rev 4. https://www.st.com/resource/en/datasheet/stm32f401re.pdf. Jan. 2025.

[Sto10] Dan (Mcld) Stowell. *Somtraining.svg*. https://commons.wikimedia.org/wiki/File:Somtraining.svg. Mai 2010.

[Str25] Cole Stryker. *What are large language models (LLMs)?* https://www.ibm.com/think/topics/large-language-models. Sep. 2025.

[SVL14] Ilya Sutskever, Oriol Vinyals und Quoc V Le. „Sequence to sequence learning with neural networks". In: *Advances in neural information processing systems*. 2014, S. 3104–3112.

[SWS07] Anoop Singhal, Theodore Winograd und Karen Scarfone. *Guide to Secure Web Services*. Techn. Ber. Special Publication 800-95. https://nvlpubs.nist.gov/nistpubs/legacy/sp/nistspecialpublication800-95.pdf. Gaithersburg, MD: National Institute of Standards und Technology (NIST), Aug. 2007.

[TC984] ISO TC97. „Basic reference model". In: *International Standard, ISO/IS* 7498 (1984).

[TDM08] A. Takanen, J.D. Demott und C. Miller. *Fuzzing for Software Security Testing and Quality Assurance*. Artech House information security and privacy series. Artech House, 2008. ISBN: 9781596932159. URL: https://books.google.de/books?id=tMuAc_y9dFYC.

[TG00] Caroline Tice und Susan L. Graham. *Key Instructions: Solving the Code Location Problem for Optimized Code*. Research Report. https://www.researchgate.net/publication/2432347_Key_Instructions_Solving_the_Code_Location_Problem_for_Optimized_Code. 130 Lytton Avenue, Palo Alto, California 94301: Compaq Systems Research Center, Sep. 2000, S. 30.

[Tou+23] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale u. a. *Llama 2: Open foundation and fine-tuned chat models*. 2023. arXiv: 2307.09288 [cs.CL]. URL: https://doi.org/10.48550/arXiv.2307.09288.

[Tou+24] Asma Touqir, Faisal Iradat, Abdur Rakib, Nazim Taskin, Hesamaldin Jadidbonab, Zaheeruddin Asif und Olivier Haas. *Systematic Review of Fuzzing in IoT: Evaluating Techniques, Vulnerabilities, and Research Gaps*. Aug. 2024. DOI: 10.21203/rs.3.rs-4963553/v1.

[TW11] Andrew S. Tanenbaum und David Wetherall. *Computer networks*. 5th ed. Boston: Prentice Hall, 2011. ISBN: 9780133485936. URL: https://learning.oreilly.com/library/view/-/9780133485936/?ar.

[Uni20] Unit 42. *2020 Unit 42 IoT Threat Report*. https://unit42.paloaltonetworks.com/iot-threat-report-2020. 2020.

[Vas+17] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser und Illia Polosukhin. „Attention is All you Need". In: *Advances in Neural Information Processing Systems*. Hrsg. von I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan und R. Garnett. Bd. 30. Curran Associates, Inc., 2017. URL: https://arxiv.org/abs/1706.03762.

[Vis+11] Arun Viswanathan, Alefiya Hussain, Jelena Mirkovic, Stephen Schwab und John Wroclawski. „A Semantic Framework for Data Analysis in Networked Systems“. In: *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11)*. Boston, MA: USENIX Association, März 2011. URL: https://www.usenix.org/conference/nsdi11/semantic-framework-data-analysis-networked-systems.

[Vos17] Nathan Voss. *afl-unicorn: Part 2 Fuzzing the 'Unfuzzable'*. https://hackernoon.com/afl-unicorn-part-2-fuzzing-the-unfuzzable-bea8de3540a5. Nov. 2017.

[Wan+19] Yunchao Wang, Zehui Wu, Qiang Wei und Qingxian Wang. „NeuFuzz: Efficient Fuzzing With Deep Neural Network“. In: *IEEE Access* 7 (2019), S. 36340–36352. DOI: 10.1109/ACCESS.2019.2903291.

[Wan+20a] Yan Wang, Peng Jia, Luping Liu, Cheng Huang und Zhonglin Liu. „A systematic review of fuzzing based on machine learning techniques“. In: *PLOS ONE* 15.8 (Aug. 2020), S. 1–37. URL: https://doi.org/10.1371/journal.pone.0237749.

[Wan+20b] Yanhao Wang, Xiangkun Jia, Yuwei Liu, Kyle Zeng, Tiffany Bao, Dinghao Wu und Purui Su. „Not All Coverage Measurements Are Equal: Fuzzing by Coverage Accounting for Input Prioritization“. In: *27th Annual Network and Distributed System Security Symposium, NDSS 2020*. San Diego, California, USA: The Internet Society, Feb. 2020, S. 17. DOI: 10.14722/ndss.2020.24422.

[Wat+16] Andrew Waterman, Yunsup Lee, David A. Patterson und Krste Asanović. *The RISC-V Instruction Set Manual, Volume I: User Level ISA, Version 2.1*. Technical Report UCB/EECS-2016-118. https://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-118.html. Electrical Engineering und Computer Sciences University of California at Berkeley, März 2016.

[Wen+19] Matthias Wenzl, Georg Merzdovnik, Johanna Ullrich und Edgar R. Weippl. „From Hack to Elaborate Technique - A Survey on Binary Rewriting“. In: *ACM Comput. Surv.* 52.3 (2019), 49:1–49:37. URL: https://doi.org/10.1145/3316415.

[Wen18] Lilian Weng. *From Autoencoder to Beta-VAE*. https://lilianweng.github.io/posts/2018-08-12-vae/. Aug. 2018.

[WHJ15] Jorge Wong-Mozqueda, Robert Haines und Caroline Jay. „Is Code Quality Related to Test Coverage?“ In: *Proceedings of the International Workshop on Sustainable Software Systems Engineering*. Jan. 2015, S. 2.

[WL90] Alexander Waibel und Kai-Fu Lee, Hrsg. *Readings in Speech Recognition*. First Edition. San Mateo, CA: Morgan Kaufmann, 1990. ISBN: 1558601244.

[Won+08] Gilbert Wondracek, Paolo Milani Comparetti, Christopher Kruegel und Engin Kirda. „Automatic Network Protocol Analysis.“ In: *NDSS*. Bd. 8. 2008, S. 1–14.

[Wri+21] Christopher Wright, William A. Moeglein, Saurabh Bagchi, Milind Kulkarni und Abraham A. Clements. „Challenges in Firmware Re-Hosting, Emulation, and Analysis“. In: *ACM Comput. Surv.* 54.1 (Jan. 2021). ISSN: 0360-0300. DOI: 10.1145/3423167. URL: https://doi.org/10.1145/3423167.

[WS92] David W. Wall und Amitabh Srivastava. *A Practical System for Intermodule Code Optimization at Link-Time*. 1992. URL: https://web.stanford.edu/class/cs343/resources/om.pdf.

[Wu22] Ziwei Wu. „A Study of Grammar-Based Fuzzing Approaches“. California Polytechnic State University, 2022. URL: https://digitalcommons.calpoly.edu/theses/2476.

[Wut24] Laurenz Wuttke. *Künstliche Neuronale Netzwerke: Definition, Einführung, Arten und Funktion*. https://datasolut.com/neuronale-netzwerke-einfuehrung/. Feb. 2024.

[Xia+24] Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel und Lingming Zhang. *Fuzz4All: Universal Fuzzing with Large Language Models*. 2024. arXiv: 2308.04748 [cs.SE]. URL: https://doi.org/10.48550/arXiv.2109.05687.

[Xu+21] Runxin Xu, Fuli Luo, Zhiyuan Zhang, Chuanqi Tan, Baobao Chang, Songfang Huang und Fei Huang. *Raise a Child in Large Language Model: Towards Effective and Generalizable Fine-tuning*. 2021. arXiv: 2109.05687 [cs.CL]. URL: https://doi.org/10.48550/arXiv.2109.05687.

[Yan+23] Chenyuan Yang, Yinlin Deng, Runyu Lu, Jiayi Yao, Jiawei Liu, Reyhaneh Jabbarvand und Lingming Zhang. *White-box Compiler Fuzzing Empowered by Large Language Models*. 2023. arXiv: 2310.15991 [cs.SE]. URL: https://doi.org/10.48550/arXiv.2310.15991.

[YG23] Anil Yemme und Shayan Srinivasa Garani. „A Scalable GPT-2 Inference Hardware Architecture on FPGA“. In: *2023 International Joint Conference on Neural Networks (IJCNN)*. IEEE. Los Alamitos, CA, USA: IEEE Computer Society, 2023, S. 1–8. URL: https://doi.org/10.1109/IJCNN54540.2023.10191067.

[Yon+23] Zheng Xin Yong, Hailey Schoelkopf, Niklas Muennighoff, Alham Fikri Aji, David Ifeoluwa Adelani, Khalid Almubarak, M Saiful Bari, Lintang Sutawika, Jungo Kasai, Ahmed Baruwa, Genta Winata, Stella Biderman, Edward Raff, Dragomir Radev und Vassilina Nikoulina. *BLOOM+1: Adding Language Support to BLOOM for Zero-Shot Prompting*. Hrsg. von Anna Rogers, Jordan Boyd-Graber und Naoaki Okazaki. Toronto, Canada, Juli 2023. URL: https://aclanthology.org/2023.acl-long.653.

[YS19] S. Bharadwaj Yadavalli und Aaron Smith. „Raising binaries to LLVM IR with MCTOLL (WIP paper)“. In: *Proceedings of the 20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems, LCTES 2019, Phoenix, AZ, USA, June 23-23, 2019*. Hrsg. von Jian-Jia

Chen und Aviral Shrivastava. ACM, 2019, S. 213–218. URL: https://doi.org/10.1145/3316482.3326354.

[Yu+17] Lantao Yu, Weinan Zhang, Jun Wang und Yong Yu. „Seqgan: Sequence generative adversarial nets with policy gradient“. In: *Thirty-first AAAI conference on artificial intelligence*. 2017.

[Yun+22] Joobeom Yun, Fayozbek Rustamov, Juhwan Kim und Youngjoo Shin. „Fuzzing of Embedded Systems: A Survey“. In: *ACM Comput. Surv.* 55.7 (Dez. 2022). ISSN: 0360-0300. DOI: 10.1145/3538644. URL: https://doi.org/10.1145/3538644.

[YZZ23] Chenyuan Yang, Zijie Zhao und Lingming Zhang. *KernelGPT: Enhanced Kernel Fuzzing via Large Language Models*. 2023. arXiv: 2401.00563 [cs.CR]. URL: https://doi.org/10.48550/arXiv.2401.00563.

[Z-W21] Z-Wave Alliance. *Z-Wave Specifications*. Z-Wave Alliance. 2021. URL: https://z-wavealliance.org/development-resources-overview/specification-for-developers/.

[Zal19] Michal Zalewski. *AFL Documentation*. Version 2.53b. https://afl-1.readthedocs.io/_/downloads/en/latest/pdf/. Juli 2019.

[Zel+24] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser und Christian Holler. *The Fuzzing Book*. CISPA Helmholtz Center for Information Security, 2024. URL: https://www.fuzzingbook.org/.

[Zha+18] Gen Zhang, Xu Zhou, Yingqi Luo, Xugang Wu und Erxue Min. „Ptfuzz: Guided fuzzing with processor trace feedback“. In: *IEEE Access* 6 (2018), S. 37302–37313.

[Zha+22] Sen Zhao, Jinfa Wang, Shouguo Yang, Yicheng Zeng, Zhihui Zhao, Hongsong Zhu und Limin Sun. „ProsegDL: Binary protocol format extraction by deep learning-based field boundary identification“. In: *2022 IEEE 30th International Conference on Network Protocols (ICNP)*. IEEE. 2022, S. 1–12.

[Zha+24a] Ao Zhang, Yiying Zhang, Yao Xu, Cong Wang und Siwei Li. „Machine Learning-Based Fuzz Testing Techniques: A Survey“. In: *IEEE Access* 12 (2024), S. 14437–14454. DOI: 10.1109/ACCESS.2023.3347652.

[Zha+24b] Hongxiang Zhang, Yuyang Rong, Yifeng He und Hao Chen. *LLAMAFUZZ: Large Language Model Enhanced Greybox Fuzzing*. 2024. arXiv: 2406.07714 [cs.CR]. URL: https://arxiv.org/abs/2406.07714.

[Zha+24c] Sen Zhao, Shouguo Yang, Zhen Wang, Yongji Liu, Hongsong Zhu und Limin Sun. „Crafting Binary Protocol Reversing via Deep Learning With Knowledge-Driven Augmentation“. In: *IEEE/ACM Transactions on Networking* 32.6 (2024), S. 5399–5414. DOI: 10.1109/TNET.2024.3468350.

[Zhe+19] Yaowen Zheng, Ali Davanian, Heng Yin, Chengyu Song, Hongsong Zhu und Limin Sun. „FIRM-AFL: high-throughput greybox fuzzing of iot firmware via augmented process emulation“. In: *28th USENIX Security Symposium (USENIX Security 19)*. 2019, S. 1099–1114.