Auswertung von Netzwerkfuzzern am Beispiel des MQTT Protokolls

Analyse der Performance von Netzwerkfuzzern unter Verwendung von Pulsar, AFLNet und boofuzz

B a c h e l o r a r b e i t

Hochschule für Angewandte Wissenschaften Hof
Fakultät Informatik
Studiengang Mobile Computing

Vorgelegt bei
Prof. Dr. Florian Adamsky
Alfons-Goppel-Platz 1
95028 Hof

Vorgelegt von
Sebastian Peschke
Mtr. Nr.: 0000000
asbd 123
456 Hof

Hof, 24. November 2024

Inhaltsverzeichnis

Abbildungsverzeichnis

Abkürzungsverzeichnis

AFL	American Fuzzy Lop
DFA	Deterministic Finite Automaton
DNS	Domain Name System
FTP	File Transfer Protocol
GPU	Graphics Processing Unit
HTTP	Hypertext Transfer Protocol
IoT	Internet of Things
IP	Internet Protocol
M2M	Machine to Machine
MQTT	Message Queueing Telemetry Transport
pcap	packet capture
QoS	Quality of Service
SMTP	Simple Mail Transfer Protocol
TCP	Transmition Control Protocol
TLS	Transport Layer Security
ZuP	Zu untersuchendes Programm

Listings

1 Einleitung

In der heutigen digital vernetzten Welt sind Netzwerke und ihre Protokolle das Rückgrat zahlreicher kritischer Anwendungen, von einfachen Webdiensten bis hin zu komplexen IoT (Internet of Things)-Systemen. Eine entscheidende Komponente der Netzwerksicherheit ist das Testen und die Absicherung dieser Protokolle gegen potenzielle Schwachstellen. Hierbei kommt die Technik des Fuzzing zum Einsatz, die durch die gezielte Überlastung eines Systems mit fehlerhaften oder unerwarteten Eingaben mögliche Sicherheitslücken aufdecken soll. Besonders relevant ist die Analyse von Netzwerkfuzzern, die speziell für die Sicherheitstests von Netzwerkprotokollen entwickelt wurden. Das MQTT-Protokoll, ein leichtgewichtiges Messaging-Protokoll für kleine Sensoren und mobile Geräte, stellt in vielen IoT- und M2M (Machine to Machine)-Anwendungen einen wichtigen Standard dar. Aufgrund seiner weitreichenden Nutzung und der oft sensiblen Natur der übertragenen Daten, ist es essenziell, die Sicherheit dieses Protokolls gründlich zu prüfen. Netzwerkfuzzer wie Pulsar, AFL (American Fuzzy Lop)Net und boofuzz bieten unterschiedliche Ansätze zur Durchführung solcher Tests. Pulsar fokussiert sich auf eine dynamische und effektive Fehlersuche durch einen Mix aus Netzwerk- und Protokoll-basiertem Fuzzing. AFLNet basiert auf dem bewährten Ansatz von AFL und passt diesen für Netzwerkprotokolle an, um eine effektive Testabdeckung zu gewährleisten. Boofuzz – eine Weiterentwicklung des etablierten Sulley-Fuzzers – bietet spezifische Funktionen für das Testen von Netzwerkdiensten und deren Protokollen. In dieser Arbeit wird die Performance dieser Netzwerkfuzzer im Kontext des MQTT-Protokolls detailliert analysiert. Ziel ist es, durch einen Vergleich der Fuzzing-Techniken und deren Effektivität Erkenntnisse zu gewinnen, die über die bloße Anwendung hinausgehen und zu einer fundierten Bewertung der getesteten Tools führen. Die Untersuchung fokussiert sich auf folgende Forschungsfragen:

Q1 Welche Performanceunterschiede gibt es unter den Netzwerkfuzzern in den Aspekten von gefundenen Bugs, Ausführgeschwindigkeit und Effizienz der generierten Eingaben?

Q2 Sind Fuzzer in der Lage künstlich platzierte Schwachstellen in Netzwerkprotokollen zu identifizieren?

Performance soll hierbei aus den Metriken der Anzahl der gefundenen Bugs, der Ausführungsgeschwindigkeit und der Effizienz der generierten Testfälle gemessen werden. Die Ergebnisse der Tests werden in Kapitel 5.4 vorgestellt und diskutiert. Hierbei wird die Forschungsfrage *Q1* beantwortet. Die Forschungsfrage *Q2* wird in Kapitel 6 beantwortet und diskutiert.

2 Hintergrund

Folgendes Kapitel beschreibt die Grundlagen der in dieser Arbeit angewendeten Testing-Methoden und Herangehensweisen. Hierfür werden zunächst die Software-Testing-Methoden vorgestellt, die im Applikations- und System-Testing verwendet werden. Anschließend wird das Fuzzing als eine der wichtigsten Methoden des Software-Testings vorgestellt und kategorisiert. Zum weiteren Verständnis werden die Grundlagen des TCP (Transmition Control Protocol)-Protokolls erläutert. Darauf aufbauend werden die Grundlagen des ZuP (Zu untersuchendes Programm)s MQTT vorgestellt und die Funktionsweise des Protokolls erläutert, um eine ausreichende Grundlage zum Verständnis der Analyse zu schaffen.

2.1 Software-Testing Methoden

Das Fuzzing ist eine Software-Testing Methode zum Finden von Bugs und Sicherheitslücken in Applikationen. Es gibt verschiedene Arten von Software-Testing Methoden, die in der Software-Entwicklung eingesetzt werden. Die drei Kategorien sind White-Box, Black-Box und Grey-Box Testing. Sie unterscheiden sich in der Art und Weise, wie sie die Software testen. Jede Methode hat ihre eigenen Vor- und Nachteile und wird in verschiedenen Situationen eingesetzt. In den folgenden Abschnitten werden die verschiedenen Software-Testing Methoden genauer erläutert.

2.1.1 White-Box testing

Es handelt sich um White-Box Testing, wenn der Tester Zugriff auf den Quellcode der Software hat. Diese Testingmethode wird auch als *Clear-Box*, *Transparent-Box* oder *Open-Box* Testing bezeichnet. Der Tester kann den Quellcode der Software analysieren und die Testfälle basierend auf dem Quellcode erstellen [1]. White-Box Testing wird normalerweise von Entwicklern durchgeführt, um sicherzustellen, dass der Code korrekt funktioniert und keine Fehler enthält. Es wird auch verwendet, um die Testabdeckung zu messen und sicherzustellen, dass alle Teile des Codes getestet wurden. White-Box Testing ist eine sehr effektive Methode, um Fehler in der Software zu finden, aber es erfordert auch eine gründliche Kenntnis des Quellcodes und der Software-Architektur. Zu White-Box Testing gehören mitunter das *Unit Testing*, bei dem einzelne Komponenten der Software getestet werden und *Path Testing* bei dem alle möglichen Programmpfade innerhalb eines zu testenden Abschnittes abgelaufen und überprüft werden sollen.
Der Vorteil dieser Testmethode besteht darin, dass hierbei komplexe und auch strukturelle

Fehler aufgedeckt werden können. Die Anwendung einer solchen Testmethode findet im optimalen Fall während der Entwicklung einer Anwendung statt.

2.1.2 Black-Box testing

Bei Black-Box Testing wird kein Zugriff auf den Quellcode und die interne Struktur der Application gebraucht. Das Ziel dieser Testmethode ist es zunächst mögliche Eingaben von Benutzern einer Software nachzuahmen [2]. Somit werden funktionalitäten einer Software getestet, ohne dass der Tester den Quellcode kennt. In der Regel werden hierbei nur bereits kompilierte Software-Module oder von außen Zugreifbare Endpunkte getestet. Anwendungsfälle einer solchen Herangehensweise entsprechen einem Penetrationstest oder eines Usability-Tests. Der Vorteil von Black-Box Testing besteht darin, dass das ZuP aus der Sicht des Answers getestet wird. Dabei wird die Software auf ihre Funktionalität und Benutzerfreundlichkeit getestet.
Black-Box Testing ist eine effektive Methode, um sicherzustellen, dass die Software den gestellten Anforderungen entspricht und keine offensichtlichen Fehler enthält. Es ist jedoch schwierig, komplexe Fehler zu finden, da der Tester keinen Zugriff auf den Quellcode hat und nicht weiß, wie die Software intern funktioniert.

2.1.3 Grey-Box testing

Grey-Box Testing kombiniert beide Aspekte der bereits genannten Ansätze. Hierbei werden Teile des ZuP offengelegt. Während beim White-Box Testing die interne Struktur oder der Code des Systems bekannt und getestet wird und beim Black-Box Testing nur die externen Funktionalitäten ohne Wissen über den internen Aufbau geprüft werden, ermöglicht Grey-Box Testing eine Balance zwischen diesen Ansätzen. Beim Grey-Box Testing hat der Tester teilweise Kenntnisse über die interne Struktur der Anwendung oder des Systems, verwendet diese Informationen jedoch hauptsächlich, um die Erstellung und Durchführung von Testszenarien zu optimieren, die auf der Benutzeroberfläche basieren [3]. Diese Methode erlaubt es, gezielte Testfälle zu entwerfen, die nicht nur auf den sichtbaren Eingaben und Ausgaben basieren, sondern auch auf dem Wissen über die zugrundeliegenden Datenflüsse, Algorithmen oder architektonischen Schwächen. Grey-Box Testing wird häufig in Szenarien eingesetzt, in denen eine vollständige Kenntnis der internen Struktur nicht notwendig oder verfügbar ist, jedoch ein gewisses Maß an Verständnis über die Architektur oder den Code notwendig ist, um tiefere Testfälle zu entwickeln. Diese Technik kann besonders nützlich sein bei der Validierung von Sicherheitsaspekten, der Integrationstests von komplexen Systemen oder der Optimierung der Testabdeckung bei großen Anwendungen. In der Praxis findet Grey-Box Testing Anwendung

in Bereichen wie Web-Applikationen, APIs und bei sicherheitskritischen Systemen, bei denen sowohl die Funktionalität als auch die interne Verarbeitung zuverlässig und sicher getestet werden müssen. Die Methode bietet somit eine ausgewogene Teststrategie, die die Vorteile von White-Box und Black-Box Testing vereint, um eine effektivere Fehlererkennung zu ermöglichen.

2.2 Fuzzing

Fuzzing ist eine Black Box Testmethode, die darauf abzielt, Fehler in Software zu finden, indem zufällige oder semi-zufällige Daten als Eingabe verwendet werden [4]. Diese Daten werden in der Regel von einem Fuzzer generiert, der die Software mit den Daten füttert und auf unerwartetes Verhalten prüft. Fuzzing ist eine effektive Methode, um Fehler in Software zu finden, die durch unerwartete Eingaben verursacht werden. Es ist eine weit verbreitete Methode, um Sicherheitslücken in Software zu finden, die von Hackern ausgenutzt werden können. Fuzzing kann auch dazu verwendet werden, um die Stabilität und Zuverlässigkeit von Software zu testen und um sicherzustellen, dass sie korrekt funktioniert.
Es gibt verschiedene Arten von Fuzzern, die unterschiedliche Strategien und Techniken verwenden, um Fehler in Software zu finden. Die Implementierung der Techniken und Strategien hängen von den Anforderungen und Zielen des Fuzzers ab. In den folgenden Abschnitten werden die verschiedenen Arten von Fuzzern, ihre Strategien und Techniken näher erläutert.

2.2.1 Arten des Fuzzing

Trotz dem, dass Fuzzing als eine Black-Box Testmethode gilt, gibt es verschiedene Arten von Fuzzing, die unterschiedliche Strategien und Techniken verwenden, um Fehler in Software zu finden. Die Arten des Fuzzing sind somit in drei Kategorien unterteilt [5]:

- Black-box Fuzzing
- White-box Fuzzing
- Grey-box Fuzzing

Jede Kategorie hat ihre eigenen Vor- und Nachteile und wird in den folgenden Abschnitten näher erläutert.
Im Black-box Fuzzing wird die Software ohne Kenntnis des ursprünglichen Quellcodes getestet. Der Fuzzer generiert zufällige oder semi-zufällige Eingaben und übergibt sie der Software. Das Generieren der Eingaben erfolgt in der Regel durch Mutation von vorhandenen Eingaben

oder durch Generierung neuer Eingaben und wird durch Regeln limitiert. Mit diesen Regeln wird sichergestellt, dass die Mehrheit der Eingaben nicht von der zu untersuchenden Software – aufgrund von bspw. Syntaxfehlern – abgelehnt wird. Bei der Durchführung des Fuzzing überwacht der Fuzzer das Verhalten der Software und prüft, ob unerwartetes Verhalten auftritt. Das Paradigma, das hierbei verfolgt wird, ist das Testen der Software ohne Kenntnis des Quellcodes, des Kontrollflusses und somit der internen Logik, um die Benutzung aus der Perspektive eines Nutzers der Software nachzustellen.
Die Vorteile dieser Herangehensweise sind, dass der Code nicht verfügbar sein muss und somit auch proprietäre Software getestet werden kann. Zudem werden Fehler gefunden, welche durch einen Nutzer der Software ausgelöst werden können. Der ausschlaggebende Nachteil dieser Vorgehensweise ist, dass die Eingaben aufgrund limitierter Informationen über die Software nur erschwert zu komplexeren Bugs führen können. Das hat zufolge, dass die gefundenen Bugs meistens einfacher Natur sind und Eingaben oftmals nicht in tiefe Verzweigungen des Codes gelangen können [6].

Im Gegensatz zum Black-box Fuzzing wird im White-box Fuzzing der Quellcode der Software benötigt. Der Fuzzer generiert Eingaben, die auf der Analyse des Quellcodes basieren. Dabei werden die Eingaben so generiert, dass sie die verschiedenen Pfade und Verzweigungen des Codes abdecken. Das Ziel ist es, die Software mit Eingaben zu füttern, die potenzielle Fehler in den verschiedenen Teilen des Codes auslösen. Durch die Kenntnis des Quellcodes kann der Fuzzer gezieltere Eingaben generieren und somit auch komplexere Fehler finden. Der Vorteil dieser Methode ist, dass sie effektiver ist als Black-box Fuzzing, da sie gezieltere Eingaben generieren kann. Sie kann auch dazu verwendet werden, um spezifische Teile des Codes zu testen und um sicherzustellen, dass sie korrekt funktionieren. Der Nachteil dieser Methode ist, dass der Quellcode verfügbar sein muss, was bei proprietärer Software ein Problem darstellen kann. Zudem kann das White-box Fuzzing aufgrund der Komplexität des Codes und der Anzahl der möglichen Pfade und Verzweigungen sehr aufwendig sein [6].

Grey-box Fuzzing ist eine Kombination aus Black-box und White-box Fuzzing. Die Verwendung eines Gray-box Fuzzers erfordert keinen Zugriff auf Sourcecode, jedoch kann dieser optional verwendet werden. Unter Verwendung des Quellcodes kann die Performance des Fuzzers verbessert werden, indem gezieltere Eingaben generiert werden. Die Vorteile des Gray-box Fuzzing sind, dass es effektiver ist als Black-box Fuzzing, da es gezieltere Eingaben generieren kann. Außerdem ermöglicht dieser Ansatz auch das Analysieren von proprietärer Software aufgrund dessen, dass der Quellcode nicht für das Fuzzen von Software notwendig ist [7]. Die Herangehensweise des Grey-box Fuzzing ist ähnlich zu dem

des Black-box Fuzzing. Jedoch können vor dem Kompilieren des Quellcodes Instruktionen des Fuzzers eingefügt werden, um das Fuzzing zu verbessern. Zudem wird unter bspw. AFL ein eigener Compiler verwendet, um Metadaten des Programms zu extrahieren und weitere Instrumentierungsanweisungen in den Code zu injizieren [8].

2.2.2 Generierung von Input

Die Generierung von Input ist ein wichtiger Bestandteil des Fuzzing-Prozesses. Hierbei unterscheiden sich die Fuzzer in ihrer Herangehensweise und den Techniken, die sie verwenden, um Eingaben zu generieren. Die Generierung von Input kann auf verschiedene Arten erfolgen, die in den folgenden Abschnitten näher erläutert werden.

Generationsbasierende [5] Fuzzer generieren Eingaben, indem sie eine Menge von Eingaben in Epochen erstellen. Jede Epoche wird als Generation bezeichnet. Diese Eingaben basieren auf einem vom Anwender festgelegten Regelwerk. Diese Herangehensweise des Generierens von Eingaben ist effektiv, um die Software mit Eingaben zu versorgen, welche nicht bereits im Vorfeld von dem ZuP als fehlerhaft befunden werden sollen. Beispielsweise ist dies Vorteilhaft, wenn besonders komplexe Syntax des Input vonnöten ist.

Mutationsbasierende [5] Fuzzer generieren Eingaben, indem sie vorhandene Eingaben, welche bereits im Vorfeld definiert wurden, verändern. Hierbei wird der Aufbau und die Struktur der Eingaben standardmäßig nicht berücksichtigt. Diese Veränderung wird Mutation genannt und kann auf verschiedene Arten erfolgen. Die Mutation kann beispielsweise durch das Hinzufügen, Entfernen oder Verändern von Bytes in der Eingabe erfolgen. Um die Güte der neu generierten Eingaben zu bestimmen, wird die Code Coverage – also die im Quellcode erreichte Tiefe – analysiert. Eingaben, die besonders viele Codesegmente erreichen werden als besonders gut gewertet und als bevorzugte Eingabe für weitere Mutationen verwendet.

2.2.3 Intelligenz von Fuzzern

Die Intelligenz verschiedener Fuzzer kann in zwei Kategorien [9] unterteilt werden:

- Brute-Force Fuzzer
- Intelligente Fuzzer

Zu den *Brute-Force* Fuzzern gehören diejenigen, die das Feedback eines Programms nicht interpretieren. Hierzu werden die Eingaben zufällig generiert und an das ZuP übergeben. Aufgrund dessen werden die Eingaben nicht auf ihre Gültigkeit oder Korrektheit überprüft.

Dumme Fuzzer sind in der Regel einfach zu implementieren und erfordern keine speziellen Kenntnisse über das ZuP. Sie sind jedoch weniger effektiv als intelligente Fuzzer [9], da sie keine Informationen über das Verhalten des Programms sammeln und somit nicht in der Lage sind, gezieltere Eingaben zu generieren. Ein Vorteil der Verwendung von dummen Fuzzern ist, dass sie in der Regel schneller sind als intelligente Fuzzer. Das ist darauf zurückzuführen, dass aufgrund der nicht verwendeten Feedback-Schleife – in der die Antworten des ZuP auf die Eingaben bewertet werden und anhand dessen eine passende Generierung der Eingaben gewählt wird – die Eingaben schneller generiert werden können.

Intelligente Fuzzer adaptieren die Generierung der Eingaben anhand des Feedbacks. Hierzu werden die Reaktion des Programms auf die Eingabe, die Anzahl der ausgeführten Instruktionen oder die Anzahl der gefundenen Bugs analysiert. Der Fokus der Analyse hängt von der Implementierung des Fuzzers ab. Intelligente Fuzzer können das Verhalten des Programms überwachen und auf unerwartete Ereignisse reagieren [10]. Sie können auch Informationen über die Struktur des Programms sammeln und diese Informationen verwenden, um gezieltere Eingaben zu generieren.
Intelligente Fuzzer sind in der Regel effektiver als dumme Fuzzer, da sie gezieltere Eingaben generieren können. Sie sind jedoch auch komplexer zu implementieren und erfordern spezielle Kenntnisse über das ZuP.

2.2.4 Strategien von Fuzzern

Fuzzer können verschiedene Strategien verfolgen, um Eingaben zu generieren und das Verhalten der Software zu überwachen. Die Strategien können in zwei Kategorien unterteilt [5] werden:

- Abdeckung von Codepfaden
- Zielgerichtetes Fuzzen

Bei dem zielgerichteten Fuzzen werden nur bestimmte Teile des Codes getestet. Der Fuzzer generiert Eingaben, die auf bestimmten Kriterien basieren, um gezielt Fehler in diesen Teilen des Codes zu finden. Zu den Kriterien gehören beispielsweise die Anzahl der ausgeführten Instruktionen, die Anzahl der gefundenen Bugs oder die Reaktion des Programms auf die Eingabe. Bei dieser Technik handelt es sich in der Regel um einen White-Box Fuzzer. Um nur auf bestimmte Teile eines Programms zu fokussieren, wird der Quellcode benötigt [11].

Bei der Strategie der Codepfadabdeckung wird versucht, möglichst viele Pfade und Verzweigungen des Codes abzudecken. Der Fuzzer generiert Eingaben, die verschiedene Pfade und Verzweigungen des Codes abdecken, um potenzielle Fehler in verschiedenen Teilen des Codes zu finden. Diese Technik wird in der Regel von Black- und Grey-Box Fuzzern verwendet, da sie keine Kenntnis des Quellcodes benötigen und sich somit anderer Informationen bedienen müssen. Die Codepfadabdeckung ist eine effektive Methode, um sicherzustellen, dass große Teile des Codes getestet wurden [5]. Somit ist es ebenso möglich besonders komplexe Bugs und Schwachstellen in einem System zu finden, da die Eingaben besonders tief in den Programmcode gelangen.

2.3 Grundlagen der TCP/IP-Kommunikation

Die TCP/IP (Internet Protocol)-Kommunikation ist ein Protokoll-Stack, der für die Kommunikation zwischen Computern verwendet wird. Der Protokoll-Stack besteht aus zwei Schichten, dem TCP- und dem IP-Protokoll [12]. Das IP-Protokoll ist für die Adressierung und das Routing von Datenpaketen zuständig. Das TCP-Protokoll ist für die zuverlässige Übertragung von Datenpaketen zuständig. Die Kommunikation zwischen zwei Computern erfolgt über eine Verbindung, die durch eine IP-Adresse und einen Port identifiziert wird. Die IP-Adresse identifiziert den Computer und der Port identifiziert den Dienst, der auf dem Computer ausgeführt wird. Die Kommunikation erfolgt über eine Reihe von Datenpaketen, die zwischen den beiden Computern ausgetauscht werden. Die Datenpakete enthalten Informationen über den Absender, den Empfänger, die Daten selbst und eine Prüfsumme, die zur Überprüfung der Datenintegrität verwendet wird. Die gesamte Kommunikation kann in drei Schritten: *Verbindungsaufbau*, *Datenübertragung* und *Verbindungsabbau* eingeteilt werden. Der Verbindungsaufbau erfolgt in der Regel über eine Reihe von Schritten, die als *Handshake* bezeichnet werden [13]. Dieser Handshake besteht aus den drei Schritten *SYN*, *SYN-ACK* und *ACK*. Im Rahmen des Verbindungsaufbaus erfolgt die Verbindung sowie die Herstellung der Verbindung zwischen den beiden Computern. Während der Datenübertragung erfolgt ein Austausch der Daten zwischen zwei Kommunikationspartnern. Der Verbindungsabbau impliziert die Trennung der Verbindung. Die Implementierung der TCP/IP-Kommunikation erfolgt in der Regel über die Socket-Programmierung. Bei der Socket-Programmierung unter UNIX-Systemen hierzu die bereits im System implementierte Socket-API verwendet. Die Socket-API stellt eine Reihe von Funktionen zur Verfügung, die es einem Programm ermöglichen, mit dem Netzwerk zu kommunizieren. Sockets werden in der Regel als Datei-Handle dargestellt und können vom Kernel gelesen und geschrieben werden. Diese Dateien sind in der Regel nicht im Namespace des Dateisystems abgelegt. Metadaten der Sockets werden

jedoch im Dateisystem abgelegt und können zu Analysezwecken von weiteren Programmen verwendet werden. Diese Metadaten können im Dateisystem unter */proc/net/tcp*, */proc/net/tcp6* und unter im Verzeichnis */proc/sys/net* eingesehen werden [14].
Der Gesamte Stack kann in vier Schichten unterteilt werden:

- Anwendungsschicht
- Transportschicht
- Internetschicht
- Netzwerk-Interface-Schicht

Die Anwendungsschicht ist die oberste Schicht des Protokoll-Stacks und enthält die Anwendungen, die die Kommunikation zwischen den Computern ermöglichen. Zu dieser Schicht gehören die von einer Anwendung verwendeten Protokolle, wie z.B. HTTP (Hypertext Transfer Protocol), FTP (File Transfer Protocol), DNS (Domain Name System) oder SMTP (Simple Mail Transfer Protocol), mithilfe derer Daten über ein Netzwerk übertragen werden. Die Funktion der zweiten Schicht des Protokoll-Stacks, der sogenannten Transportschicht, besteht in der Übertragung von Datenpaketen. In dieser Schicht obliegt die zuverlässige Übertragung von Datenpaketen dem TCP-Protokoll. Die Internetschicht stellt die dritte Schicht des Protokoll-Stacks dar und ist für die Adressierung sowie das Routing von Datenpaketen zuständig. In dieser Schicht obliegt dem IP-Protokoll die Adressierung und das Routing von Datenpaketen. Die Netzwerkschicht stellt die unterste Schicht des Protokoll-Stacks dar und ist für die Verbindung mit einem Netzwerk sowie der dazugehörigen Hardware zuständig.
In dieser Arbeit wird die Anwendungsschicht des Protokoll-Stacks analysiert.

2.4 Grundlagen des Mosquitto MQTT-Brokers

Mosquitto ist ein Open-Source-Message-Broker, der das MQTT-Protokoll implementiert. MQTT ist ein Publish/Subscribe-Protokoll, das für Netzwerke mit geringer Bandbreite, hoher Latenz oder unzuverlässigen Verbindungen entwickelt wurde. Es wird häufig in der IoT-Kommunikation verwendet, um Daten zwischen Geräten, Sensoren und Anwendungen auszutauschen. MQTT, und damit auch Mosquitto, baut auf dem TCP/IP-Stack auf, was eine zuverlässige Datenübertragung zwischen Client und Server gewährleistet. TCP sorgt dabei für eine gesicherte, verbindungsorientierte Kommunikation, bei der Datenpakete in der richtigen Reihenfolge und fehlerfrei übertragen werden. Diese Basis auf dem TCP/IP-Stack macht MQTT zu einem robusten und weit verbreiteten Protokoll für den Einsatz in vernetzten Umgebungen, insbesondere im IoT-Bereich. Mosquitto bietet eine Möglichkeit,

MQTT-Nachrichten zwischen verschiedenen Clients zu vermitteln. Der MQTT-Broker ermöglicht es Clients, Nachrichten an bestimmte Themen (Topics) zu senden (Publish) und/oder zu empfangen (Subscribe). Durch seine geringe Overhead-Belastung und die Unterstützung für QoS (Quality of Service)-Ebenen, stellt Mosquitto eine zuverlässige Lösung für die Kommunikation in verteilten Systemen dar. Es bietet auch erweiterte Funktionen wie Authentifizierung, Verschlüsselung (TLS (Transport Layer Security)) und Integration mit anderen Diensten.
Das Protokoll MQTT definiert drei Arten von Nachrichten [15]:

- **Publish:** Ein Client sendet eine Nachricht an einen Broker, die dieser an alle Abonnenten weiterleitet.
- **Subscribe:** Ein Client abonniert ein bestimmtes Thema (Topic), um Nachrichten zu empfangen, die von anderen Clients zu diesem Thema gesendet werden.
- **Unsubscribe:** Ein Client beendet das Abonnement eines Themas und empfängt keine weiteren Nachrichten zu diesem Thema.

Der erste Schritt in diesem Zustand ist das Starten des MQTT-Brokers. Der Broker dient als Vermittler für die Nachrichten, die zwischen verschiedenen MQTT-Clients ausgetauscht werden. Dieser Schritt ist entscheidend für den weiteren Betrieb des Systems. Nach dem Starten des MQTT-Brokers geht das System in den Initialisierungszustand über. Hier wird die Netzwerkschnittstelle initialisiert, die für die Kommunikation über MQTT erforderlich ist. Diese Initialisierung stellt sicher, dass das System in der Lage ist, sich mit anderen Geräten im Netzwerk zu verbinden und Nachrichten zu senden oder zu empfangen. Nach erfolgreicher Initialisierung wechselt das System in den Zustand *Listening*. In diesem Zustand wartet das System entweder darauf, dass es eine MQTT-Nachricht empfängt oder es ist in einem Leerlaufzustand (Idle). Dieser Zustand ist zentral für das System, da hier die Hauptoperationen entweder initiiert oder beendet werden. Wenn eine MQTT-Nachricht empfangen wird, wechselt das System in den Verarbeitungszustand. In diesem Zustand werden die empfangenen Nachrichten verarbeitet, und es kann entweder eine Nachricht veröffentlicht oder das System kann sich für ein neues Thema (Topic) abonnieren [16]. Sollte die Verarbeitung eine Veröffentlichung erfordern, geht das System in den Zustand *Publishing* über. In diesem Zustand wird die verarbeitete Nachricht an den entsprechenden MQTT-Topic veröffentlicht, sodass andere abonnierte Clients diese empfangen können. Alternativ kann das System in den Zustand *Subscribing* wechseln, wenn die Verarbeitung das Abonnieren eines neuen Topics erfordert. Dieser Zustand ermöglicht es dem System, Nachrichten von einem neuen Thema zu empfangen. Wenn keine Nachricht zur Verarbeitung vorliegt, bleibt das System im Leerlauf. Dieser Zustand wird auch erreicht, wenn alle anstehenden Aufgaben erledigt sind und das

System auf neue Nachrichten oder Befehle wartet. Sobald ein Herunterfahren (Shutdown) angefordert wird, wechselt das System in den Zustand Terminating. In diesem Zustand werden alle notwendigen Ressourcen aufgeräumt, um das System ordnungsgemäß zu beenden. Der endgültige Zustand des Systems ist *Stopped*. Nachdem alle Ressourcen bereinigt wurden, ist das System inaktiv und bereit, vollständig heruntergefahren zu werden.
Diese Nachrichten werden über das TCP-Protokoll zwischen Client und Broker ausgetauscht. Der Broker verfügt über eine interne Datenstruktur, die die Themen und die zugehörigen Abonnenten speichert. Diese Speicherung und das Weiterreichen dieser N achrichten kann in einer State Machine abgebildet werden.

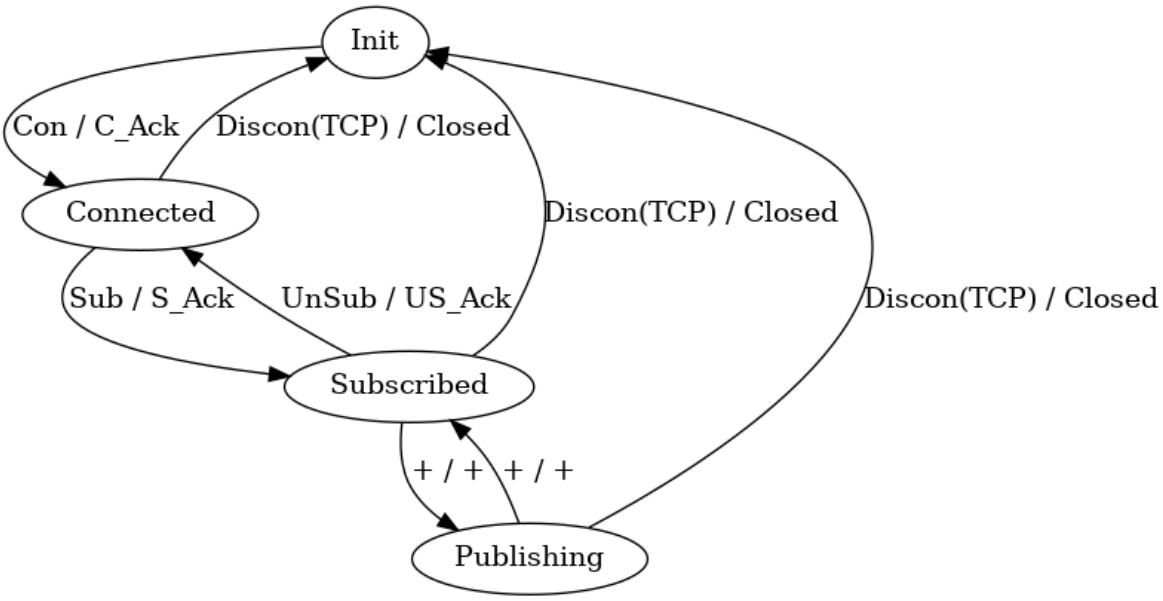

Abbildung 1: State Machine des Mosquitto MQTT-Protokolls. Sie beschreibt alle Zustände, die der Broker nach Empfangen von Nachrichten durchläuft. Beginnend mit der Überprüfung der Nachricht, über das Weiterleiten der Nachricht an die Abonnenten, bis hin zum Senden der Nachricht an die Clients.

Die Abbildung 1 zeigt eine Zustandsmaschine (State Machine) für das MQTT-Protokoll und beschreibt den Übergang zwischen verschiedenen Verbindungszuständen eines MQTT-Clients. Diese Zustandsmaschine veranschaulicht, wie der MQTT-Client in verschiedenen Situationen reagiert, z.B. beim Verbinden, Abonnieren, Veröffentlichen und Trennen der Verbindung.
Der Ausgangszustand der MQTT-Zustandsmaschine ist der Init-Zustand. Hier befindet sich der MQTT-Client, bevor eine Verbindung zum Broker hergestellt wird oder nach einer Trennung. Es ist der Ausgangspunkt und auch der Zustand, zu dem der Client nach einer Trennung zurückkehrt. Wenn der Client eine Verbindung zum Broker herstellt (Con), wird ein Verbindungsbestätigungsnachricht (C_Ack) empfangen und der Zustand wechselt zu Connected.

Bei einer Trennung der TCP-Verbindung (Discon(TCP)) wechselt der Zustand zu Closed. In diesem Zustand ist der Client erfolgreich mit dem MQTT-Broker verbunden.
Der Client kann jetzt Abonnement- und Veröffentlichungsaktionen durchführen. Der Client sendet eine Abonnementanfrage (Sub) an den Broker und erhält eine Bestätigungsnachricht (S_Ack). Der Zustand wechselt dann zu Subscribed. Bei einer Trennung der TCP-Verbindung (Discon(TCP)) wechselt der Zustand zu Closed. In diesem Zustand hat der Client erfolgreich ein oder mehrere Themen (Topics) abonniert. Der Client ist nun in der Lage, Nachrichten von den abonnierten Themen zu empfangen. Der Client kann eine Nachricht veröffentlichen.
Nach jedem erfolgreichen Veröffentlichungsprozess bleibt der Client im Subscribed-Zustand oder wechselt für die Veröffentlichung kurzzeitig in den Publishing-Zustand. Der Client kann eine Abmeldung (UnSub) von einem Thema anfordern und erhält eine Abmeldebestätigungsnachricht (US_Ack). Der Zustand kehrt dann zu Connected zurück. Bei einer Trennung der TCP-Verbindung (Discon(TCP)) wechselt der Zustand zu Closed. In diesem Zustand veröffentlicht der Client Nachrichten an die abonnierten Themen.
Es ist ein temporärer Zustand, in dem die Veröffentlichung von Nachrichten erfolgt. Nach der Veröffentlichung einer Nachricht kehrt der Client in den Subscribed-Zustand zurück, um weitere Nachrichten empfangen zu können. Bei einer Trennung der TCP-Verbindung (Discon(TCP)) wechselt der Zustand zu Closed.
Dies ist ein Endzustand, der erreicht wird, wenn die Verbindung zwischen dem Client und dem Broker getrennt wird. Der Client ist nun nicht mehr in der Lage, Nachrichten zu senden oder zu empfangen. Wenn der Client die Verbindung zum Broker wiederherstellt, kehrt der Zustand zu Init zurück, von wo aus er den Verbindungsprozess neu starten kann.

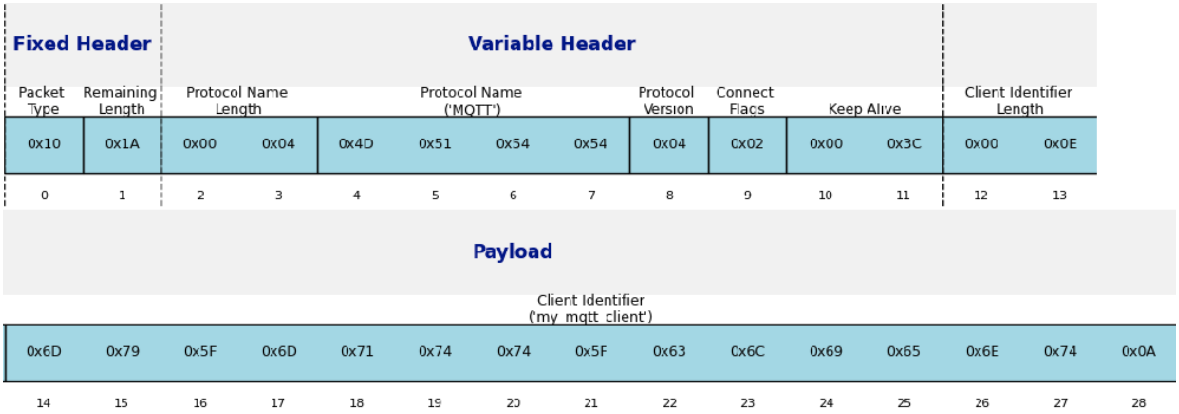

Abbildung 2: Format einer MQTT-Nachricht am beispiel eines Connect-Pakets.

Die Struktur der zu sendenden Nachrichten ist von der verwendeten MQTT-Version der Clients abhängig. Die MQTT-Version 3.1.1 – die in dieser Arbeit verwendet wurde – definiert ein festes Header-Format, das aus einem Steuerbyte und einer variablen Länge besteht. Als

Nächstes folgt ein Variabler Header, der weitere Information über die Länge des Protokollnamens und den verwendeten Protokollnamen, sowie die verwendete Protokollversion und ein Connect-Flag-Byte enthält. das zehnte und elfte Byte enthalten die Keep-Alive-Zeit, die die maximale Zeit in Sekunden angibt, die der Client auf eine Antwort des Brokers warten soll. Die restlichen Bytes enthalten die Client-ID, die Länge des Benutzernamens und den Benutzernamen selbst, die Länge des Passworts und das Passwort selbst. Die MQTT-Nachrichtenstruktur [15] ist in Abbildung 2 dargestellt.
Die gesendeten Nachrichten haben im kürzesten Fall eine Länge von 2 Bytes, wobei das Steuerbyte für einen Verbindungsabbau gesendet wird und somit nur aus dem Fixed-Header besteht. Die Obergrenze der Größe eines validen Nachrichtenpakets beträgt 256 Megabytes [15].

2.5 Verwendete Fuzzer

In diesem Kapitel werden die in dieser Arbeit verwendeten Fuzzer vorgestellt. Die Fuzzer AFLNet, boofuzz und Pulsar werden im Kontext der Performance-Analyse von Netzwerkfuzzern verwendet und auf ihre Effektivität bei der Identifizierung von Schwachstellen in Netzwerkprotokollen untersucht. Die Fuzzer boofuzz und AFLNet wurden in dieser Arbeit ausgewählt, da sie weit verbreitete Fuzzer mit umfangreicher Dokumentation sind. Pulsar wurde aufgrund seiner einzigartigen Fähigkeit, Netzwerkprotokolle zu erlernen und zu simulieren, ausgewählt. Die Auswahl der drei Fuzzer ermöglicht es, verschiedene Ansätze und Techniken des Netzwerkfuzzings zu vergleichen und einen möglichst umfassenden Einblick in die Herangehensweisen der Fuzzer zu erhalten.

2.5.1 AFLNet

AFL ist ein Fuzzer, der auf dem Prinzip der Codeabdeckung basiert. Er zählt somit zu der Familie der Grey-Box Fuzzer und basiert auf einer mutationsbasierten Fuzzing-Technik. Bei AFL handelt es sich ebenso um einen Applikationsfuzzer, welcher mithilfe eines Eingabestreams Daten an ein ZuP sendet und somit die Software auf Schwachstellen testet.
Zur Analyse des Programms wird in dieser Arbeit jedoch ein Derivat von AFL(AFLNet) verwendet. Es unterscheidet sich von dem ursprünglichen AFL dadurch, dass es speziell für die Analyse von Netzwerkprotokollen entwickelt wurde. Die Kernfunktionalität von AFL bleibt jedoch erhalten und wird um die Möglichkeit erweitert, Netzwerkprotokolle zu analysieren. Hierzu müssen die Netzwerkpakete, die an das ZuP gesendet werden, in einem speziellen Format vorliegen. Die Extraktion der Netzwerkpakete erfolgt mithilfe eines pcap (packet capture)-Dump-Files. Diese Dump-Files enthalten den Netzwerkverkehr, der zwischen zwei

Kommunikationspartnern stattgefunden hat. Um diese Dump-Files zu generieren und somit den Netzwerkverkehr aufzunehmen wird das Tool `tcpdump` verwendet. Zur Generierung der Netzwerkpakete wird das pcap-Dump-File in ein AFL-kompatibles Format umgewandelt, indem die rohen Datenflüsse extrahiert und in eine Datei geschrieben werden. Die Extraktion der Datenflüsse kann mit Netzwerkanalyse-Tools wie `Wireshark` oder `tcpdump` erfolgen. Hierzu wird im Repository von AFLNet [17] bereits eine Herangehensweise vorgestellt, wie die Datenflüsse mit Wireshark extrahiert werden können. Diese Datenflüsse werden dann als Bytestream von AFL als Eingabe verwendet, um das ZuP zu fuzzen.

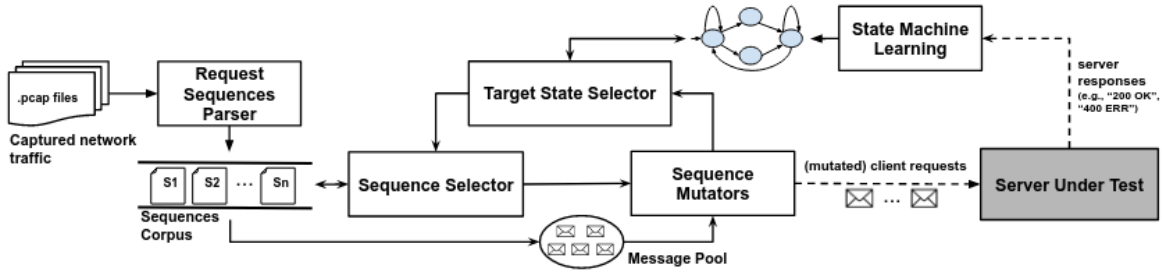

Abbildung 3: Die Abbildung zeigt die Architektur von AFLNet nach Pham Van-Thuan [18].

Zuerst muss das ZuP gestartet und mir Eingaben gefüttert werden. Während der Transaktion der Nachrichten wird der einhergehende Traffic mit einem Aufnahmeprogramm wie `tcpdump` aufgezeichnet. Die aufgezeichneten Daten werden in ein pcap-Dump-File geschrieben und müssen anschließend analysiert werden. Dabei muss der Netzwerkverkehr in ein Format umgewandelt werden, das von AFLNet verstanden wird. Dieses Format besteht in dem Fall dieser Arbeit aus einem Bytestream, der die Nachrichten des Netzwerkverkehrs enthält. Die Nachrichten werden in einer Datei abgelegt und können dann von AFLNet als Eingabe verwendet werden. Besonderes Augenmerk hierbei liegt auf der Analyse des Netzwerkverkehrs und der daraus resultierenden State Machine. AFLNet analysiert den Netzwerkverkehr und extrahiert die Nachrichten, die zwischen Client und Server ausgetauscht werden. Werden bei der Ausführung von Nachrichten bestimmte Zustände erreicht, so werden diese Zustände in einer State Machine abgebildet und die State Machine mit dem neuen Zustand als Node erweitert. Die State Machine wird mithilfe von AFLNet erlernt und ermöglicht es, den Zustand des ZuP nachzuverfolgen. Das Erlernen der State Machine erfolgt durch das Aufstellen eines Graphen, der die Zustände des ZuP abbildet. Hierbei werden die Zustände basierend auf den gesendeten Nachrichten und den vom ZuP generierten Antworten identifiziert. Dieser Graph wird initial bei der Testphase der Nachrichten erstellt und bei jeder neuen Nachricht erweitert. Die Erweiterung des Graphen erfolgt durch das Hinzufügen von neuen Zuständen, die durch die Nachrichten erreicht werden [18].

Anschließend werden die Nachrichten mit den enthaltenen Nachrichtensequenzen an einen in AFLNet implementierten Sequence-Mutator übergeben. Dieser Mutator generiert aus den gesendeten Nachrichten neue mutierte Nachrichten, die an das ZuP übergeben werden können.

2.5.2 Boofuzz

Boofuzz ist ein Open-Source-Fuzzing-Framework, das zur automatisierten Sicherheitsprüfung von Software verwendet wird. Es wurde als Fork des populären Sulley Fuzzing Frameworks entwickelt und bietet verbesserte Stabilität, erweiterte Funktionen und aktive Wartung. Boofuzz ist in Python geschrieben und ermöglicht das Erstellen von Fuzzing-Kampagnen durch die Definition von Testfällen, die systematisch verschiedene Protokollfelder und Eingabeparameter abdecken. Das Framework unterstützt sowohl Netzwerkprotokolle als auch dateibasierte Anwendungen, was es vielseitig einsetzbar macht. Dabei kann es beispielsweise Protokolle wie HTTP, FTP oder Telnet fuzzen, um Schwachstellen in Netzwerkdiensten zu identifizieren. Ein zentrales Merkmal von boofuzz ist seine Fähigkeit zur Überwachung des Zustands des Zielsystems während des Fuzzings. Hierbei handelt es sich jedoch nicht um *state-awareness* – also den tatsächlich erreichten Zustand eines Programms – sondern ob ein Dienst abgestürzt ist, und entsprechend reagieren, indem es den Testprozess anpasst oder erneut startet. Da keine Informationen über den internen Zustand des Zielsystems benötigt werden und die Generierung der Eingaben willkürlicher Natur ist, handelt es sich bei boofuzz um einen *"Brute Force"* Black-Box-Fuzzer.

2.5.3 Pulsar

Pulsar ist ein Fuzzer und wird zu der Familie der Black-Box-Fuzzer gezählt. Der Fuzzer wird dafür verwendet, um Internet Protokolle zu testen. Er ermöglicht es, ohne jedweden Quellcode anhand von gesammelten Traffic-Dumps eine State Machine eines ZuP zu erlernen. Das Besondere an diesem Fuzzer ist, dass er anhand von gesammelten Daten ein Model trainiert, welches es dem Untersucher des Programmes ermöglicht, das Programm zu simulieren und schlussendlich gezielt zu fuzzen. Bei der Simulation des Protokolls kann der Fuzzer sowohl die Rolle des Clients, als auch die Roller des Servers einnehmen. Gerade dieser Punkt ermöglicht es dem Untersucher des Programms große Flexibilität bei der Untersuchung zu erreichen. Somit kann ein großer Teil des Programms untersucht werden und möglichst tief in die Struktur des ZuP eingegriffen werden.
Das Training des Modells beruht auf der Analyse des eingefangenen Netzwerkverkehrs. Dabei werden verschiedene Nachrichten von sowohl Client- als auch Serverseite auf Byte-Ebene untersucht und ähnliche Strukturen extrahiert. Die daraus extrahierten Nachrichtensequenzen

werden darauffolgend in einen endlich-deterministischen Vektor abgebildet. Die Zusammenkunft aus mehreren Vektoren ergibt ein Cluster. Anhand der entstandenen Cluster wird eine approximierte Abbildung der State-Machine des Protokolls geschlussfolgert. Bei der Erschließung der State Machine können die tatsächlichen Zustände nur teilweise erschlossen werden und ist somit von der Güte des gesammelten Netzwerkverkehrs abhängig. Der Netzwerkverkehr wird bei der Untersuchung von Pulsar annotiert, um die Nachrichten von Client und Server unterscheiden zu können. Zusammenhänge zwischen den Nachrichten werden anhand von bereits beobachteten Nachrichten erschlossen. Diese werden in Tupeln miteinander verglichen und miteinander assoziiert. Nachdem die Assoziation aller Mitteilungen abgeschlossen ist, besteht ein Markov Modell zweiter Ordnung. Dieses Modell wird in einem Graphen abgebildet und kann somit als eine Annäherung der State Machine interpretiert werden. Das Markov Modell wird im anschluss mit einem DFA (Deterministic Finite Automaton) minimiert. Der dabei entstandene DFA erlaubt es Analysten die Zustände manuell untersuchen zu können und gegebenenfalls die Zustände zu verfeinern und nachzuvollziehen [19].

3 Verwandte Arbeiten

Die Evaluierung und Analyse der Performance von Fuzzern ist ein Forschungsgebiet in der Cybersicherheitsforschung. In der vorliegenden Sektion erfolgt eine Darstellung relevanter wissenschaftlicher Arbeiten und Methodiken zur Performance-Analyse von Fuzzern sowie eine Erörterung moderner Ansätze zum Fuzzing und zur Analyse von Netzwerkprotokollen.

Die Forschung hat sich in zwei Hauptrichtungen entwickelt: Zum einen wird Fuzzing auf eine Vielzahl von Zielsystemen angewendet, von großen verteilten Anwendungen bis hin zu eingebetteten Geräten [5, 20–23]. Zum anderen wurden die internen Abläufe von Fuzzern kontinuierlich verbessert, um eine effizientere Entdeckung von Schwachstellen zu ermöglichen [24–27]. In den letzten sechs Jahren wurden über 280 wissenschaftliche Arbeiten zu Fuzzing veröffentlicht, die zahlreiche Verbesserungen und neue Techniken vorschlagen [24]. Darüber hinaus wird in der Forschung die Anwendung von Mutationsanalysen zur Bewertung von Fuzzern untersucht [27]. Diese Technik ermöglicht es, Fuzzer anhand einer Vielzahl von unvoreingenommenen Mutationen zu vergleichen, um deren Fähigkeit zur Fehlererkennung zu bewerten. Studien zeigen, dass heutige Fuzzer nur einen kleinen Prozentsatz der Mutationen erkennen können, was eine Herausforderung für zukünftige Forschungsarbeiten darstellt [27]. Ein innovativer Ansatz in diesem Bereich ist das „Smart Fuzzing“, das maschinelles Lernen nutzt, um Test-Suiten für Netzwerkangriffe zu konstruieren [10, 19, 28]. Dieser Ansatz hat sich als effektiv erwiesen, um verschiedene unsichere Zustände in Cyber-Physical Systems zu identifizieren, und zeigt das Potenzial, neue Angriffe zu entdecken, die in herkömmlichen Benchmarks nicht erfasst werden.
Ein weiterer Aspekt ist das Reverse Engineering von Netzwerkprotokollen, das als automatisierter Prozess zur Extraktion von Protokollformat, Syntax und Semantik definiert ist [28]. Dieser Prozess erfolgt durch die Überwachung und Analyse der Eingaben und Ausgaben der Protokollsoftware, ohne auf die Protokollspezifikation angewiesen zu sein [25]. Dies ermöglicht eine tiefere Einsicht in die Funktionsweise von Protokollen, insbesondere bei nicht dokumentierten oder proprietären Protokollen.
Benchmarking und Auswertung von Fuzzern sind entscheidende Aspekte der Fuzzing-Forschung, um die Effektivität und Effizienz verschiedener Fuzzing-Methoden zu vergleichen. Bekannte Benchmarking-Tools sind FuzzBench [29], das von Google entwickelt wurde und eine standardisierte Umgebung für die Bewertung von Fuzzern bietet oder LAVA [30]. Es ermöglicht die Evaluierung von Fuzzern anhand verschiedener Zielprogramme und bietet wichtige Erkenntnisse über die Leistung von Fuzzern, insbesondere in Bezug auf die Anzahl der verwendeten Anfangsseed-Daten und die Verwendung eines gesättigten Korpus [24].

4 Methodik

In diesem Kapitel wird die Methodik zur Analyse der Performance von Netzwerkfuzzern vorgestellt. Die Methodik umfasst die Auswahl der Fuzzer, die Definition der Testumgebung, die Auswahl der Testfälle und die Durchführung der Tests.

Um möglichst viele potenzielle Schwachstellen, die das ZuP aufweisen könnte, zu identifizieren, wurde das ZuP mit einem Address Sanitizer am Vorbild von Holz [27] kompiliert. Der Address Sanitizer ist ein Werkzeug, das von der llvm-Toolchain bereitgestellt wird und zur Identifizierung von Speicherfehlern in C- und C++-Programmen verwendet wird. Der Address Sanitizer überwacht den Speicherzugriff und identifiziert potenzielle Speicherfehler wie Buffer Overflows, Memory Leaks, Use-after-free, Use-after-return, Use-after-scope und Double-free Bugs erkennt und das Programm kontrolliert mit einem Stoppsignal zum Absturz führt [31].
Die Tests wurden auf einem Rechner mit einem Intel Core i9-9900K Prozessor und 64 GB RAM durchgeführt. Das Betriebssystem des Rechners ist zur Zeit der Tests Parrot Security 6.2 (lorikeet), welches auf Debian basiert. Die Tests wurden bis auf Tests mit Pulsar auf der Host Maschine durchgeführt. Für Pulsar wurde eine virtuelle Umgebung mit Docker [32] eingerichtet.

Alle Kampagnen wurden mit einer Laufzeit von 48 Stunden durchgeführt. Zur Analyse der Performance der Fuzzer werden verschiedene Metriken verwendet. Die Metriken umfassen die Anzahl der gefundenen potenziellen Schwachstellen, die Ausführungsgeschwindigkeit und die Effizienz der generierten Testfälle anhand der erfolgreichen Verbindungen mit dem MQTT-Broker. Für das Loggen der Metriken wurde ein Skript entwickelt, das die Ausgabe des ZuP von stdout und stderr in eine Datei schreibt. Anhand dieser Dateien können die aufgetretenen Fehler mit deren Stacktrace des Address Sanitizers und die Anzahl der erfolgreichen Verbindungen mit dem MQTT-Broker ermittelt werden. Da das jedoch nicht unter AFLNet möglich ist, wurde hierzu eine preloaded library entwickelt, die die Ausgabe des Address Sanitizers und des stdin-Streams in eine Datei umleitet. Die Metriken werden in Diagrammen visualisiert und miteinander verglichen, um die Performance der Fuzzer zu bewerten. Das Ziel der Analyse ist es, die Effektivität der Fuzzer bei der Identifizierung von Schwachstellen in Netzwerkprotokollen zu bewerten und die Unterschiede in der Performance der Fuzzer zu identifizieren. Die Ergebnisse der Tests werden in Kapitel 5.4 vorgestellt und diskutiert. Hierbei wird die Forschungsfrage *Q1* 1 aus Kapitel 1 beantwortet.

5 Analyse der Fuzzer Performance

In diesem Kapitel wird die Performance von AFLNet, Pulsar und boofuzz analysiert. Der Begriff Performance ist jedoch sehr vielseitig zu interpretieren und wird in dieser Arbeit unter den Gesichtspunkten Effektivität, Geschwindigkeit und Reproduzierbarkeit bewertet.
Die Effektivität eines Fuzzers wird anhand der Anzahl der gefundenen Bugs gemessen. Die Geschwindigkeit eines Fuzzers wird anhand der Anzahl der generierten Eingaben pro Sekunde gemessen. Die Reproduzierbarkeit eines Fuzzers wird anhand der Anzahl der reproduzierten Bugs gemessen. Zur Analyse der Performance werden die Fuzzer AFLNet, Pulsar und boofuzz verwendet.

Zur Analyse der Effektivität wurden drei Schwachstellen im MQTT Protokoll implementiert. Die Schwachstellen belaufen sich auf drei Buffer Overflows und Seiteneffekte von der Aufhebung von Sanity-Checks beim Empfangen einer Publish-Nachricht. Diese implementierten Schwachstellen wurden an verschiedenen Stellen im MQTT Broker Mosquitto eingefügt, sodass auch eine Bewertung der Komplexität der gefundenen Schwachstelle erfolgen kann. Die einfachste Schwachstelle ist ein Buffer Overflow in der Funktion `mqtt_handle_connect` (Listing 2) und wird genau dann getriggert, wenn ein Client eine Verbindung zum Broker aufbaut und der Client im Payload über eine `will`-Nachricht verfügt. Die mittlere Schwachstelle ist ein Integer Overflow in der Funktion `mqtt_handle_publish` (Listing 3) und wird getriggert, wenn ein Client eine Nachricht an den Broker sendet und die Nachricht eine bestimmte Länge überschreitet. Diese Schwachstelle kann nur getriggert werden, wenn ein Client `C1` ein Topic mit einer Nachricht veröffentlicht hat und ein zweiter Client `C2` dieses Topic abonniert hat. Nur in diesem Fall wird auch die von `C1` veröffentlichte Nachricht an `C2` weitergeleitet und auch vom Broker verarbeitet. Die komplexeste Schwachstelle ist ein Memory Leak in der Funktion `mqtt_handle_subscribe` (Listing 4) und wird getriggert, wenn ein Client ein Topic abonniert und der Broker die Abonnement-Nachricht verarbeitet.

5.1 Fuzzing mit AFLNet

Das folgende Kapitel beschreibt das Aufsetzen der Testumgebung und die darin durchgeführten Experimente mit dem Fuzzing-Tool AFLNet.

5.1.1 Setup

Da AFLNet legacy Software ist, wird es nicht mehr aktiv weiterentwickelt und an neue Frameworks und Bibliotheken angepasst. Zum Kompilieren von AFLNet wird hierzu ein

angepasstes Dockerfile [33] verwendet. Dieses angepasste Dockerfile enthält alle notwendigen Abhängigkeiten, um AFLNet zu kompilieren. Hierzu zählen die Abhängigkeiten `llvm` und `clang` in der Version 11. `llvm` wird benötigt, um den AFL Compiler `afl-clang-fast` zu kompilieren. Dieser wird für optimierte Instrumentierung des Codes des ZuP verwendet und bezweckt somit höhere Ausführgeschwindigkeiten während der Fuzzing Kampagne. Für die erfolgreiche Verwendung des Compilers `afl-clang-fast` musste für die Kompatibilität mit einer neueren Ubuntu-Version ein Patch [34] für das Kompilieren des Compilers geschrieben werden.
Das ZuP Mosquitto wird ebenfalls in einem Docker Container kompiliert. Hierzu wird der Compiler `afl-clang-fast` verwendet. Die Instrumentierung des Codes erfolgt durch das Flag `-fsanitize=address`. Dieses Flag aktiviert *address sanitization* und ermöglicht es, dass AFLNet die Speicherzugriffe des ZuP überwachen kann und somit Buffer Overflows und Memory Leaks erkennt. Zum Starten des Mosquitto-Bianrys wurde außerdem eine *preloaded-library* [35] verwendet, die alle Ausgaben des ZuP aus den Kanälen `stdout` und `stderr` in eine log-Datei `mqtt-aflnet_stdout.log` weiterleitet. Diese Log-Datei wird benötigt, um die Ausgaben des ZuP zu analysieren und somit die Effektivität der generierten Testcases zu bewerten.
Die Testcases wurden mithilfe eines Python-Skripts [36] generiert. Die Struktur wurde hierzu aus bereits existierenden Nachrichten einer pcap-Datei mithilfe von Wireshark extrahiert. Anhand der Struktur der Bytes wurden die Nachrichten in ein Format umgewandelt, das von AFLNet verstanden wird. Die in Kapitel 2.4 beschriebene Struktur einer MQTT Nachricht wurde dazu verwendet. In dieser Nachricht kann der Fixed-Header den Bytes `0x10, 0x30, 0x82, 0xA2, 0xC0` und `0xE0` entsprechen. Diese stehen für die verschiedenen Anfragen `CONNECT, PUBLISH, SUBSCRIBE, UNSUBSCRIBE, PINGREQ` und `DISCONNECT`. Die Variable Header und Payload wurden zufällig generiert und in die Testcases eingefügt. Die Testcases wurden in einem Verzeichnis `aflnet_in/` abgelegt und von AFLNet als Eingabe verwendet. Damit möglichst viele States beim Ausführen des ZuP erreicht werden, wurden ebenfalls ungültige Pakete generiert. Sie dienen AFLNet dazu, unerwartete Zustände des ZuP zu erreichen und somit unerwartete Bugs zu finden. Des Weiteren wurden Nachrichtensequenzen generiert, die den Zustand des ZuP verändern. Hierzu wurden alle Nachrichten, die der ZuP verarbeiten kann, in einer Sequenz aneinandergereiht. Diese Sequenzen wurden ebenfalls in das Verzeichnis `aflnet_in/` abgelegt und von AFLNet als Eingabe verwendet.

5.1.2 Analyse der Effektivität gesendeter Pakete

Die Fuzzing-Kampagne wurde über einen Zeitraum von 48 Stunden durchgeführt. Zur Analyse der Effektivität der gesendeten Pakete wurde eine preload bibliothek verwendet, die die Ausgaben des ZuP der Kanäle `sdtout` und `stderr` in eine Log-Datei `mqtt-aflnet_stdout.log` umleitet. Mit den folgenden Experimenten und Auswertungen soll ein Teil der Forschungsfrage *Q1* 1 beantwortet werden. Nun wird analysiert, wie effektiv die generierten Nachrichten des AFLNet-Fuzzers sind.
Diese Log-Datei wurde auf die Häufigkeit der Ausgaben von erfolgreichen und fehlgeschlagenen Verbindungsaufbauten analysiert.

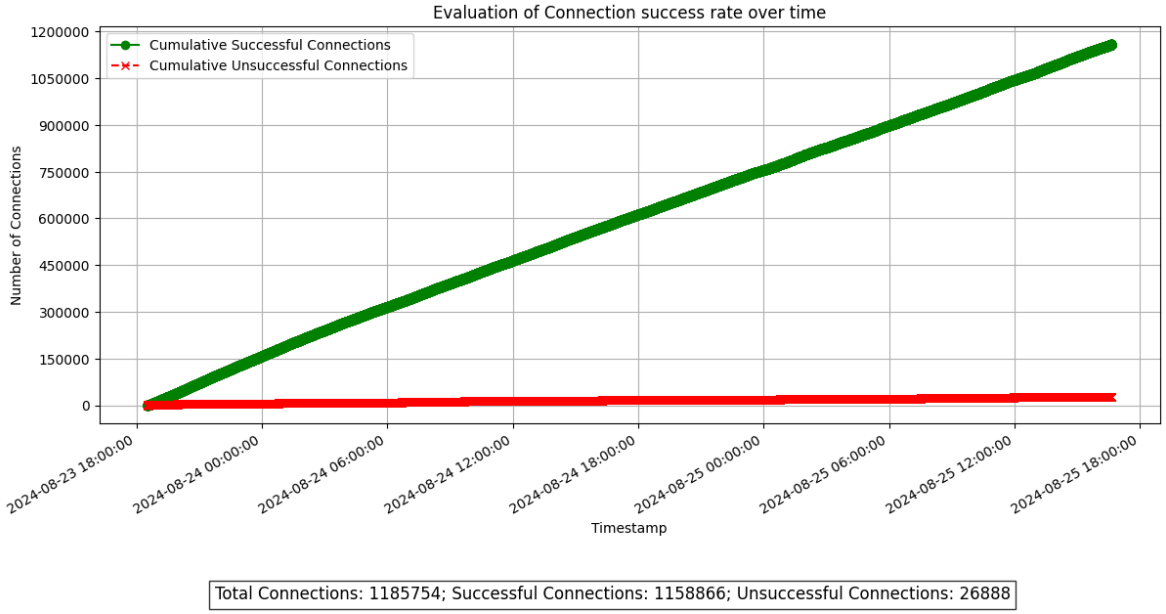

Abbildung 4: Evaluation der Verbindungsaufbauten des ZuP Mosquitto. Die Log-Datei `mqtt-aflnet_stdout.log` wurde auf die Häufigkeit der Ausgaben von erfolgreichen und fehlgeschlagenen Verbindungsaufbauten analysiert.

Die grüne Linie (siehe Abb. 4) repräsentiert die Anzahl der erfolgreichen Verbindungen im Laufe der Zeit. Die Linie steigt linear an und erreicht gegen Ende der Kampagne eine Gesamtzahl von 1.158.866 erfolgreichen Verbindungen. Dies deutet darauf hin, dass das getestete System eine sehr hohe Erfolgsrate bei Verbindungsversuchen aufweist. Die lineare Zunahme zeigt, dass die Rate der erfolgreichen Verbindungen konstant hoch geblieben ist, ohne signifikante Einbrüche oder Schwankungen. Die rote Linie (siehe Abb. 4) zeigt die Anzahl der erfolglosen Verbindungen an. Diese bleibt über den gesamten Zeitraum sehr niedrig und erreicht gegen Ende der Kampagne etwa 26.888 erfolglose Verbindungen. Dies entspricht einer Erfolgsrate von über 97 % und einer Fehlerrate von ca. $\sim 2.32\,\%$ in denen AFLNet keine erfolgreichen Verbindungsaufbauten mit dem MQTT-Broker über das MQTT-Protokoll

entstanden sind. Der minimale Anstieg im Vergleich zu den erfolgreichen Verbindungen weist darauf hin, dass die Mehrheit der Verbindungen erfolgreich war. Bei der Analyse der Fehleranfälligkeit von nicht validen Nachrichten wurde festgestellt, dass am Anfang der Fuzzing-Kampagne viele Fehlermeldungen auftraten. Mit der Zeit wurden jedoch weniger Fehlermeldungen generiert, da die State Machine des ZuP erlernt wurde und somit vermehrt valide Nachrichten generiert wurden.

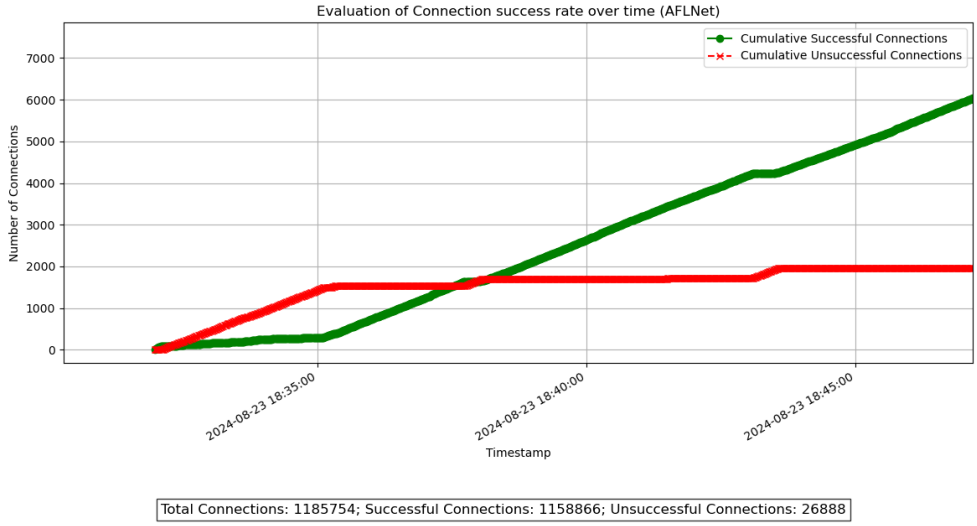

Abbildung 5: Dieses Diagramm zeigt eine vergrößerte Ansicht der ersten 10 Minuten der Fuzzing-Kampagne aus Abbildung 4 mit AFLNet zur Veranschaulichung der Fehleranfälligkeit zu Beginn der Fuzzing-Kampagne. Aus dem Diagramm geht hervor, dass zu Beginn der Kampagne viele erfolglose Verbindungen mit dem ZuP generiert wurden, die jedoch im Laufe der Zeit abnahmen.

Das Diagramm (siehe Abb. 5) zeigt deutlich, dass die Lernphase der State-Machine zu Beginn der Fuzzing-Kampagne von Bedeutung ist. Die hohe Anzahl von erfolglosen Verbindungen in der Anfangsphase deutet darauf hin, dass die anfänglichen Eingaben von AFLNet nicht korrekt auf das getestete System abgestimmt waren. Erst nach einer gewissen Zeit, in der die State-Machine genügend Informationen gesammelt hat, um die Struktur des Protokolls besser zu verstehen, beginnt die Anzahl der erfolgreichen Verbindungen anzusteigen. Dieses Verhalten zeigt, dass das System Zeit benötigt, um die richtigen Eingabenmuster zu lernen. Dieser Lernprozess führt zu einer sukzessiven Verbesserung der Qualität der generierten Eingaben und damit zu einer steigenden Erfolgsrate bei den Verbindungen.

5.1.3 Analyse der Geschwindigkeit

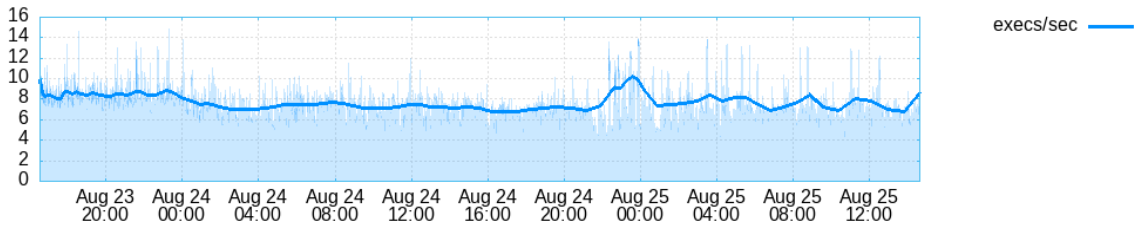

Abbildung 6: Diese Abbildung zeigt die Ausführgeschwindigkeit von AFLNet. Es wird die Anzahl der Nachrichtensequenzen pro Sekunde (y-Achse), die an das ZuP während des Zeitrahmens der Kampagne (x-Achse) gesendet wurden, veranschaulicht.

Die Ausführungsrate ist eine Metrik, um den Fortschritt und die Effektivität einer Fuzzing-Kampagne zu bewerten. Eine hohe und stabile Ausführungsrate deutet auf einen effizienten Fuzzing-Prozess hin, während Schwankungen oder ein Rückgang auf potenzielle Engpässe oder Herausforderungen während der Kampagne hinweisen können.
Die Ausführungsrate liegt im Durchschnitt zwischen 6 und 10 execs/sec und zeigt eine gewisse Variabilität, wie durch die Spitzen und Täler in der hellblauen Schattierung dargestellt. Anzumerken ist, dass das ZuP mit einem AddressSanitizer kompiliert wurde, was die Ausführungsgeschwindigkeit beim Fuzzing verlangsamt. Das resultiert aus dem zusätzlichen Speicher, welchen bei Start des Programms allokiert wird, um die Speicherzugriffe zu überwachen. Zu Beginn der Kampagne gibt es einen kurzen Zeitraum mit erhöhter Variabilität, bevor sich die Ausführungsrate stabilisiert. Im weiteren Verlauf der Kampagne ist eine leichte Abnahme der Ausführungsrate zu beobachten, gefolgt von einer leichten Erholung gegen Ende der Kampagne. Die Kurve zeigt während der gesamten Kampagne eine relativ konstante Ausführungsrate mit periodischen Schwankungen. Diese Schwankungen können auf mehrere Faktoren zurückzuführen sein, darunter:

- Komplexität der zu testenden Eingaben: Komplexere Testfälle können mehr Zeit in Anspruch nehmen, was die Ausführungsrate verringert.
- Systemressourcen: Veränderungen in der Verfügbarkeit von Systemressourcen (z.B. CPU, Speicher) können ebenfalls die Ausführungsrate beeinflussen.
- Optimierungen oder Veränderungen im Fuzzing-Prozess: Wenn das Fuzzing-Tool neue Pfade oder besonders anspruchsvolle Testfälle entdeckt, kann dies die Rate der durchgeführten Tests vorübergehend senken.

Ein leichtes Absinken der Ausführungsrate zwischen dem 24. August, 12:00 Uhr, und dem 25. August, 00:00 Uhr, ist bemerkenswert. Dies könnte darauf hindeuten, dass in dieser Phase anspruchsvollere Testfälle bearbeitet wurden oder dass die Systemressourcen temporär begrenzt waren. Die anschließende Erholung der Ausführungsrate könnte durch die Verarbeitung weniger komplexer Testfälle oder eine bessere Ausnutzung der Systemressourcen erklärt werden. Die insgesamt stabile Ausführungsrate über den Zeitraum von 48 Stunden deutet auf einen stabilen Fuzzing-Prozess hin. Die beobachteten Schwankungen sind im Rahmen dessen, was bei längeren Fuzzing-Kampagnen zu erwarten ist, und weisen auf die normale Dynamik im Fuzzing-Prozess hin, wenn unterschiedliche Testfälle mit variierender Komplexität ausgeführt werden. Die leichte Abnahme der Ausführungsrate gegen Ende der Kampagne könnte als Hinweis darauf gewertet werden, dass das Tool zunehmend anspruchsvollere Testfälle oder Pfade bearbeitet hat. Außerdem anzumerken ist, dass eine Ausführung von AFLNet dem Starten des ZuP, dem anschließenden Senden einer Nachricht aus dem Fuzzing Corpus und dem Beenden des ZuP entspricht. Dem zu entnehmen ist, dass bei Komplexen IoT Protokollen die Ausführungsrate geringer ausfallen kann, da die Initialisierung des ZuP und das Senden einer Nachricht mehr Zeit in Anspruch nehmen kann.

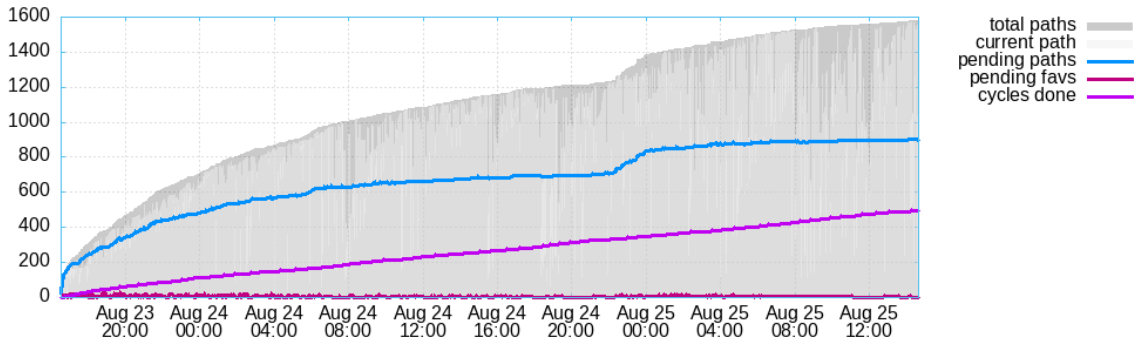

Abbildung 7: Das Diagramm veranschaulicht das Eingabegenerierungsverhalten von AFLNet während der Fuzzing-Kampagne. Die y-Achse gibt einen numerischen Wert an, welcher im Kontext zu den abgebildeten Graphen zu betrachten ist.

Das gezeigte Diagramm veranschaulicht den Verlauf einer 48-stündigen Fuzzing-Kampagne mit dem AFLNet-Tool, wobei verschiedene Metriken im Zusammenhang mit der Testabdeckung und dem Fortschritt des Fuzzing-Prozesses dargestellt werden. Im Folgenden werden die verschiedenen Kurven analysiert und ihre Bedeutung in Bezug auf die Kampagne erläutert.

Die graue Fläche (siehe Abb. 7) zeigt die Gesamtanzahl der explorierten Pfade während der Kampagne. Diese Zahl steigt kontinuierlich an und erreicht gegen Ende der Kampagne über 1400 Pfade. Der gleichmäßige Anstieg deutet darauf hin, dass der Fuzzer während der gesamten Kampagne effektiv neue Pfade identifizieren konnte. Interessanterweise gibt es

Phasen, in denen der Anstieg steiler ist, was auf eine besonders hohe Entdeckungsrate neuer Pfade in diesen Perioden hinweist.
Die hellblaue Linie (siehe Abb. 7) repräsentiert den aktuell im Test befindlichen Pfad. Diese Kurve zeigt, wie das Fuzzing-Tool ständig neue Pfade auswählt und testet. Die Linie verläuft relativ flach, was darauf hinweist, dass das Fuzzing-Tool regelmäßig zwischen den Pfaden wechselt und keiner besonders lange untersucht wird. Diese regelmäßige Bewegung zwischen den Pfaden kann auf eine gleichmäßige Verteilung der Testressourcen hinweisen.
Die Anzahl der noch zu testenden Pfade (blau, siehe Abb. 7) zeigt einen Anstieg, der während der gesamten Kampagne andauert und gegen Ende der 48 Stunden fast 600 erreicht. Dies bedeutet, dass das Fuzzing-Tool konstant neue Pfade entdeckt hat, die es noch zu testen gilt. Der kontinuierliche Anstieg zeigt, dass das System immer neue potenziell interessante Zustände produziert, die eine weitere Untersuchung erfordern.
Die lila Kurve (siehe Abb. 7) stellt die favorisierten Pfade dar, die noch nicht getestet wurden. Diese Zahl steigt ebenfalls stetig an, jedoch langsamer als die „pending paths“(siehe Abb. 7). Dies deutet darauf hin, dass das Fuzzing-Tool bestimmte Pfade als besonders wichtig erachtet und diese bevorzugt behandelt, obwohl noch viele andere Pfade auf ihre Untersuchung warten. Diese Priorisierung kann dazu beitragen, dass besonders interessante oder sicherheitsrelevante Pfade frühzeitig getestet werden. Die magentafarbene Kurve (siehe Abb. 7) zeigt die Anzahl der abgeschlossenen Testzyklen. Diese steigt über die Zeit an, was die Fortschritte im Fuzzing-Prozess darstellt. Ein Zyklus umfasst die vollständige Durcharbeitung des aktuellen Satzes von Testfällen. Der Anstieg zeigt, dass das Fuzzing-Tool kontinuierlich neue Zyklen durchläuft, was auf eine fortlaufende und systematische Prüfung der Pfade hinweist.

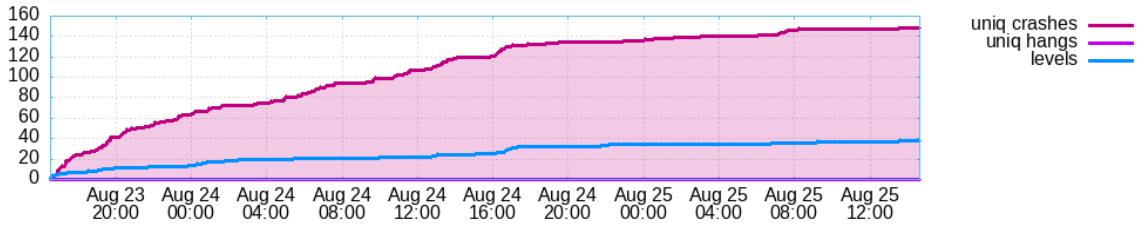

Abbildung 8: Diese Abbildung zeigt die Anzahl der gefundenen Bugs (auf der y-Achse) von AFLNet in Relation zur vergangenen Zeit (x-Achse).

Die Anzahl der einzigartigen Abstürze (pink siehe Abb. 8) zeigt einen kontinuierlichen Anstieg über den gesamten Kampagnenzeitraum. Besonders auffällig ist der steile Anstieg zu Beginn der Kampagne, gefolgt von einem moderateren Anstieg, der sich über den Rest der Kampagne hinweg fortsetzt. Dies deutet darauf hin, dass in den ersten Stunden der Kampagne eine größere Anzahl an Abstürzen entdeckt wurde, was typisch für Fuzzing-Kampagnen ist, bei denen anfänglich viele triviale Schwachstellen gefunden werden. Im weiteren Verlauf

der Kampagne werden dann weniger neue Abstürze entdeckt, was darauf hinweist, dass das System bereits viele der simpleren potenziellen Schwachstellen aufgedeckt hat. Die Anzahl der einzigartigen Hänger (magenta, siehe Abb. 8) bleibt über den gesamten Kampagnenzeitraum stabil bei null. Dies könnte darauf hindeuten, dass entweder keine Hänger im getesteten System aufgetreten sind oder dass das verwendete Fuzzing-Setup nicht in der Lage war, solche Zustände effektiv zu identifizieren. Die Levels-Kurve (blau, siehe Abb. 8) zeigt nur einen leichten Anstieg in den ersten Stunden der Kampagne und bleibt danach weitgehend konstant. Dies könnte darauf hinweisen, dass nur wenige neue Code-Pfade (Levels) während des Fuzzings entdeckt wurden, was möglicherweise auf die Komplexität des Zielsystems oder auf die Grenzen der verwendeten Mutationsstrategien hinweist. Die Fuzzing-Kampagne mit AFLNet war erfolgreich darin, eine beträchtliche Anzahl einzigartiger Abstürze zu entdecken, insbesondere in den ersten Stunden des Fuzzings. Der Mangel an identifizierten Hängern und der begrenzte Anstieg an Levels könnte jedoch darauf hinweisen, dass entweder das Zielsystem keine anfälligen Hängerzustände aufweist oder dass das Fuzzing-Setup verbessert werden könnte, um diese Art von Fehlern besser zu identifizieren.

5.1.4 Analyse der gefundenen Bugs

Die gefundenen Bugs nach der 48-stündigen Fuzzing-Kampagne belaufen sich auf 151 Bugs (siehe Abb. 8). Hierzu wurden alle Abstürze mithilfe von Bash-Skripts dokumentiert [37] und analysiert [38]. Bei allen gefundenen Bugs handelt es sich um *Segmentation Faults* und konnten in vier Kategorien eingeteilt werden:

1. Absturz durch Buffer Overflow in der Funktion `packet__write_bytes` in `libpacket_datatypes.c`
2. Absturz durch Buffer Overflow in der Funktion `packet__write_bytes` in `libpacket_datatypes.c` mit einer QoS-Nachricht
3. Absturz durch Buffer Overflow in der Funktion `packet__write_bytes` in `libpacket_datatypes.c` mit einer QoS-Nachricht und einem subscriber
4. Absturz durch Buffer Overflow in der Funktion `handle__publish` in `srchandle_publish.c`

Die Abstürze der vierten Kategorie sind auf die eigene Implementierung eines Buffer Overflows zurückzuführen. Dieser Buffer Overflow wird getriggert, da die Sanity-Checks (1) in der Funktion `handle__publish` auskommentiert wurden. Diese Sanity-Checks überprüfen, ob die Länge der Nachricht, die an den Broker gesendet wird, korrekt ist. Der Fehler tritt nach

dem auskommentierten Sanity-Check auf, wenn darauf geprüft wird, ob die Nachricht, die gesendet werden soll bereits existiert. Hierzu soll die gesendete Nachricht – mit einer bereits vorhandenen Nachricht-ID – mit der bereits zwischengespeicherten Nachricht verglichen werden. Diese Nachricht mit der gleichen ID ist jedoch größer als die bereits zwischengespeicherte Nachricht und führt bei dem Vergleich mit `memcmp` zum Absturz des ZuP.
Die anderen drei Kategorien sind gleichen Ursprungs. Hier handelt es sich um einen Buffer Overflow in der Funktion `packet__write_bytes` in der Datei `libpacket_datatypes.c`. Diese Schwachstelle wird bei dem Senden der Nachricht an ein Topic ausgelöst. Hierbei werden Nachrichten vom Broker empfangen und das Topic weitergeleitet. Wenn Subscriber auf das Topic warten, wird die Nachricht an die Subscriber mittels des Syscalls `memcpy` weitergeleitet.
Die Gesamterscheinungen der Bugs belaufen sich auf 151 Bugs, von denen 105 dem Fehler in der Funktion `handle__publish` entsprechen. Die restlichen 46 Bugs entstammen den Buffer Overflows in der Funktion `packet__write_bytes`.
Die gefundenen Bugs wurden mithilfe des Tools `aflnet-replay` reproduziert und auf ihre Validität überprüft. Sie zeigten jedoch keine Wirkung auf den Broker ohne manuell implementierte Schwachstellen.

Zusammenfassend lässt sich sagen, dass AFLNet nur zwei Schwachstellen im Broker Mosquitto gefunden hat, von denen beide von der Implementierung der Sanity-Checks abhängen. Wichtig anzumerken ist, dass AFLNet einen Absturz genau dann als einzigartig betrachtet, wenn die an das ZuP übergebene Nachricht einen neuen Pfad im Code des ZuP erreicht. Die Stelle, an der das ZuP zum Absturz gebracht wird, ist für AFLNet nicht ausschlaggebend.

5.2 Fuzzing mit boofuzz

In dem folgenden Kapitel wird das Aufsetzen der Testumgebung und die darin durchgeführten Experimente mit dem Fuzzing-Tool boofuzz beschrieben.

5.2.1 Setup

Zur Entwicklung einer Fuzzing-Kampagne mit boofuzz sind mehrere Schritte erforderlich. Zunächst muss das Zielsystem identifiziert und analysiert werden, um die relevanten Protokollspezifikationen und Kommunikationsmuster zu verstehen. Darauf aufbauend werden die Testfälle definiert, die die verschiedenen Protokollfelder und Eingabeparameter abdecken sollen. Die Testfälle werden in boofuzz als *Fuzz-Template* definiert, das die Struktur und den Inhalt der zu füllenden Nachrichten beschreibt. Das Fuzz-Template besteht aus einer

Reihe von Feldern, die die verschiedenen Teile der Nachricht repräsentieren, wie z.B. den Fixed Header, den Variable Header und den Payload bei MQTT-Nachrichten. Jedes Feld kann verschiedene Eigenschaften wie den Typ, die Länge und den Wertebereich der Eingabe enthalten. In dem Fall der MQTT-Nachrichten können die Felder z.B. den Steuerpakettyp, die QoS-Ebene und den Topic-Namen, wie in Kapitel 2.4 beschrieben, repräsentieren. Hierbei ist es wichtig, die Protokollspezifikationen und die erwarteten Eingabewerte zu berücksichtigen, um realistische Testfälle zu erstellen. Nachdem die Testfälle definiert sind, wird eine Fuzzing-Kampagne gestartet, bei der boofuzz automatisch eine Vielzahl von zufälligen oder systematischen Eingaben generiert und an das Zielsystem sendet. Während des Fuzzings kann boofuzz den Zustand des Zielsystems überwachen und auf unerwartete Ereignisse wie Abstürze oder Fehler reagieren. Dieser Schritt ist jedoch nicht standardmäßig eingebaut und muss vom Benutzer implementiert werden. Die Implementierung der Reaktion auf unerwartete Ereignisse kann z.B. das Neustarten des Zielsystems, das Anpassen der Fuzzing-Strategie oder das Protokollieren von Fehlern umfassen und erfolgt über ein in boofuzz implementiertes Interface. Dieses Interface beinhaltet die Monitor-Klassen `ProcessMonitor, NetworkMonitor` und `CallbackMonitor`. Die `ProcessMonitor`-Klasse überwacht den Zustand des Zielsystems, indem sie den Prozessstatus überwacht und auf Abstürze oder Fehler reagiert. Dieser Monitor wurde zum Beispiel in der Analyse des MQTT-Brokers Mosquitto verwendet, um auf Abstürze des Brokers zu reagieren und die in den Abstürzen enthaltene debug-Informationen abzugreifen und zu speichern.

Eine in dieser Arbeit verwendete Kernkomponente von boofuzz ist die Option ein Feld als `fuzzable` zu definieren. Ein `fuzzable` Feld ist ein Feld, das zufällig generierte Eingaben erhalten darf und somit für das Fuzzing relevant ist. Mit dieser Option können Testfälle erstellt werden, die systematisch verschiedene Eingaben in bestimmte Felder einfügen, um präzisere und effektivere Fuzzing-Kampagnen durchzuführen. Die `fuzzable`-Option ermöglicht es, die Eingaben gezielt zu steuern und bestimmte Protokollfelder oder Eingabeparameter zu testen, um potenzielle Schwachstellen zu identifizieren. Gerade im Fall von MQTT-Nachrichten ist es wichtig, die verschiedenen Felder wie den Fixed Header, den Variable Header und den Payload systematisch zu füllen.

Für das Entwickeln einer Fuzzing-Kampagne wurden im Rahmen dieser Arbeit zwei Module entwickelt:

- boo-fuzzer-mqtt
- boo-fuzzer-mqtt-efficient

Das Modul `boo-fuzzer-mqtt` enthält die Implementierung einer Fuzzing-Kampagne für den MQTT-Broker weitestgehend ohne statisch definierte Felder. Alle Teilaspekte eines Pakets als fuzzable zu definieren kann in vielen Fällen Sinn machen, um die gesamte Struktur eines Netzwerkpakets zu testen. Hierzu können alle individuellen Teile einer MQTT-Nachricht auf richtige Verarbeitung getestet werden.
Das Modul `boo-fuzzer-mqtt-efficient` hingegen definiert statische Felder für den Fixed Header, den Variable Header und das Topic. Das Topic wird hierbei als `fuzzing/topic` fest vordefiniert und gepublisht, sodass jede einhergehende Client-Verbindung bereits ein Topic zur Verfügung hat. Das hat zur Folge, dass die Wahrscheinlichkeit, dass ein gültiges Paket generiert wird, signifikant erhöht wird. Für das Starten einer Fuzzing-Kampagne mit boofuzz sind folgende Schritte notwendig:

- Installation von boofuzz und den erforderlichen Abhängigkeiten
- Definition der Testfälle und Fuzzing-Strategien
- Starten der Fuzzing-Kampagne und Überwachung des Zielsystems
- Analyse der Ergebnisse und Identifizierung von Schwachstellen

Zusätzlich kommt hinzu, dass bei der effizienten Lösung der Broker bereits in einen präparierten Zustand versetzt wird, sodass weniger Overhead bei der initialen Fuzzingstrategie entsteht. Außerdem muss ein Monitor für das ZuP definiert werden, sodass auch das automatische Starten, Neustarten und Beenden des Zielsystems möglich ist und bei Abstürzen die entsprechenden Informationen des Absturzes sammeln zu können.

5.2.2 Analyse der Effektivität gesendeter Pakete

In der Untersuchung von MQTT-Brokern mittels Fuzzing ist es entscheidend, die Wahrscheinlichkeit zu verstehen, mit der ein zufällig generiertes Paket tatsächlich gültig ist. Diese Wahrscheinlichkeit hängt stark davon ab, welche Teile des MQTT-Pakets zufällig generiert werden und welche festgelegt (d.h. nicht gefuzzt) sind. In diesem Abschnitt wird die Berechnung dieser Wahrscheinlichkeit detailliert beschrieben, sowohl für den Fall, dass alle Teile des Pakets gefuzzt werden, als auch für den Fall, dass bestimmte Teile, wie der Fixed Header und der Variable Header, festgesetzt sind. Dieser Teil adressiert die Forschungsfrage *Q1* 1 und beschäftigt sich mit der Auswertung und Analyse der Effektivität und Erfolgsrate der generierten Eingaben mit boofuzz.

Ein MQTT-Paket besteht aus mehreren wesentlichen Teilen:

- Fixed Header: Enthält den Steuerpakettyp (z.B. `PUBLISH`, `SUBSCRIBE`) und Flags sowie die verbleibende Länge des Pakets.
- Variable Header: Enthält je nach Steuerpakettyp spezifische Informationen wie z.B. Paket-IDs, QoS-Level oder den Namen des Topics.
- Payload: Enthält die tatsächlichen Daten, die gesendet werden (z.B. die Nachricht im `PUBLISH`-Paket) oder eine Liste von Topic-Filtern im `SUBSCRIBE`-Paket.

Bit	7	6	5	4	3	2	1	0
byte 1	MQTT Control Packet Type (2)				Reserved			
	0	0	1	0	0	0	0	0
byte 2	Remaining Length (2)							
	0	0	0	0	0	0	1	0

Abbildung 9: Aus der Spezifikation von MQTT Version 3.1.1 [15] zeigt die Struktur des Fixed Headers in einem MQTT-Paket. Er besteht aus 2 Bytes und enthält den Steuerpakettyp (*hier* 0010 für `CONNECT`) und Flags (hier *Reserved*) sowie die verbleibende Länge des Pakets.

Der Fixed Header besteht aus drei Hauptbestandteilen 9:

- Steuerpakettyp: (4 Bits): MQTT definiert 14 Pakettypen (im Bereich von `0001` bis `1110`), was bedeutet, dass es 14 mögliche Werte für diese 4 Bits gibt
- Flags (4 Bits): Die Flags hängen vom Pakettyp ab, was die Anzahl der gültigen Kombinationen verringert
- Verbleibende Länge: Die verbleibende Länge gibt an, wie viele Bytes nach dem Fixed Header folgen

Die Spezifikation von MQTT sieht vor, dass von den 14 möglichen Werten für den Steuerpakettyp nur 10 gültige Werte für einen validen Steuerpakettypen existieren, die von dem Client zum Broker gesendet werden können [39]. Ausgegangen wird von der Generierung eines validen Connect Pakets, welches den Steuerpakettyp `CONNECT` hat. Betrachtet wird zunächst die Wahrscheinlichkeit, dass der Fixed Header gültig ist. Hierbei wird folgende Annahme getroffen:

$$P(\text{gültiger Fixed Header}) = P(\text{gültiger Steuerpakettyp}) \times P(\text{gültige Flags}) \times P(\text{gültige verbleibende Länge})$$

Die Wahrscheinlichkeit, dass der Steuerpakettyp gültig ist, beträgt:

$$P(\text{gültiger Steuerpakettyp}) = \frac{M}{N} = \frac{1}{16}$$

Wobei M die Anzahl der gültigen Werte für den Steuerpakettyp `CONNECT` 0001 ist und N die Anzahl der möglichen Werte, die für die 4 Bits des Steuerpakettyps verwendet werden können. Die Wahrscheinlichkeit, dass die Flags gültig sind, beträgt:

$$P(\text{gültige Flags}) = \frac{K}{N} = \frac{1}{16}$$

Wobei K die Anzahl der gültigen Kombinationen für die Flags ist und N die Anzahl der möglichen Werte für die 4 Bits der Flags. Hierbei ist zu beachten, dass die Flags bei einem `CONNECT`-Paket reserviert sind und somit nur die gültige Kombination 0000 für die Flags existieren.
Das Zweite Byte des Fixed Headers enthält die verbleibende Länge des Pakets. Die verbleibende Länge ist ein variable-length-integer, der die Anzahl der Bytes angibt, die nach dem Fixed Header folgen. Sie kann Werte von 0 bis 127 annehmen. Somit ergibt sich die Wahrscheinlichkeit, dass die verbleibende Länge gültig ist:

$$P(\text{gültige verbleibende Länge}) = \frac{128}{256}$$

Die Wahrscheinlichkeit, dass der Fixed Header gültig ist, beträgt:

$$P(\text{gültiger Fixed Header}) = \frac{1}{16} \times \frac{1}{16} \times \frac{128}{256} \approx 1.938 \times 10^{-3}$$

Der Variable Header enthält zusätzliche Informationen, die je nach Pakettyp variieren. Für ein `CONNECT`-Paket enthält der Variable Header die Bytes für den Protokollnamen, die Protokollversion, Connect Flags und Keep Alive. Die Länge des Variable Headers beträgt 10 Bytes. Die Wahrscheinlichkeit, dass der Variable Header gültig ist, beträgt:

$$\begin{aligned} P(\text{gültiger Variable Header}) &= P(\text{gültige Länge des Protokollnamens}) \\ &\times P(\text{gültiger Protokollnamen}) \times P(\text{gültige Protokollversion}) \times P(\text{gültige Connect Flags}) \\ &\times P(\text{gültiges Keep Alive}) \end{aligned} \quad (1)$$

Die Länge des Protokollnamens ist festgelegt und beträgt 2 Bytes. Die Wahrscheinlichkeit, dass die Länge des Protokollnamens gültig ist, beträgt:

$$P(\text{gültige Länge des Protokollnamens}) = \frac{1}{256} \times \frac{1}{256}$$

	Description	7	6	5	4	3	2	1	0
Protocol Name									
byte 1	Length MSB (0)	0	0	0	0	0	0	0	0
byte 2	Length LSB (4)	0	0	0	0	0	1	0	0
byte 3	'M'	0	1	0	0	1	1	0	1
byte 4	'Q'	0	1	0	1	0	0	0	1
byte 5	'T'	0	1	0	1	0	1	0	0
byte 6	'T'	0	1	0	1	0	1	0	0

Abbildung 10: Zeigt die Struktur des Protokollnamens, entnommen aus der Spezifikation von MQTT Version 3.1.1 [40]. Der Protokollname ist festgelegt und muss `MQTT` sein und ist sechs Bytes lang.

Der Protokollname ist festgelegt auf vier Bytes und muss `MQTT` sein 10. Aus der Struktur in der Abbildung 10 ergibt sich die Wahrscheinlichkeit, dass der Protokollname gültig ist:

$$P(\text{gültiger Protokollname}) = \frac{1}{256}^{4}$$

Zudem kommt hinzu, dass die Protokollversion festgelegt ist und `0x4` sein muss. Die Protokollversion ist ein Byte lang und kann 256 mögliche Werte annehmen. Somit ergibt sich die Wahrscheinlichkeit, dass die Protokollversion gültig ist:

$$P(\text{gültige Protokollversion}) = \frac{1}{256}$$

Die Connect Flags sind ein Byte lang und können 256 mögliche Werte annehmen. Die Wahrscheinlichkeit, dass die Connect Flags gültig sind, erschließt sich aus folgendem Regelwerk:

- Bit 0 muss immer 0 sein
- Bit 1 kann 0 oder 1 sein. Wenn Bit 1 den Wert 0 hat, dann müssen die Bits 3-5 auch 0 sein
- Bit 2 kann 0 oder 1 sein
- Bit 3 und 4 können die Werte 00, 01 oder 10 annehmen, wenn Bit 1 den Wert 1 hat

- Bit 5 kann nur 1 sein, wenn Bit 1 den Wert 1 hat, ansonsten muss Bit 5 den Wert 0 besitzen
- Bit 6 und 7 können 0 oder 1 sein

Daraus ergibt sich die Wahrscheinlichkeit, dass die Connect Flags gültig sind:

$$P(\text{gültige Connect Flags}) = P(\text{Bit0}) \times P(\text{Bit1}) \times P(\text{Bit2}) \times P(\text{Bit3,4}) \times P(\text{Bit5}) \times P(\text{Bit6,7}) \tag{2}$$

- Wenn Bit 2 = 0:
  - Bit 3 + 4 = 00 (1 Möglichkeit)
  - Bit 5 = 0 (1 Möglichkeit)

$$P(\text{Bit 3 + 4 + 5} \mid \text{Bit 2} = 0) = \frac{1}{8}$$

(es gibt 8 mögliche Kombinationen für Bit 3, 4 und 5, aber nur eine gültige) (3)

- Wenn Bit 2 = 1:
  - Bit 3 + 4 = 00, 01 oder 10 (3 Möglichkeiten)
  - Bit 5 = 0 oder 1 (2 Möglichkeiten)

$$P(\text{Bit 3 + 4 + 5} \mid \text{Bit 2} = 1) = \frac{6}{8}$$

(es gibt 8 mögliche Kombinationen für Bit 3, 4 und 5, und 6 davon sind gültig) (4)

$$P(\text{Bit 3 + 4 + 5}) = P(\text{Bit 2} = 0) \times P(\text{Bit 3 + 4 + 5} \mid \text{Bit 2} = 0) + P(\text{Bit 2} = 1) \times P(\text{Bit 3 + 4 + 5} \mid \text{Bit 2} = 1) \tag{5}$$

$$P(\text{Bit 3 + 4 + 5}) = \frac{1}{2} \times \frac{1}{8} + \frac{1}{2} \times \frac{6}{8} = \frac{1}{16} + \frac{6}{16} = \frac{7}{16}$$

Somit ergibt sich aus der Gleichung 2 die Wahrscheinlichkeit, dass die Connect Flags gültig sind:

$$P(\text{gültige Connect Flags}) = \frac{1}{2} \times 1 \times 1 \times \frac{7}{16} \times 1 \times 1 = \frac{7}{32}$$

Das Keep Alive-Intervall ist ein 2-Byte-Wert, der die Zeit in Sekunden angibt, die der Client auf eine Antwort des Brokers wartet. Das Intervall kann Werte von 0 bis 65535 annehmen. Die Wahrscheinlichkeit, dass das Keep Alive-Intervall gültig ist, beträgt 1, da alle Werte innerhalb der zwei Bytes zugelassen werden.
Die Wahrscheinlichkeiten für einen gültigen Variable Header sind somit:

$$P(\text{gültiger Variable Header}) = \left(\frac{1}{256}\right)^2 \times \left(\frac{1}{256}\right)^4 \times \frac{1}{256} \times \frac{7}{32} \times 1 \approx 3.036 \times 10^{-18}$$

Um die Anzahl der Versuche zu berechnen, die notwendig sind, um mit einer 99-prozentigen Wahrscheinlichkeit ein gültiges Paket mit korrekt generiertem Fixed und Variable Header zu erzeugen, wird folgende Formel verwendet:

$$n = \frac{\log(1 - P_{\text{Ziel}})}{\log(1 - P_{\text{Erfolg}})}$$

Dabei beschreibt P_{Ziel} die Zielwahrscheinlichkeit, dass ein gültiges Paket generiert wird, und P_{Erfolg} die Wahrscheinlichkeit, dass ein gültiger Fixed und Variable Header generiert wird. Die Gesammtwahrscheinlichkeit für ein Paket mit einem gültigen Fixed und Variable Header beträgt:

$$P(\text{Erfolg}) = P(\text{gültiger Fixed Header}) \times P(\text{gültiger Variable Header}) = 1.938 \times 10^{-3} \times 3.036 \times 10^{-18} \approx 5.89 \times 10^{-21} \quad (6)$$

Nun kann die Anzahl der Versuche berechnet werden, die notwendig sind, um mit einer 99-prozentigen Wahrscheinlichkeit ein gültiges Paket zu generieren:

$$n = \frac{\log(1 - 0.99)}{\log(1 - 5.89 \times 10^{-21})} \approx 7.819 \times 10^{20}$$

Bei einer gemessenen Ausführgeschwindigkeit von ca. 78 Paketen pro Sekunde würde es also ca. 10^{19} Sekunden dauern, um mit einer Wahrscheinlichkeit von 99 % ein gültiges Paket zu generieren. Das entspricht ca. 3.22×10^{11} Jahren für die Generierung eines korrekten Fixed und Variable Headers.
Diese Berechnungen zeigen, dass das Fixieren von Teilen des Pakets (wie Fixed Header und Variable Header) die Anzahl der benötigten Pakete zur Erzeugung eines gültigen Pakets erheblich reduziert. Dies ist besonders wichtig für die Effizienz von Fuzzing-Tests, da es die Anzahl der Tests verringert, die benötigt werden, um die gewünschten Ergebnisse zu erzielen.

Durch das Fixieren bestimmter Schlüsselparameter des MQTT-Pakets, wie dem Fixed Header und dem Topic, wird die Wahrscheinlichkeit, dass ein gültiges Paket generiert wird, signifikant erhöht. Dies ist besonders wichtig für effizientes Fuzzing, da es die Anzahl der möglichen ungültigen Pakete reduziert und die Effektivität der Tests steigert.

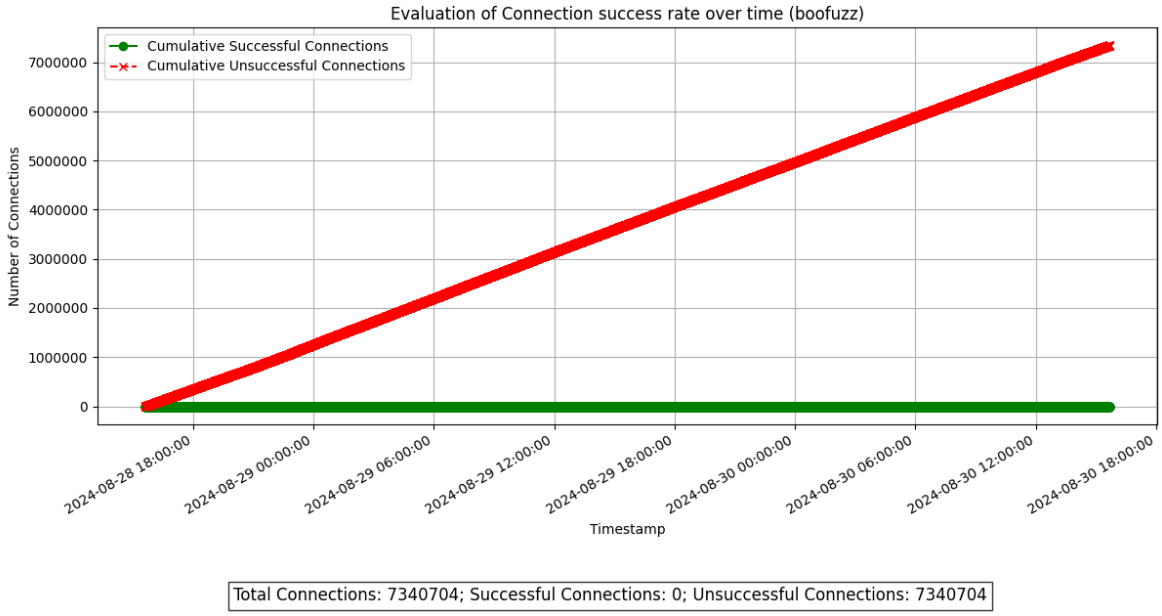

Abbildung 11: Vergleich der gültigen Pakete bei zufälligem Fuzzing mit boofuzz ohne statisch definierte Felder

Diese Grafik 11 ist das Ergebnis einer Fuzzing-Kampagne, bei der über einen Zeitraum von 48 Stunden eine extrem hohe Anzahl an Verbindungsversuchen durchgeführt wurde. Auch trotz der hohen Anzahl an Verbindungsversuchen, die in dieser Grafik dargestellt sind, konnte kein gültiges Paket generiert werden. Das Fixieren der bereits genannten Felder führte bei einer weiteren Kampagne über 48 Stunden auch zu keinem gültigen Paket.

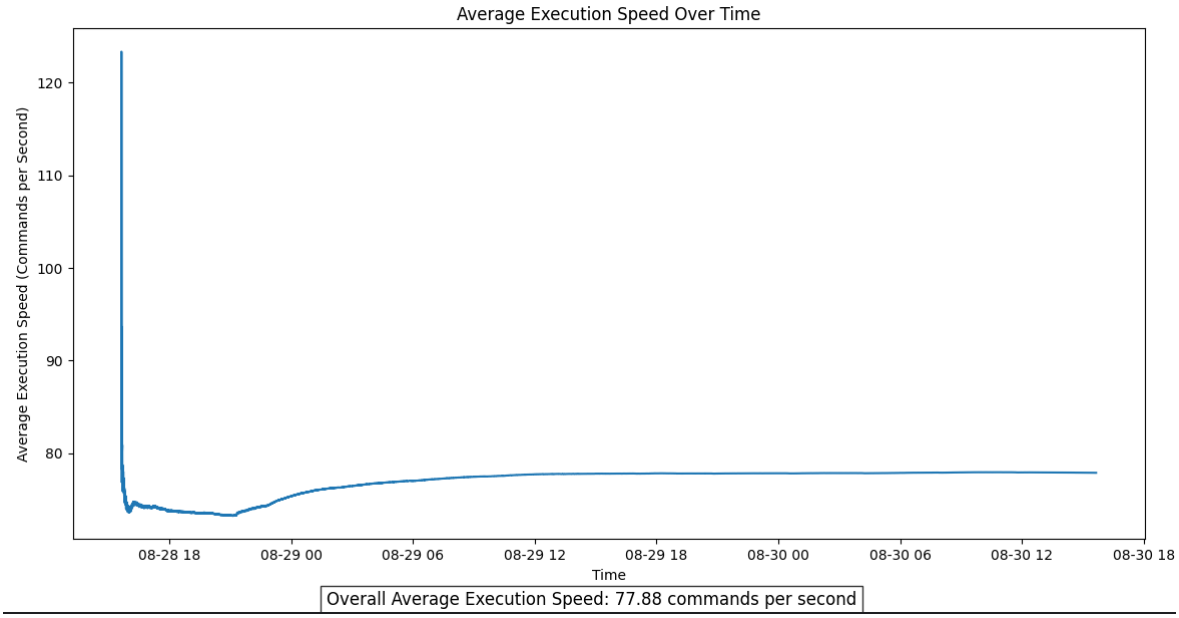

Abbildung 12: Durchschnittliche Ausführungsgeschwindigkeit von boofuzz in einem Zeitraum von 48 Stunden

Die durchschnittliche Ausführungsgeschwindigkeit von boofuzz ist in dieser Grafik dargestellt. In ihr ist zu erkennen, dass die Geschwindigkeit des Fuzzings über den Zeitraum von 48 Stunden relativ konstant bleibt. Die einzige Ausnahme bildet der Anfangszeitraum der Fuzzing-Kampagne, in dem die Geschwindigkeit kurzzeitig rapide abfällt. Dies hängt mit der Mutationsstrategie von boofuzz zusammen, die zu Beginn der Kampagne eine Vielzahl von großen Mutationen durchführt, um das Zielsystem zu erkunden. Zunächst wurden relativ kleine Pakete generiert, die jedoch aufgrund der geringen Wahrscheinlichkeit eines gültigen Pakets keine Ergebnisse lieferten. Im späteren Verlauf wurden jedoch größere Pakete generiert, was die Mutation erheblich verlangsamte. Wichtig anzumerken ist die Strategie von boofuzz. Trotz definition von statischen feldern im Header-Bereich wurden zufällige Bytes an den Kopf des Pakets angehängt. Dies führte dazu, dass die Wahrscheinlichkeit eines gültigen Pakets weiterhin sehr gering war.

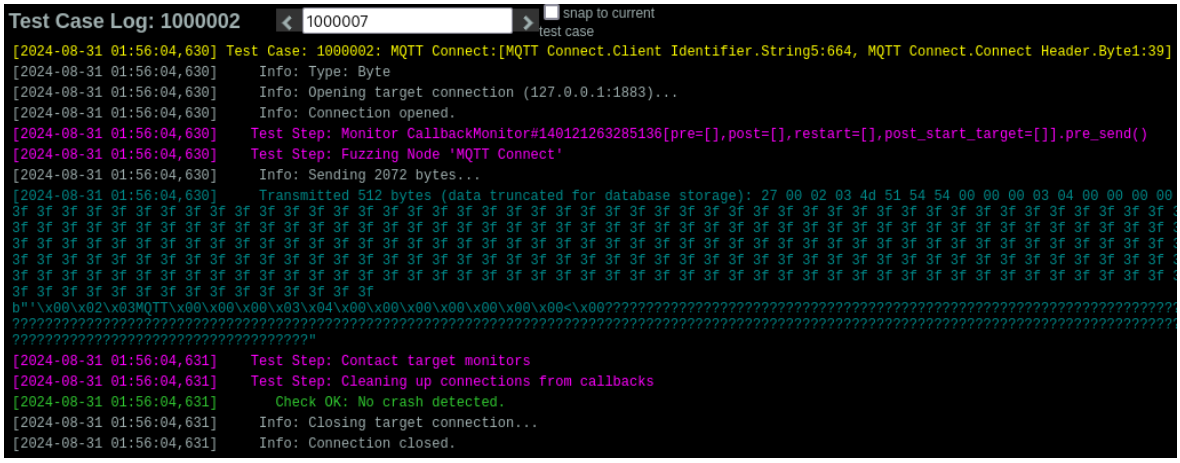

Abbildung 13: Diese Grafik zeigt einen Einblick zur Generierung von weiterhin ungültigen Paketen aufgrund des zufälligen Anhängens von Bytes im Header des Pakets.

Erkennbar ist die Generierung falscher Pakete durch den festen definierten Header `0x020x03MQTT`, welcher aus der Abbildung 13 hervorgeht.

Aus diesem Kapitel geht hervor, dass die Effektivität dieses Fuzzers stark von der Definition mehrerer Felder abhängt. Im Falle von komplexeren Protokollen wie MQTT ist es wichtig, dass eine Definition der Struktur der Felder in der Implementierung des Fuzzers vonnöten ist, sodass die generierten Eingaben möglichst Effektiv sind. Hierzu ist es wichtig, mehrere Aspekte des Protokolls zu definieren und zu fuzzen, um möglichst strukturiert und effektiv an die Generierung von gültigen Paketen heranzugehen. Dies kann in Zukunft dadurch umgesetzt werden, dass mehrere Instanzen des Fuzzers gestartet werden, die jeweils auf ein bestimmtes Feld des MQTT-Pakets fokussiert sind und somit die Wahrscheinlichkeit eines gültigen Pakets erhöht wird.

5.3 Pulsar

Im Folgenden werden alle Schritte zur Vorbereitung und Durchführung der Fuzzing Kampagne mit Pulsar beschrieben und anschließend der daraus entstandene Erkenntnisgewinn geschlussfolgert.

5.3.1 Setup

Pulsar wurde mithilfe des von Pahl bereitgestellten Docker-Images [32] installiert. Das Docker-Image wurde auf einem Ubuntu 20.04 LTS System installiert. Weitere Abhängigkeiten, welche nicht in der Installation der Docker-Images enthalten sind, wurden manuell installiert und

in einer `requirements.txt` Datei festgehalten. Für das Fuzzing mit Pulsar wird ein Netzwerkverkehr benötigt. Dieser Netzwerkverkehr wird in Form von pcap-Dateien gespeichert. Hierfür wurde ein Python-Skript entwickelt, welches MQTT spezifische Nachrichten an den Broker sendet und möglichst viele Zustände vom Broker erreicht. Die Nachrichten werden mithilfe des Tools `tcpdump` aufgezeichnet und in pcap-Dateien abgelegt. Die pcap-Dateien werden in einem Ordner *traffic* abgelegt. Für den Start werden zusätzlich die pcap-Dateien im Ordner *traffic* über ein live-volume in den Docker-Container eingehängt. Alle gesammelten pcap-Dateien dienen für den weiteren Verlauf der Untersuchung. Pulsar verwendet die pcap-Dateien, um die State Machine zu erlernen.

Die Daten wurden in Kombination eines Python-Skripts, der Fuzzing Kampagne mit AFLNet und dem Tool `tcpdump` gesammelt. Das Python-Skript [41] und AFLNet [42] wurden im Falle der Vorbereitung auf die Fuzzing Kampagne mit Pulsar als Input-Generatoren verwendet. Mithilfe dieser Tools wurde der Netzwerkverkehr aufgezeichnet und in pcap-Dateien abgelegt. Die pcap-Dateien wurden in den Docker-Container mithilfe eines live-volumes eingehängt und Pulsar wurde gestartet [43]. Die gesammelten Daten wurden dann von Pulsar als Trainingsdaten verwendet, um das zu analysierende Protokoll zu erlernen.

5.3.2 Erkenntnisgewinn

Die Erkenntnisse des Fuzzing Kampagne mit Pulsar belaufen sich dabei auf die Art und Weise, wie Pulsar den Netzwerkverkehr analysiert und das Protokoll erlernt. Bei der Generierung von Input mit AFLNet und dem Python-Skript wurde darauf geachtet, dass möglichst viele Zustände erreicht werden. Jedoch wurde bei der Analyse des Netzwerkverkehrs festgestellt, dass Pulsar nicht in der Lage war, die State Machine des MQTT Protokolls korrekt zu erlernen. Die Ursache hierfür ist, dass Pulsar die Startsequenz des Protokolls nicht korrekt erlernen konnte. Durch das Generieren von Input mit AFLNet wurden viele falsche Pakete generiert, welche auch häufiger auftraten. Mit unter ist das der Fall gewesen, dass Pulsar eine oft vorkommende Sequenz von vielen *A*s als Startsequenz des Protokolls interpretiert hat. Diese Sequenzen sollten einen Fehlerzustand im Protokoll darstellen und nicht als Startsequenz interpretiert werden. Da jedoch viele Implementierungen von Fehlerbehebungen in der State Machine des Protokolls vorhanden sind, wurde von AFLNet auch eine Vielzahl von Fehlerzuständen generiert. Dies liegt an der Art, wie AFLNet geeignete eingaben bewertet. Eine Eingabe, die von AFLNet als geeignet bewertet wird, wird auch häufiger generiert. Diese geeigneten Eingaben werden als geeignet bewertet, wenn sie viele Zustände erreichen. Da jedoch viele Fehlerzustände in der State Machine des Protokolls vorhanden sind, werden auch

viele Fehlerzustände generiert. Die vorhergehenden Versuche eines Verbindungsaufbaus mit einer Connect-Nachricht wurden nicht als Startsequenz interpretiert. Dies hatte zur Folge, dass im Zeitrahmen dieser Arbeit Pulsar nicht effektiv zum Laufen gebracht werden konnte. Pulsar benötigt nämlich viel Zeit eine Markov-Kette zu erlernen und die State Machine zu approximieren, wenn besonders viel Netzwerkverkehr aufgezeichnet wurde. Im Rahmen dieser Arbeit wurden bis zu 7 Gigabyte an Netzwerkverkehr aufgezeichnet und Pulsar benötigte mehrere (ca. 35) Stunden, um die State Machine zu erlernen.

Eine weitere Erkenntnis ist – aufgrund der Geschwindigkeit des Lernprozesses – dass Pulsar noch keine Hardwarebeschleunigung mit bspw. einer GPU (Graphics Processing Unit) unterstützt. Die Hardwarebeschleunigung würde den Lernprozess beschleunigen und somit auch die State Machine schneller erlernen.

5.4 Vergleich der erhobenen Metriken

In diesem Kapitel werden die erhobenen Metriken der Fuzzer AFLNet, Pulsar und boofuzz miteinander verglichen. Die Metriken wurden anhand der Anzahl der generierten Eingaben pro Sekunde und der Anzahl der reproduzierten Bugs gemessen.

Bei der Reinen Betrachtung der Anzahl der gesendeten Eingaben pro Sekunde ist Pulsar der langsamste Fuzzer. Aufgrund der Implementierung des Fuzzers ist es ohne selbst geschriebene Skripte nicht möglich, die von Pulsar generierten Eingaben zu automatisiert an das zu testende Programm zu senden. Die Anzahl der generierten Eingaben pro Sekunde, die an das ZuP gesendet werden, ist somit vom Tester abhängig. Die Frage der Automatisierung stellt sich hierbei als entscheidend heraus, da die manuelle Eingabe von Eingaben in das ZuP die Geschwindigkeit des Fuzzers erheblich beeinflusst. Der dadurch einhergehende Mangel von Automatisierung führt dazu, dass ein weiteres Monitoring des ZuP notwendig ist, um die generierten Eingaben und den Zustand des ZuP zu überwachen.
Der nächst schnellere Fuzzer ist AFLNet mit durchschnittlich 8 Ausführungen pro Sekunde. Bei der Ausführungsgeschwindigkeit von AFLNet ist zu beachten, dass der Fuzzer aufgrund der Fuzzing-Strategie in Kombination des bereits integrierten Monitoring des ZuP etwas langsamer ist. Die Geschwindigkeit des Fuzzers ist jedoch ausreichend, um eine Vielzahl von Eingaben zu generieren und an das ZuP zu senden. Außerdem anzumerken ist, dass AFLNet das ZuP immer wieder erneut startet. Das hat zur Folge, dass alle Zustände des ZuP immer wieder zurückgesetzt werden und somit die Initialisierung des Programms immer wieder erneut durchgeführt werden muss. Eine Möglichkeit, um das Fuzzing-Verhalten zu beschleu-

nigen, wäre einen Fuzzing-Harness für das Programm zu schreiben, welcher es ermöglichen würde den *persistent mode* von AFL zu verwenden. Mit ihm ist es möglich einen Zustand des ZuP statisch zu implementieren, welcher dem des Programms nach dem Initialisieren des ZuP entspricht. Diese Technik ist jedoch sehr aufwändig und benötigt sehr umfangreiches Wissen über die Funktionsweise des ZuP, welches nur durch Sourcecodeanalyse oder Reverse Engineering des bereits kompilierten Programms möglich ist. Dieser Fuzzing-Harnes kann unter umständen dazu führen, dass wichtige Programmpfade und Kontrollflüsse des ZuP vernachlässigt oder weggelassen werden können und die Wahrscheinlichkeit von false positives erhöht.
Der schnellste Fuzzer ist boofuzz mit durchschnittlich 77,88 Ausführungen pro Sekunde. Diese Anzahl von Ausführungen pro Sekunde ist auf die Implementierung des Fuzzers zurückzuführen. Boofuzz gehört zu der gattung der Batch-Mutation-Fuzzer und generiert eine Vielzahl von Eingaben auf einmal. Mithlife dieser Technik ist es möglich, eine Vielzahl von Eingaben zu generieren, welche jedoch nach Analyse der gesammelten Logs nach simplen Brute-Force angriffen aussehen.

Die Anzahl der Ausführungen pro Sekunde ist jedoch nicht ausschlaggebend für die Effektivität eines Fuzzers. Sie wird anhand der Anzahl der gefundenen Bugs pro Zeit gemessen. Die Anzahl der gefundenen Bugs beläuft sich bei AFLNet auf 3 Bugs, bei Pulsar und boofuzz auf keinen einzigen. Trotz einer niedrigen Anzahl an Ausführungen pro Sekunde konnte AFLNet bereits in sieben Minuten (vgl. Abb. 8) den ersten Bug finden. Dies spricht für eine hohe Effektivität des Fuzzing-Ansatzes von code-coverage und der Fuzzing-Strategie von AFLNet. Die Zunahme der Präzision von generierten validen Paketen spiegelt sich ebenso in der Grafik 4 wider.

Zusammenfassend ist in Hinsicht der Forschungsfrage *Q1* (siehe 1) festzustellen, dass AFLNet im Rahmen der vollzogenen Experimente der effektivste Fuzzer ist. Die Anzahl der gefundenen Bugs und die Effizienz der generierten Eingaben sprechen für eine hohe Effektivität des Fuzzers im Gegensatz zu den anderen Fuzzern. Im Hinblick auf die Ausführungsgeschwindigkeit ist boofuzz der schnellste Fuzzer, jedoch konnte er keine Bugs finden. Stellt man beide Fuzzer einander gegenüber, so beläuft sich die Ausführungsgeschwindigkeit von AFLNet auf durchschnittlich 8 Ausführungen pro Sekunde (siehe Abb. 6), während boofuzz auf durchschnittlich 77,88 Ausführungen pro Sekunde (siehe Abb. 12) kommt.
Auch in der Hinsicht der Effizienz der generierten Eingaben ist AFLNet der effizienteste Fuzzer. Die generierten Eingaben sind präzise und enthalten mit zunehmender Zeit der Kampagne weniger ungültige Eingaben (siehe Abb. 4). Dies ist jedoch nicht der Fall bei boofuzz,

welcher eine Vielzahl von Eingaben generiert, die nach Brute-Force-Angriffen aussehen und direkt beim Empfangen von dem MQTT-Broker verworfen werden 13.
Zu dem Fuzzer Pulsar konnten jedoch keine aufschlussreichen Ergebnisse erzielt werden.

6 Konklusion und zukünftige Arbeiten

Abschließend ist zu sagen, dass die Effektivität eines Fuzzers nicht anhand der Anzahl der generierten Eingaben pro Sekunde gemessen werden kann. Bei komplexeren Protokollen wie MQTT ist es wichtig, dass die generierten Eingaben valide sind und möglichst viele Zustände des ZuP erreichen. Ähnliche Schlussfolgerungen wurden bereits in der Arbeit von Holz et. al. in *„SoK: Prudent Evaluation Practices for Fuzzing“* [24] gezogen. Bei einem Fuzzer wie boofuzz, welcher eine Vielzahl von Eingaben generiert, ist es wichtig, dass die generierten Eingaben strenger definiert werden müssen und nicht nur auf dem Prinzip von Brute-Force basieren sollten. Im Falle von Pulsar ist es wichtig, dass der Fuzzer die Startsequenz des Protokolls korrekt erlernen kann, um ein Programm effektiv testen zu können. Auch wenn Pulsar mithilfe vieler Testdaten die State Machine des Protokolls erlernen kann, ist es wichtig auf den gesendeten Netzwerkverkehr zu achten. Geeignete Testdaten sollten so generiert werden, dass möglichst viele Zustände des ZuP erreicht werden und möglichst viele valide Eingaben generiert werden, welche der Fuzzer Pulsar als Anhaltspunkt zum Erlernen valider Eingaben benötigt.

Trotz der Tatsache, dass AFLNet in der Lage war, einige Fehler in der manipulierten Implementierung von MQTT zu finden, war der Fuzzer nicht in der Lage, alle Fehler zu finden. Dies zeigt, dass die Implementierung von Protokollen und Netzwerkdiensten weiterhin fehleranfällig ist und dass Fuzzing Kampagnen auf einen größeren Zeitraum mit mehreren Iterationen durchgeführt werden sollten, um komplexere Schwachstellen zu finden. Die Schwachstellen, die von AFLNet nicht gefunden wurden belaufen sich auf den Buffer Overflow in der Verarbeitung des Verbindungsaufbaus mit dem Broker, den Off-by-One Fehler in der Verarbeitung von Publish Nachrichten und den Use-After-Free Fehler in der Verarbeitung von Subscribe Nachrichten. Die Forschungsfrage *Q2* (siehe Aufzählung 1) konnte somit nur teilweise beantwortet werden, da sowohl boofuzz als auch Pulsar keine potenziellen Schwachstellen entdecken konnten und AFLNet nur drei von sechs entdeckt hat. Für bessere Ergebnisse müssen die Eingaben und die Implementierungen der Kampagnen weiter untersucht werden und weitere Experimente durchgeführt werden.
Zudem wurde mit der Rechnung in 5.2 eine Möglichkeit gegeben die Anzahl der benötigten Zeit der Generierung eines validen Pakets gegeben. Aufgrund der gezeigten Berechnung ist es ebenso ratsam mehrere Instanzen des boofuzz Fuzzers zu starten, die bestimmte Felder eines MQTT-Pakets zu fuzzen und jeder Fuzzer-Instanz ein genau definiertes Feld zuzuweisen.

In Zukunft sollte die Fuzzing Kampagne mit Pulsar weitergeführt werden. Hierbei soll-

te darauf geachtet werden, dass fürs Erste die Startsequenz des Protokolls korrekt erlernt wird, sodass Pulsar einen validen Startpunkt für eine effektivere Fuzzing Kampagne hat. Außerdem werden in Zukunft weitere Experimente und Fuzzing Kampagnen mit boofuzz durchgeführt, um die Effektivität des Fuzzers besser zu verstehen und zu testen. Auf lange Frist soll das automatisierte Testen und Lernen von Protokollen ohne jedwede menschliche Interaktion und Vorkenntnisse möglich sein. Eine eigene Implementierung einer kontinuierlich lernenden Pipeline für das automatisierte Testen von Protokollen wäre ein monumentales Ziel, welches in Zukunft erreicht werden sollte. Diese Pipeline sollte in der Lage sein, Protokolle anhand von bspw. Brute-Force ansätzen zu testen und zu lernen, um daraufhin valide Eingaben zu generieren und somit die State Machine des Protokolls möglichst genau approximieren zu können. Diese Entwicklungen könnten in Zukunft dazu beitragen, dass Protokolle effektiver und sicherer implementiert werden können und somit die Sicherheit von Netzwerken und Systemen erhöht wird.

Anhang A Listings

Listing 1: Sanity-Checks des Brokers nach erhalten einer Publish Nachricht in der Funktion `handle__publish` aus `srchandle_publish.c`

```
FIXME Intentionally commented out the sanity check for the payload
   ↪ length to be able to trigger buffer overflow
if(msg->payloadlen){
   if(db.config->message_size_limit && msg->payloadlen > db.config->
      ↪ message_size_limit){
      log__printf(NULL, MOSQ_LOG_DEBUG, "Dropped␣too␣large␣PUBLISH␣
         ↪ from␣%s␣(d%d,␣q%d,␣r%d,␣m%d,␣'%s',␣...␣(%ld␣bytes))",
               context->id, dup, msg->qos, msg->retain, msg->
                  ↪ source_mid, msg->topic, (long)msg->payloadlen);
      reason_code = MQTT_RC_PACKET_TOO_LARGE;
      goto process_bad_message;
   }
   msg->payload = mosquitto__malloc(msg->payloadlen+1);
   if(msg->payload == NULL){
      db__msg_store_free(msg);
      return MOSQ_ERR_NOMEM;
   }
   /* Ensure payload is always zero terminated, this is the reason
      ↪ for the extra byte above */
   ((uint8_t *)msg->payload)[msg->payloadlen] = 0;

   if(packet__read_bytes(&context->in_packet, msg->payload, msg->
      ↪ payloadlen)){
      db__msg_store_free(msg);
      return MOSQ_ERR_MALFORMED_PACKET;
   }
}
```

Listing 2: Buffer Overflow in der Funktion `handle__connect` der Quellcodedatei `src/handle_connect.c`

```
  int connect__on_authorized(struct mosquitto *context, void *
     ↪ auth_data_out, uint16_t auth_data_out_len){
      [...]
    if(context->will) {
      log__printf(NULL, MOSQ_LOG_DEBUG, "Will␣message␣specified␣(%ld␣
         ↪ bytes)␣(r%d,␣q%d).",
           (long)context->will->msg.payloadlen,
           context->will->msg.retain,
           context->will->msg.qos);
    // FIXME: Intentional buffer overflow
           char vuln_buffer[512];
```

```
            memcpy(vuln_buffer, context->will, context->will->msg.
               ↪ payloadlen);

            log__printf(NULL, MOSQ_LOG_DEBUG, "\t%s", context->will->
               ↪ msg.topic);
     }
       [...]
   }
```

Listing 3: Buffer Overflow in der Funktion `handle__publish` in der Quelldatei `src/handle_publish.c` des Mosquitto Brokers

```
int handle__publish(struct mosquitto *context){
    [...]
    rc = mosquitto_acl_check(context, msg->topic, msg->payloadlen, msg->
       ↪ payload, msg->qos, msg->retain, MOSQ_ACL_WRITE);
        if(rc == MOSQ_ERR_ACL_DENIED){
           log__printf(NULL, MOSQ_LOG_DEBUG,
                 "Denied␣PUBLISH␣from␣%s␣(d%d,␣q%d,␣r%d,␣m%d,␣'%s',␣...␣
                    ↪ (%ld␣bytes))",
                 context->id, dup, msg->qos, msg->retain, msg->source_mid
                    ↪ , msg->topic,
                 (long)msg->payloadlen);
           reason_code = MQTT_RC_NOT_AUTHORIZED;
           goto process_bad_message;
        }else if(rc != MOSQ_ERR_SUCCESS){
    // FIXME intentional buffer overflow
            memcpy(vuln_buffer, msg->payload, msg->payloadlen);
           db__msg_store_free(msg);
           return rc;
        }
    [...]
}
```

Listing 4: Memory Leak in der Funktion `connect__on_authorised` der Quellcodedatei `src/handle_subscribe.c` des Mosquitto Brokers

```
    int handle__on_authorised(struct mosquitto *context){
        [...]
        if(!slen){
                log__printf(NULL, MOSQ_LOG_INFO,
                      "Empty␣subscription␣string␣from␣%s,␣disconnecting.
                         ↪ ",
                      context->address);
                mosquitto__free(sub);
                mosquitto__free(payload);
                return MOSQ_ERR_MALFORMED_PACKET;
             }
```

```
            if(mosquitto_sub_topic_check(sub)){
               log__printf(NULL, MOSQ_LOG_INFO,
                     "Invalid␣subscription␣string␣from␣%s,␣
                        ↪ disconnecting.",
                     context->address);
               mosquitto__free(sub);
               mosquitto__free(payload);
   // FIXME - this is a memory leak use after free
               strcpy(sub, "Use␣After␣free");
               return MOSQ_ERR_MALFORMED_PACKET;
            }
        [...]
   }
```

Literatur

[1] OWASP. *Static Code Analysis*. URL: https://owasp.org/www-community/controls/Static_Code_Analysis (besucht am 30. 08. 2024).

[2] science direct. *Black-Box Testing*. URL: https://www.sciencedirect.com/topics/computer-science/black-box-testing (besucht am 30. 08. 2024).

[3] André C. Coulter. „Graybox Software Testing in Real-Time in the Real World“. In: 2001. URL: https://api.semanticscholar.org/CorpusID:29193682.

[4] OWASP Foundation. *Fuzzing*. URL: https://owasp.org/www-community/Fuzzing (besucht am 11. 03. 2024).

[5] Maialen Eceiza, 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* 8.13 (2021), S. 10390–10411. DOI: 10.1109/JIOT.2021.3056179. URL: https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=9344712 (besucht am 11. 03. 2024).

[6] Hongliang Liang. *Fuzzing: State of the Art*. URL: https://wcventure.github.io/FuzzingPaper/Paper/TRel18_Fuzzing.pdf (besucht am 30. 08. 2024).

[7] Sutton Michael und Greene Adam und Amini Pedram. *Fuzzing: Brute Force Vulnerability Discovery*. 2007.

[8] Michal Zalewski. *Instrumentieren eines Programms mit AFL*. URL: https://github.com/google/AFL/blob/master/docs/technical_details.txt#L23-L76 (besucht am 11. 03. 2024).

[9] Li Jun und Zhao Bodong und Zhang Chao. „Fuzzing: a survey“. In: *Cybersecurity* 1.1 (Juni 2018), S. 6. ISSN: 2523-3246. DOI: 10.1186/s42400-018-0002-y. URL: https://doi.org/10.1186/s42400-018-0002-y (besucht am 30. 08. 2024).

[10] Chen Yuqi und Poskitt Christopher M. und Sun Jun und Adepu Sridhar und Zhang Fan. „Learning-Guided Network Fuzzing for Testing Cyber-Physical System Defences“. In: *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2019, S. 962–973. DOI: 10.1109/ASE.2019.00093. URL: https://arxiv.org/pdf/1909.05410 (besucht am 30. 08. 2024).

[11] Böhme Marcel und Pham Van-Thuan und Nguyen Manh-Dung und Roychoudhury Abhik. „Directed Greybox Fuzzing“. In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. CCS '17. Dallas, Texas, USA: Association for Computing Machinery, 2017, S. 2329–2344. ISBN: 9781450349468. DOI: 10.1145/3133956.3134020. URL: https://mboehme.github.io/paper/CCS17.pdf.

[12] A.L. Russell. *'Rough Consensus and Running Code' and the Internet-OSI Standards War*. 2006. DOI: 10.1109/MAHC.2006.42.

[13] *TCP Protocol Specification RFC 793*. URL: https://www.rfc-editor.org/rfc/rfc793#section-3.4 (besucht am 01. 09. 2024).

[14] kernel.org. *man tcp(7)*. URL: https://www.man7.org/linux/man-pages/man7/tcp.7.html (besucht am 01. 09. 2024).

[15] OASIS. *MQTT Version 3.1.1 Spezifikation*. URL: https://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html (besucht am 01. 09. 2024).

[16] MQTT. *MQTT man page*. URL: https://mosquitto.org/man/mqtt-7.html (besucht am 01. 09. 2024).

[17] Thaun Pham. *AFLNet Repository*. URL: https://github.com/aflnet/aflnet (besucht am 11. 09. 2024).

[18] Van-Thuan Pham, Marcel Böhme und Abhik Roychoudhury. „AFLNet: A Greybox Fuzzer for Network Protocols“. In: *Proceedings of the 13rd IEEE International Conference on Software Testing, Verification and Validation : Testing Tools Track*. 2020.

[19] Gascon Hugo und Wressnegger Christian und Yamaguchi Fabian und Arp Daniel und Rieck Konrad. „Pulsar: Stateful Black-Box Fuzzing of Proprietary Network Protocols“. In: *Security and Privacy in Communication Networks*. Hrsg. von Thuraisingham Bhavani und Wang XiaoFeng und Yegneswaran Vinod. Cham: Springer International Publishing, 2015, S. 330–347. ISBN: 978-3-319-28865-9.

[20] Maialen Eceiza, Jose Luis Flores und Mikel Iturbe. „Improving fuzzing assessment methods through the analysis of metrics and experimental conditions“. In: *Computers & Security* 124 (Jan. 2023), S. 102946. ISSN: 0167-4048. DOI: 10.1016/j.cose.2022.102946. URL: https://www.sciencedirect.com/science/article/pii/S0167404822003388 (besucht am 07. 09. 2024).

[21] Eisele Max u. a. „Embedded fuzzing: a review of challenges, tools, and solutions“. In: *Cybersecurity* 5.1 (Sep. 2022), S. 18. ISSN: 2523-3246. DOI: 10.1186/s42400-022-00123-y. URL: https://doi.org/10.1186/s42400-022-00123-y (besucht am 27. 05. 2024).

[22] Jiongyi Chen u. a. „IoTFuzzer: Discovering Memory Corruptions in IoT Through App-based Fuzzing“. In: *Network and Distributed System Security Symposium*. 2018. URL: https://api.semanticscholar.org/CorpusID:158965.

[23] Andrea Fioraldi u. a. „LibAFL: A Framework to Build Modular and Reusable Fuzzers“. In: *Proceedings of the 29th ACM conference on Computer and communications security (CCS)*. CCS '22. Los Angeles, U.S.A.: ACM, Nov. 2022. URL: https://dl.acm.org/doi/pdf/10.1145/3548606.3560602 (besucht am 11. 03. 2024).

[24] M. Schloegel u. a. „SoK: Prudent Evaluation Practices for Fuzzing“. In: *2024 IEEE Symposium on Security and Privacy (SP)*. Los Alamitos, CA, USA: IEEE Computer Society, Mai 2024, S. 1974–1993. DOI: 10.1109/SP54263.2024.00137. URL: https://doi.ieeecomputersociety.org/10.1109/SP54263.2024.00137.

[25] Shihao Jiang u. a. *A Survey of Network Protocol Fuzzing: Model, Techniques and Directions*. en. arXiv:2402.17394 [cs]. Feb. 2024. URL: http://arxiv.org/abs/2402.17394 (besucht am 27. 05. 2024).

[26] Andrea Fioraldi u. a. „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/system/files/woot20-paper-fioraldi.pdf (besucht am 11. 03. 2024).

[27] Philipp Görz und Björn Mathis und Keno Hassler und Emre Güler und Thorsten Holz und Andreas Zeller und Rahul Gopinath. „Systematic Assessment of Fuzzers using Mutation Analysis“. In: *32nd USENIX Security Symposium (USENIX Security 23)*. Anaheim, CA: USENIX Association, Aug. 2023, S. 4535–4552. ISBN: 978-1-939133-37-3. URL: https://www.usenix.org/conference/usenixsecurity23/presentation/gorz.

[28] Javad Garshasbi und Mehdi Teimouri. „CNNPRE: A CNN-Based Protocol Reverse Engineering Method“. In: *IEEE Access* 11 (2023). Conference Name: IEEE Access, S. 116255–116268. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2023.3325391. URL: https://ieeexplore.ieee.org/document/10287339/?arnumber=10287339 (besucht am 08. 09. 2024).

[29] Google. *FuzzBench: Fuzzer Benchmarking as a Service*. URL: https://github.com/google/FuzzBench.

[30] Brendan Dolan-Gavitt u. a. „LAVA: Large-Scale Automated Vulnerability Addition“. In: *2016 IEEE Symposium on Security and Privacy (SP)*. ISSN: 2375-1207. Mai 2016, S. 110–121. DOI: 10.1109/SP.2016.15. URL: https://ieeexplore.ieee.org/document/7546498 (besucht am 08. 09. 2024).

[31] llvm.org. *AddressSanitizer*. 6. Sep. 2024. URL: https://clang.llvm.org/docs/AddressSanitizer.html.

[32] Sebastian Pahl. *Docker Container für Pulsar*. URL: https://github.com/ItsMagick/pulsar (besucht am 01. 09. 2024).

[33] Sebastian Peschke. *Dockerfile für das Kompilieren von AFLNet und MQTT*. URL: https://github.com/ItsMagick/Fuzzing_Benchmarking/blob/main/Dockerfile (besucht am 02. 09. 2024).

[34] Sebastian Peschke. *llvm patch*. URL: https://github.com/ItsMagick/Fuzzing_Benchmarking/blob/main/afl-llvm-pass.patch (besucht am 02. 09. 2024).

[35] Sebastian Peschke. *Preload Bibliothek für MQTT*. URL: https://github.com/ItsMagick/Fuzzing_Benchmarking/blob/main/scripts/std_out_redirect.c (besucht am 02. 09. 2024).

[36] Sebastian Peschke. URL: https://github.com/ItsMagick/Fuzzing_Benchmarking/blob/main/scripts/create_aflnet_inputs.py (besucht am 02. 09. 2024).

[37] Sebastian Peschke. *Skript zur Extraktion der Abstürze aus den AFLNet stdout logs*. URL: https://github.com/ItsMagick/Fuzzing_Benchmarking/blob/main/scripts/document_crashes.sh (besucht am 02. 09. 2024).

[38] Sebastian Peschke. URL: https://github.com/ItsMagick/Fuzzing_Benchmarking/blob/main/scripts/test_crashes.sh (besucht am 02. 09. 2024).

[39] OASIS MQTT. *Valide Steuerpakettypen für MQTT Fixed Headers*. URL: https://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html#_Toc398718021 (besucht am 10. 09. 2024).

[40] OASIS MQTT. *Struktur des MQTT Variable Headers eines Connect Pakets*. URL: https://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html#_Toc398718030 (besucht am 10. 09. 2024).

[41] Sebastian Peschke. *Skript zur generierung von Eingaben für das MQTT Protokoll*. URL: https://github.com/ItsMagick/Fuzzing_Benchmarking/blob/main/scripts/create_aflnet_inputs.py (besucht am 02. 09. 2024).

[42] Sebastian Peschke. *Script zum Extrahieren des Netzwerkverkehrs*. URL: https://github.com/ItsMagick/Fuzzing_Benchmarking/blob/main/scripts/extract_traffic.sh (besucht am 02. 09. 2024).

[43] Sebastian Peschke. *Skript zum Ausführen von Pulsar*. URL: https://github.com/ItsMagick/pulsar/blob/ff1852d82fa61acc67aff6cd42d6f63d1f5bdca5/run (besucht am 02. 09. 2024).

Erklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe; die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht. Zu den nicht trivialen eingesetzten Hilfsmitteln zählen DeepL Write als Grammatik- und Rechtschreibprüfer. Die Arbeit wurde nach meiner besten Kenntnis bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

Hof, den 11.09.2024

Sebastian Peschke

Abstract

Die Bachelorarbeit *Auswertung von Netzwerkfuzzern am Beispiel des MQTT Protokolls* untersucht die Effektivität und Leistungsfähigkeit von drei spezifischen Netzwerkfuzzern—Pulsar, AFLNet und boofuzz—im Kontext des MQTT-Protokolls, das weit verbreitet in IoT- und M2M-Anwendungen zum Einsatz kommt. Netzwerkfuzzing ist eine Methode des Softwaretestens, die darauf abzielt, Sicherheitslücken und Schwachstellen in Netzwerkprotokollen aufzudecken, indem gezielt fehlerhafte oder unerwartete Eingaben an ein System gesendet werden. Die Arbeit beginnt mit einer umfassenden Einführung in die Grundlagen des Softwaretestens und des Fuzzings, wobei verschiedene Testmethoden wie White-Box, Black-Box und Grey-Box Testing erläutert werden. Es folgt eine detaillierte Darstellung des MQTT-Protokolls und der spezifischen Anforderungen, die es an Fuzzing-Tools stellt.

Im Kern der Arbeit steht die Analyse der drei Fuzzer, die hinsichtlich ihrer Fähigkeit, Sicherheitslücken im MQTT-Broker Mosquitto zu identifizieren, verglichen werden. Dazu wurden verschiedene Metriken herangezogen, darunter die Anzahl gefundener Bugs, die Geschwindigkeit der Testfallgenerierung und die Reproduzierbarkeit der entdeckten Schwachstellen. AFLNet, ein Grey-Box-Fuzzer, der speziell für Netzwerkprotokolle entwickelt wurde, zeigte in der Untersuchung eine hohe Effizienz bei der Entdeckung von Schwachstellen, insbesondere bei der Analyse von Verbindungsfehlern und Buffer Overflows. Boofuzz, ein Black-Box-Fuzzer, der durch Zufall generierte Testfälle verwendet, zeigte eine geringere Effizienz, konnte jedoch durch seine einfache Handhabung und Flexibilität punkten. Pulsar, das sich durch einen feedback-orientierten Ansatz auszeichnet, bot eine ausgeglichene Performance zwischen Effektivität und Geschwindigkeit, erwies sich jedoch als weniger effektiv in der Reproduzierbarkeit von Bugs.

Die Arbeit schließt mit einem Vergleich der Stärken und Schwächen der getesteten Fuzzer und bietet Empfehlungen für deren Einsatz in der Praxis. Zudem werden Ansätze für zukünftige Arbeiten aufgezeigt, die darauf abzielen, die Fuzzing-Techniken weiter zu optimieren und ihre Anwendung in sicherheitskritischen Netzwerksystemen zu verbessern. Die Ergebnisse dieser Bachelorarbeit liefern wertvolle Erkenntnisse für die Weiterentwicklung von Fuzzing-Kampagnen und tragen zur Verbesserung der Sicherheitsüberprüfung von Netzwerkprotokollen wie MQTT bei.