Fuzzing von IoT Binarys für Ruhm und Ehre

Grey-box Fuzzing eines Netzwerkprotokolls eines Binarys mithilfe des American Fuzzy Lops

P r a x i s 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.: 00000000
lorem ipsum 123
456 Hof

Hof, 24. November 2024

Inhaltsverzeichnis

Abbildungsverzeichnis

Abkürzungsverzeichnis

AFL	American Fuzzy Lop
AFL++	American Fuzzy Lop plus plus
ARM	Advanced RISC Machines
BASH	Bourne Again Shell
BWRAP	Bubblewrap
CLI	Command Line Interface
CPU	Central Processing Unit
CVSS	Common Vulnerability Scoring System
DDoS	Distributed Denial of Service
ELF	Executable and Linkable Format
FS	File System
GCC	GNU Compiler Collection
GDB	GNU Debugger
GNU	GNU's Not Unix
HDD	Hard Disk Drive
ID	Identifier
IoT	Internet of Things
IP	Internet Protocol
kB	Kilobyte
NMAP	Network Mapper
PDF	Portable Document Format
QEMU	Quick Emulator
RAM	Random Access Memory
RCE	Remote Code Execution
SNS	System and Network Security
SSD	Solid State Drive
SSH	Secure Shell

TCP . . . . . Transmition Control Protocol

UNIX . . . . . Uniplexed Information Computing System

URL . . . . . Uniform Resource Locator

Listings

1 Einleitung

Immer mehr gewinnen smarte Geräte Bedeutung in unser aller Alltag. Die Verwendung solcher Geräte reicht von Glühlampen, welche sich an den Stromverbrauch des Endnutzers anpassen, bis hin zu einem Beamer, welcher vielseitig und alltäglich in Firmen, Freizeitanlagen und auch Bildungsinstituten zum Einsatz kommt. Gerade bei diesen Geräten sollte, aufgrund der vielseitigen Ausnutzungsmöglichkeiten für beispielsweise DDoS (Distributed Denial of Service) Angriffe, ein genaues Augenmerk auf Sicherheit gesetzt werden [1].
Um solchen Gefahren vorzubeugen, führt die Forschungsgruppe *Systems and Network Security* regelmäßige Schwachstellenscans durch. Die Scans werden mithilfe eines angepassten Abbildes des Greenbone [2] Schwachstellenscanners durchgeführt. Nach einem der Scans ist Pahl – einem wissenschaftlichen Mitarbeiter der Forschungsgruppe SNS (System and Network Security)[3] – auf eine bisher unentdeckte Schwachstelle im Web-Interface eines in der Hochschule eingesetzten IoT (Internet of Things)-Geräts gestoßen. Hierbei handelt es sich um eine Path Traversal [4] Schwachstelle in der URL (Uniform Resource Locator) des Web-Interfaces eines an der Hochschule verwendeten IoT-Geräts. Genauere Details zu der Arbeit können zum Zeitpunkt der Verfassung dieses Dokuments, aufgrund einer noch nicht abgeschlossenen *responsible disclosure*, nicht preisgegeben werden.

Genau dieser Problematik widmet sich diese Arbeit, in der in den folgenden Kapiteln darauf eingegangen wird, wie es zur Findung von Schwachstellen in der Firmware von IoT-Geräten kommt. Sie dient als Anleitung für das methodische Testen von Firmware. Die Herangehensweise hierbei ist das automatisierte Testen mithilfe eines Fuzzers, welcher durch Generierung von Input Schwachstellen in einem Programm finden soll [5]. In dieser Arbeit wird insbesondere auf die Verwendung des Fuzzers AFL++ (American Fuzzy Lop plus plus) eingegangen.

Zuerst sollen kurz technische Hintergründe erläutert werden, um ein gewisses Grundverständnis der zum Einsatz gebrachten Technologien aufzubauen. Hierbei wird das Grundlegende Setup und die zugrunde liegende Firmware erklärt und eine grobe Übersicht über AFL[6] gegeben. In dieser Arbeit wird überwiegend die Erweiterung des Fuzzers AFL++ verwendet. AFL wird nur als Referenz auf die grundlegende Funktionsweise des Fuzzers hergenommen.

Im Anschluss werden die genaueren Modi und Terminologien des AFL++ [7] geklärt, welche in dieser Arbeit Verwendung finden. Dazu gehört, wie AFL++ mit der zu untersuchenden Applikation interagiert und kommuniziert. Genauer eingegangen wird zudem auf das Fuzzing von Netzwerkapplikationen, welche den Tester vor Herausforderungen stellt, da diese

Funktionalität nicht für AFL++ implementiert ist. Ebenfalls werden die mit dem Fuzzing verbundenen Risiken angesprochen.

Daraufhin soll auf den Aufbau und die Implementierung der Testumgebung zur Analyse der Firmware für ein tieferes Verständnis der Testumgebung eingegangen werden. Hierbei wird auf die Funktionalitäten der Firmware eingegangen, welche einen Einblick in den Aufbau der Firmware und der Interaktion mit dem zugrunde liegenden Betriebssystem geben. Zudem wird erklärt, wie mit der Applikation interagiert werden kann.

Nachdem ein allgemeines Verständnis der verwendeten Technologien vermittelt wurde, wird die Durchführung des Fuzzing der Applikation beschrieben. Dieser Abschnitt ist vor allem für unerfahrene Tester und Schwachstellenforscher, denen hier ein Einblick in die Instrumentation und Modifikation zum Lesen von Dateien einer Applikation geboten wird. Zudem werden Möglichkeiten zur Optimierung des Fuzzingprozesses sowie deren Anwendung auf eine laufende Fuzzing Kampagne offengelegt.

Zum Abschluss sollen noch die besonderen Hindernisse bei der Durchführung des Fuzzings genauer betrachtet werden. Das soll vor allem bei der Umsetzung späterer Projekte dabei verhelfen, da hierbei auch auf allgemeine Problemstellungen eingegangen wird. Anschließend daran soll ein Ausblick zu alternativen Umsetzungen des Fuzzings gegeben werden. Abschließend soll das Projekt in seiner Gesamtheit reflektiert werden.

1.1 Motivation

Diese Arbeit soll als Anleitung zum Entwickeln einer Fuzzing-Kampagne mit AFL dienen. Mit der immer weiter ansteigenden Nutzung ovn IoT-Geräten im Alltag muss in dem Bereich der Netzwerksicherheit von untereinander verbundenen Geräten ein besonderes Augenmerk geworfen werden. Die derzeitige Schätzung der aktuell mit dem Internet verbundenen IoT-Geräte liegt bei ca. 3,5 Milliarden Geräten im Jahr 2023 [8].
Die derzeit führende Sprache für die Entwicklung der Firmware solcher Geräte ist die Programmiersprache C. Auch trotz bereits existierender Frameworks und Programmiersprachen, wie Rust oder Go wird die Programmiersprache C oft den anderen Optionen vorgezogen und zählt somit zu einer der meistverwendeten Programmiersprachen [9]. Das ist der Leichtgewichtigkeit und vor allem Geschwindigkeit der Programmiersprache geschuldet. Diese zwei Faktoren spielen eine tragende Rolle bei der Implementierung von Software für IoT-Geräte aufgrund der sehr limitierten Ressourcen, die solche Geräte mit sich bringen. Bei

der Auswahl der passenden Programmiersprache werden jedoch wichtige Kriterien, wie die Sicherheit einer Programmiersprache weniger in Erwägung gezogen. In Programmen, die in der Programmiersprache C geschrieben wurden, werden die häufigsten schwerwiegenden Schwachstellen mit einer Bewertung eines CVSS (Common Vulnerability Scoring System) Score von mindestens 7 gefunden. Dabei ist diese Programmiersprache für über 50 % aller öffentlich dokumentierten Schwachstellen verantwortlich [10]. Das CVSS ist ein Bewertungssystem, das die Schwere von Sicherheitslücken kategorisiert. Die dabei verwendete Skala der CVSS Version 3.0 ordnet hierbei Schwachstellen mit einem Score von 0.0 als gering, 0.1-3.9 als niedrig, 4.0-6.9 als medium, 7.0-8.9 als hoch und 9.0-10.0 als kritisch ein [11]. Als kritisch eingestufte Schwachstellen sind in der Regel Schwachstellen, welche es einem Angreifer erlauben, RCE (Remote Code Execution) auszuführen und somit die Möglichkeit besteht, unbefugt Informationen und Daten über ein System zu erlangen. Zu der Klasse der meistverbreiteten Schwachstellen dieser Art gehören Memory Corruption Schwachstellen, welche es ermöglichen auf unbefugten Speicher eines Systems zugreifen zu können. Genau diese Schwachstellen beispielsweise in Form eines Buffer Overflows gehören zu den meist gefundenen Schwachstellen in Programmen, welche auf der Programmiersprache C basieren. Aufgrund der weiten Verbreitung und gewinnenden Bedeutung von IoT-Geräten im alltäglichen Leben soll diese Arbeit eine Möglichkeit darbieten, mithilfe von Fuzzing automatisiert Schwachstellen in diesen Geräten zu finden. Mithilfe solcher Methodiken soll in Zukunft die Ausnutzung schwerwiegender Schwachstellen vorgebeugt werden.

Das Fuzzing von Netzwerkapplikationen ist derzeit in den Startlöchern und eine weitaus unerforschte Disziplin. Dabei zu beachten ist, dass die Performance der Protokoll-Fuzzer weitaus unter der, der Applikations-Fuzzer liegt. Somit bietet diese Arbeit einen alternativen Ansatz zum Fuzzen von Netzwerkprotokollen auf Applikationsebene, um diesem Performanceverlust entgegenzuwirken.

1.2 Zielsetzung

Wie bereits erwähnt, soll diese Arbeit eine Möglichkeit offenlegen, wie ein Netzwerkprotokoll auf Applikationsebene gefuzzt werden kann, sodass die Leistung des Fuzzers nicht darunter leidet. Das bedeutet nicht nur, dass hier ein Weg des Protokoll-Fuzzings beschrieben wird, sondern auch die generelle Herangehensweise an eine Fuzzing-Kampagne und die tieferen Hintergründe des AFL Fuzzers genauer erläutert werden.

Im Laufe dieser Arbeit soll die Vorgehensweise, mit der die Fuzzing-Kampagne umge-

setzt wurde, dargestellt werden. Des Weiteren werden verwendete Technologien genau erklärt und deren Rolle im Fuzzing-Prozess beleuchtet werden. Dabei soll vor allem auf die Funktionsweise und die Interaktion von den verwendeten Technologien auf die Applikation zum allgemeinen Verständnis der Applikation eingegangen werden. Sobald die Technologien ausreichend erklärt wurden, folgt das Eingehen auf die genauere Implementierung und Instrumentierung der zu untersuchenden Applikation. Dieser Teil dient als Dokumentation zur tatsächlichen Umsetzung einer Fuzzing-Kampagne.

2 Hintergrund

Es gibt verschiedene Mittel und Wege, eine Applikation zu fuzzen. Man muss sich jedoch im Klaren darüber sein, um welche Art von Fuzzing es sich handelt und welche Technologien verwendet werden können. Bei der Auswahl der Technologien muss man sich zum einen Gedanken darüber machen, welche Testumgebungen zum Einsatz kommen, welche Komponenten analysiert werden sollen. Zum anderen muss man wissen, wie die Analyse durchgeführt werden soll. Da die Alternativen der verwendeten Technologien in einer zu einem späteren Zeitpunkt geschriebenen Arbeit genauer erklärt werden, werden in dieser Arbeit nur die verwendeten Technologien und die dazugehörigen Terminologien geklärt.

2.1 Internet of Things Gerätesicherheit

IoT kann wie folgt definiert werden:
"Das Internet of Things ist eine Gruppe von Infrastrukturen, welche verbundene Objekte miteinander verbindet und somit das Verwalten, Data-Mining und die Verfügbarkeit der von ihnen generierten Daten stellen."[vgl. 12, S. 2]

Das zu untersuchende Gerät erfüllt die Charakteristik der Interkonnektivität, Stellung von Daten und der Sensorik. Umgesetzt werden diese Merkmale mittels eines TCP/ IP-Stack, mit dem dieses Gerät gesteuert werden kann. Oftmals wird die hohe Verfügbarkeit und Bedienbarkeit zur erleichterten Bedienung und Automatisierung des Alltags gewünscht. Diese Funktionalitäten stehen jedoch oftmals im Kontrast zur Gerätesicherheit von Infrastrukturen und der Integrität der darin enthaltenen Daten.

IoT-Geräte sind im Bestfall in einem separaten Netzwerk mit einer sehr restriktiven Konfiguration. Von diesem Szenario ist jedoch nicht immer auszugehen, da gerade kleine und mittelständische Unternehmen und Institutionen nicht immer das Know-how zu einer sauberen Trennung von Netzwerken mit sich bringen. In solchen Szenarien können gerade Geräte mit Protokollen, mit denen es möglich ist, Geräte fernzusteuern, verheerende Folgen auf die Gesundheit eines Netzwerkes haben. Dieser Protokolle kann sich ein erfahrener Angreifer zu eigen machen, um unbefugten Zugang zur Infrastruktur eines Netzwerks zu erlangen.

Um die Gerätesicherheit solcher Geräte weiter zu stärken und den Einfallswinkel potenzieller Schwachstellen möglichst gering zu halten, wird der TCP-IP-Stack genauer analysiert.

2.2 Reconnaissance

Um ein genaueres Bild des IoT-Geräts zu erhalten, wurden Informationen über das System gesammelt. Dieser Prozess wird in der Cyber Security auch als *Reconnaissance* bezeichnet. Reconnaissance kann in die zwei Bereiche *aktive* und *passive* Reconnaissance eingeteilt werden. Aktive Reconnaissance beschäftigt sich dabei hauptsächlich mit der direkten Interaktion mit einem System. Bei der passiven Reconnaissance hingegen beschäftigt man sich mit der Recherche ohne Interaktion mit dem System. Dieses Kapitel macht Gebrauch von beiden Aspekten dieses Konzepts.

Die im Kapitel 1 beschriebene Path-Traversal-Schwachstelle resultierte darin, dass ein Snapshot des IoT- Geräts und somit auch der Firmware gemacht werden konnte. Die Firmware beinhaltet ein Hauptbinary mit dem Namen *mmapp*, das für alle Benutzerfunktionen des Geräts verantwortlich ist. Bei diesem Binary handelt es sich um ein dynamisch gelinktes Closed-Source-ELF (Executable and Linkable Format)-Binary in einer 32-Bit-Architektur. Die Plattform, für die das Binary kompiliert wurde, ist eine ARM (Advanced RISC Machines)v7 Plattform.

Listing 1: Zeigt die Ausgabe des `file` Befehls auf das Hauptbinary mmapp. `file` ist ein Programm, welches versucht eine gegebende Datei zu klassifizieren.

```
$ file mmapp
mmapp: ELF 32-bit LSB executable, ARM, EABI5 version 1 (SYSV),
   ↪ dynamically linked, interpreter /lib/ld-linux-armhf.so.3, for GNU
   ↪ /Linux 2.6.16, BuildID[sha1]=618
   ↪ c3b2ab4868d6d25480a6c1c9d88498c4d7379, stripped
```

Da das Binary dynamisch gelinkt ist, ist davon auszugehen, dass es von Bibliotheken außerhalb des Binarys abhängig ist. Während der Laufzeit des Programms werden die benötigten Bibliotheken in den Adressraum des Programms geladen. In Listing 13 sind die Bibliotheken aufgeführt, die von mmapp verwendet werden. Ein Portscan mit NMAP (Network Mapper) zeigt offene TCP-Ports.

Listing 2: Zeigt die Ausgabe des Portscans des Geräts mit nmap. Die hier verwendeten Flags `-sV -p-` bezwecken die Ausgabe der Version der Services aller Ports.

```
$ sudo nmap -sV -p- 10.0.0.10
Starting Nmap 7.93 (https://nmap.org) at 2023-10-31 14:31
   ↪ CET
Nmap scan report for 10.0.0.10
Host is up (0.00030s latency).
Not shown: 65529 closed tcp ports (reset)
PORT       STATE  SERVICE      VERSION
22/tcp     open   ssh          Dropbear sshd 2022.83
80/tcp     open   http         Node.js Express framework
3060/tcp   open   http         Node.js
4352/tcp   open   pjlink       PJLink projector control
7142/tcp   open   unknown
41794/tcp  open   tcpwrapped
```

Die Ports *80* und *3060* verweisen auf ein öffentlich zugängliches Webinterface. Zusätzlich befindet sich auf dem Port *4352* ein textbasiertes Protokoll namens *pjlink*. Der für diese Arbeit relevante Port ist heit der Port mit der Nummer *7142*. Laut der Dokumentation des Herstellers [13], existiert ein Netzwerkprotokoll zur Fernsteuerung des Geräts auf dem offenen TCP-Port 7142. Dieses Protokoll erwartet Bytes als Input. Für die Kommunikation mit dem Gerät über TCP wird ein Python-Skript [14] verwendet, das zunächst nur die in der Dokumentation festgehaltenen Befehle enthält. Die nach dem Ausführen des Scripts erhaltenen Antworten werden in einer Tabelle [15] festgehalten.
Das Senden der Befehle erfordern das Mitgeben einer Checksumme. Die Checksumme berechnet sich aus der Summer aller im Befehl enthaltenen Bytes. Im Anschluss wird nur das letzte Byte betrachtet, das der mitzusendenden Checksumme entspricht.

2.3 Einführung in AFL

AFL ist ein Werkzeug zur automatisierten Programmprüfung. Es wurde 2013 von Michal Zalewski entwickelt und basiert ursprünglich auf dem Ansatz des Sourcecode-guided Fuzzing. Dieser Ansatz schreibt vor, dass auf jeden bedingten Sprung, wie beispielsweise einer if-Anweisung in vielen Hochsprachen wie C oder Java, geachtet wird. Ein solcher Verzweigungspunkt, an dem der Zustand des Programms verändert werden kann, wird als *basic block* bezeichnet.

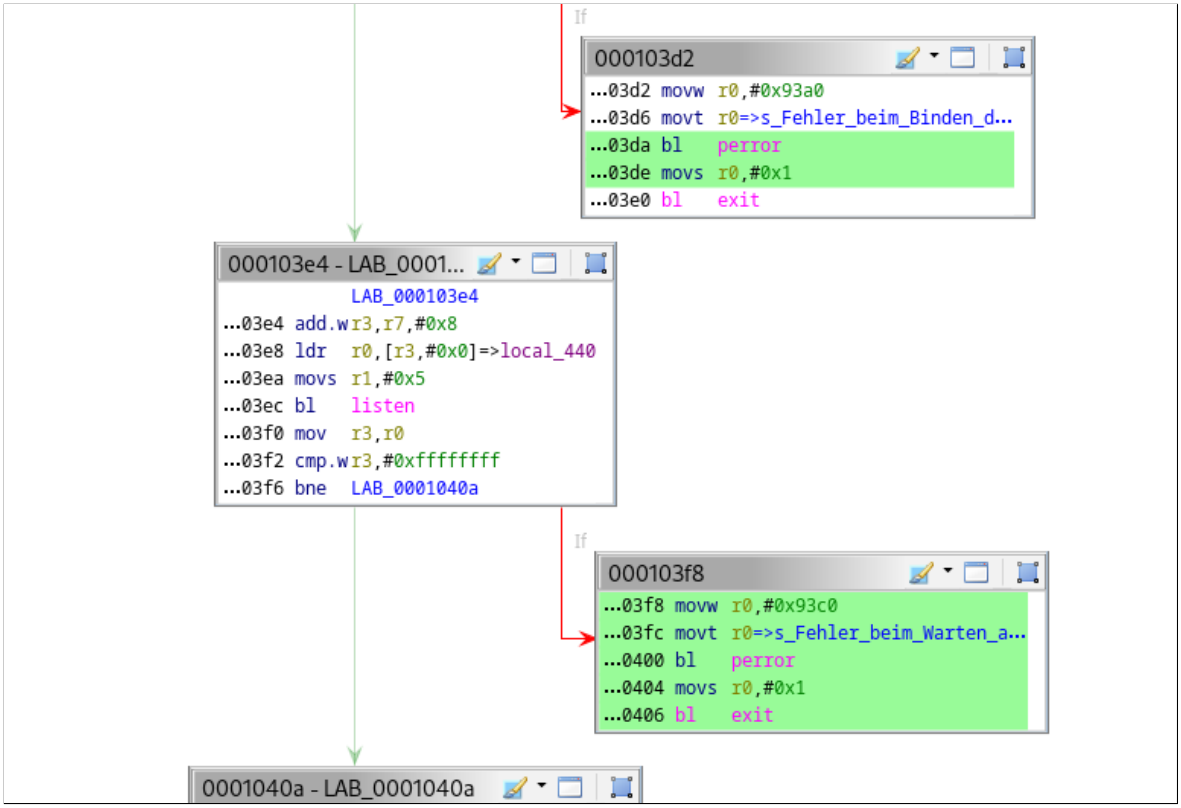

Abbildung 1: Zeigt ein Beispiel mehrerer basic blocks. Der hier gezeigte Disassembly und der damit verbundene Kontrollflussgraph entstammt dem selbst implementierten TCP-Servers aus dem Kapitel 5.3.2. Verwendet wurde hierbei das Reverse-Engineering-Tool Ghidra.

Ein basic block ist dadurch gekennzeichnet, dass die erste Instruktion als Einstiegspunkt (*entry point*) und die letzte Instruktion als Ausstiegspunkt (*exit point*) bezeichnet wird. Ein entry point ist in der Regel die erste Instruktion nach einem Sprung im Codesegment. Der exit point kann das Ende des Codes oder eine Sprunginstruktion im Codesegment sein. Die Verbindung zweier basic blocks (in Abbildung 1 mit Pfeilen dargestellt) wird als *branch edge* bezeichnet. in dieser Arbeit wird ein Pfad von mehreren miteinander verbundenen basic blocks als Codepfad bezeichnet.

Der Programmcode wird klassischerweise zuerst mit dem von AFL bereitgestellten custom compiler `afl-cc` kompiliert. In diesem Schritt wird das Programm von AFL auf bedingte Sprünge untersucht und an jeder Instruktion wird eigener Programmcode an diesen Stellen injiziert. Anschließend wird der dabei entstandene Programmcode kompiliert und ist somit für die Fuzzing-Kampagne vorbereitet.

Zur Optimierung der Fuzzing-Kampagne kann der Korpus nach dem ersten Durchlauf mit einem weiteren Tool von AFL (`afl-cmin`) minimiert werden. Bei der Minimierung des Korpus wird darauf geachtet, ob beim Ausführen des Programms neue Programmpfade traversiert werden. Danach wird die Fuzzing-Kampagne mit dem Tool `afl-fuzz` gestartet. Dabei werden in einem Testdurchlauf zuerst alle gesammelten Inputs auf ihre Validität geprüft, indem AFL darauf achtet, ob das zu untersuchende Programm in einen definierten Zustand gelangt und fehlerfrei terminiert. Zusätzlich wird geprüft, welche Programmpfade abgelaufen werden und ob Testfälle neue Programmpfade erreichen. Diese Strategie bezeichnet man als covered-based Strategie [vgl. 16, S. 7].

Nachdem alle Testfälle erfolgreich durchgeführt wurden, beginnt die Mutation des Inputs. Hierzu gibt es auch verschiedene Ansätze, die von Fuzzern verfolgt werden.
Ein elementarer Bestandteil des AFL ist der darin enthaltene Forkserver. Dieser ist für die Vorbereitung des Binaries für die Fuzzing-Kampagne verantwortlich. Das Binary durchläuft den Syscall `execve()` einmalig, welcher dafür verantwortlich ist, das zu startende Programm in den aktuellen Prozess zu laden. Zudem ist der Forkserver dafür verantwortlich sicherzustellen, dass das zu instrumentierende Programm nur ein Mal zum Ausführen gelinkt wird und ein Einsprungspunkt für den Fuzzer nach der Initialisierung gesetzt wird. An diesem Einsprungspunkt werden nach der Initialisierung die Instruktionen – oder auch Testcases – der Fuzzing Kampagne von AFL injiziert. Dadurch wird die Performance verbessert, da das Programm bei mehrfacher Ausführung nur jeweils ein Mal initialisiert wird und der ursprüngliche Zustand des Prozesses wiederhergestellt wird. Anschließend werden Kopien des Prozesses an der Stelle des Forkservers mithilfe des `fork()` Syscalls erstellt. Daher stammt auch der Name Forkserver. Er setzt einen Startpunkt für den Fuzzer fest, an dem der Prozess des Programms in dem zu der Zeit feststehenden Zustand mithilfe des `fork()` Syscalls kopiert wird. Dieser Prozess wird auch *Forken* genannt. Dabei entsteht eine logische Einteilung der beiden Prozesse wobei der Elternprozess der Prozess ist, welcher vor dem Forken bereits vorhanden war. Der aus dem Forken entstandene Prozess heißt Kindprozess.

Die von AFL genutzte Syntax besteht aus drei Komponenten.

Listing 3: Syntax des AFL Fuzzing-Befehls

```
$ afl-fuzz -i in/ -o out/ -- /app/mmapp @@
```

Man beginnt mit dem Tool `afl-fuzz`. Daraufhin wird das Flag `-i` mit dem Pfad zum Ordner (hier 3: *in/*) gesetzt, der den Korpus enthält. Anschließend daran wird das Flag `-o` mit dem Pfad zu dem Ordner (hier 3: *out/*) gesetzt, in dem die zur Laufzeit der Fuzzing-Kampagne

generierten Ergebnisse abgelegt werden. Zur besseren Lesbarkeit kann nach dem letzten Ordner ein - - angefügt werden. Danach wird der Pfad zur zu untersuchenden Applikation angegeben. Zuletzt wird die Art der Übermittlung des Inputs dem Binary beschrieben. Die Zeichen `@@` stehen für die Kommunikation mit dem Binary mit Dateien. Die Zeichen dienen als Platzhalter für AFL++ und werden zur Laufzeit des Fuzzers mit den generierten Daten mit dem Namen *.curr_input* ersetzt [17]. Dadurch werden ganze Dateien an das zu untersuchende Binary übergeben. Wenn man jedoch das `@@` weglässt, so wird der Inhalt der im Input-Verzeichnis enthaltenen Dateien sequentiell an das Binary über stdin übergeben.

In dieser Arbeit wird ausschließlich AFL++ verwendet. AFL++ ist eine Abzweigung von AFL und wird von einer breiten Community erweitert und verbessert. AFL und AFL++ unterscheiden sich maßgeblich in der Effizienz der Mutation, der Entdeckung von Codepfaden und einigen in AFL nicht umgesetzten Features. Zur VereinfachungIn wird in den folgenden Kapiteln nur von AFL als Abkürzung für AFL++ gesprochen.

3 American Fuzzy Lop

Bei der Wahl eines Fuzzers gilt es auf folgende Merkmale zu achten:

- Kompatibilität mit verschiedenen CPU (Central Processing Unit)-Architekturen
- Art des Fuzzers
- Strategie des Fuzzers
- Features des Fuzzers
- Benutzbarkeit des Fuzzers

Gerade bei IoT-Programmen ist die Kompatibilität mit verschiedenen CPU-Architekturen ausschlaggebend. Im Fall dieser Arbeit handelt es sich um ein Netzwerk-Binary mit einer ARMv7-Architektur. AFL++ stellt die in diesem Projekt benötigten Werkzeuge bereits zur Verfügung. Die genauere Funktionsweise, Strategie, Kompatibilität sowie die entscheidenden Features von AFL++ werden im Folgenden näher betrachtet.

3.1 Grey-Box-Fuzzing unter AFL++

Ein Fuzzer kann in die Kategorien *White-Box-Fuzzer*, *Black-Box-Fuzzer* und *Grey-Box-Fuzzer* eingeteilt werden. Diese Einteilung ergibt sich aus der Informationsgewinnung des zu untersuchenden Programms.

Ein White-Box-Fuzzer nutzt Informationen von vorhandenem Quellcode und der dessen Logik. Dadurch ist es mit einem White-Box-Fuzzer möglich, sehr komplexe Schwachstellen innerhalb eines Programms zu finden.
Ein Black-Box-Fuzzer hingegen ist ein Fuzzer, der keine Verwendung des enthaltenen Quellcodes und der darin enthaltenen Logik macht. Er prüft lediglich, ob das zu untersuchende Programm den von ihm verwendeten Input akzeptiert[16].
AFL++ ist ein Werkzeug, das ursprünglich dazu vorgesehen ist, eigenen Code an jeder Zweigstelle (branch) im Programmcode eines zu untersuchenden Programms mithilfe des vorhandenen Quellcodes einzufügen. Diesen Prozess nennt man Instrumentation. Durch diesen Schritt kann die erreichte Tiefe (*code coverage*) eines Inputs verfolgt werden [18]. Dieser Schritt wird bereits beim Kompilieren des Programms mithilfe eines angepassten Compilers von AFL ausgeführt. Da AFL++ die Informationen der erreichten Codepfade verwendet, um die Qualität des Inputs einzuschätzen, wird der Fuzzer als Grey-Box-Fuzzer eingestuft.
Da in diesem Projekt eine Applikation untersucht wird, bei der kein Quellcode vorhanden

ist, funktioniert AFL++ anders. AFL++ fügt während der Laufzeit einer emulierten Applikation Instruktionen zur Instrumentierung hinzu. Hierbei werden, ähnlich wie beim zuvor angesprochenen Ansatz, die basic blocks auf Byte-Ebene betrachtet. Dazu wird der Bytecode nicht angepasst. Stattdessen wird jedes Byte jeder Vergleichsinstruktion mithilfe von Hooks, die in AFL++ implementiert sind, verglichen. Dadurch wird ermittelt, ob ein neuer Pfad im Programm gefunden wurde [vgl. 7, S. 6].

3.2 Auswahl des Korpus der Fuzzing Kampagne

Der Korpus – die von AFL und AFL++ verwendeten Inputs – spielt eine zentrale Rolle in der Fuzzing-Kampagne. Er besteht aus Daten, die von dem zu untersuchenden Binary gelesen und verarbeitet werden. Sie dienen als Anhaltspunkt für AFL++, um das Programm zu starten und erfolgreich zu terminieren und somit möglichst viele Programmpfade abzulaufen. Der Korpus kann, je nach Anwendungsfall, unterschiedlich aussehen. Beispielsweise erwartet ein Programm wie Adobe Acrobat Reader, Dateien mit der Dateiendung *.pdf*. Die interne Struktur der darin enthaltenen Daten muss in PDF (Portable Document Format)-Syntax 10 vorliegen. Ein CLI (Command Line Interface)-Programm erwartet andererseits Daten, die ihm bereits beim Start übergeben werden müssen (siehe Listing 11). Der initiale Korpus beim Starten des Binarys mit AFL++ kann aus mindestens einer Datei bestehen. Die optimale Dateigröße beträgt unter einem kB (Kilobyte). Mehrere Dateien im Korpus sollen nur vorhanden sein, wenn diese auch verschiedene Programmpfade - also andere Zustände - des Programms erreichen.

Das Netzwerkprotokoll der zu untersuchenden Applikation verarbeitet Bytes. Hierbei können die an das Protokoll gesendeten Bytes in ihrer Länge variieren. So steht der Befehl `\x02\x00\x00\x00\x00\x02` für das Hochfahren des Gerätes. Bei solchen Standardbefehlen stehen alle Bytes bereits fest und müssen nicht manuell angepasst werden. Es existieren ebenfalls variable Befehle. Der Befehl `\x03\x9A\x00\x00\x01\<Var1>\<CKS>` ist für die Anfrage der CO_2 Ersparnisse einer speziellen Peripherie verantwortlich. Beim Senden dieses Befehls müssen vom Benutzer händisch eingetragene Werte mitgegeben werden. Das Symbol `<Var1>` steht für ein Byte und das Symbol `<CKS>` steht für die Prüfsumme. Sie besteht aus dem letzten Byte der Quersumme aller zu sendenden Bytes. In dem Protokoll, welches diese Befehle verarbeitet, existiert eine *state machine*, welche das Programm anhand der gelieferten Daten in einen definierten Zustand versetzt.

Mit dem Wissen ist ein Befehl in zwei Teile einteilbar.

Listing 4: Formaler Aufbau eines Befehls des Binärprotokolls bestehend aus einem Befehlsrumpf und der dazugehörigen Prüfsumme.

```
<Byte01> <Byte02> <Byte03> <Byte04> <Byte05> ... <CKS>
```

Der zu sendende Befehl besteht mindestens aus fünf Bytes, gefolgt von der bereits beschriebenen Prüfsumme.

3.3 Features von AFL++

AFL ist ein sehr umfangreiches Werkzeug zum Testen der Stabilität von Programmen. Das volle Potenzial des Fuzzers wurde im Rahmen dieser Arbeit nicht ausgeschöpft, da es nicht den Anforderungen der Arbeit entspricht. Jedoch wurden einige Modi und Werkzeuge von AFL verwendet. Die in dieser Arbeit verwendeten Modi und Techniken von AFL werden im Folgenden erläutert. Zuerst wird ein Überblick über den QEMU (Quick Emulator)-Mode zur allgemeinen Instrumentierung des zu untersuchenden Programms gegeben. Im Anschluss werden die Technologien und Ansätze des Fuzzings einer Netzwerkapplikation näher beschrieben, um ein grobes Verständnis der Aufgabenstellung zu schaffen.

3.3.1 QEMU-Mode zur Instrumentation von Binärdateien

Wie bereits in der Sektion 2.3 beschrieben, werden beim Fuzzen des Zielbinarys die Instrumentationsinstruktionen für AFL zur Laufzeit des Programms bereitgestellt. Für das Untersuchen eines Binarys, welches eine andere Architektur als das Hostsystem besitzt, stellt AFL einen Modus zum Emulieren eines anderen Systems zur Verfügung. Dieser Modus in AFL wird *QEMU-Mode* genannt.
QEMU ist ein Open-Source Emulations- und Virtualisierungsprogramm. Die in QEMU enthaltenen Emulationen beschränken sich auf zwei Modi [19]:

- System Emulation
- User Mode Emulation

In dem *System Emulation* Modus wird ein komplettes System als virtualisiertes Modell bereitgestellt. Dazu gehören physische Komponenten wie CPU und physischer Speicher. Die Besonderheit dieses Modus ist, dass die CPU auch direkt mit dem Hostsystem von einem Hypervisor verwendet werden kann.
Bei der *User Mode Emulation* wird die CPU immer vollständig emuliert. Dieser Modus

ermöglicht, es ein Programm platformunabhängig in einer virtualisierten Umgebung, zu starten. AFL verwendet zur Untersuchung eines Binarys ohne Zugriff auf den dazugehörigen Quellcode eine eigene, leicht angepasste Version der User Mode Emulation [20].

Die von AFL benötigten Daten der Codepfadabdeckung werden über einen zwischen QEMU und dem Fuzzer geteilten Speicher an den Fuzzer übergeben. Dabei ist AFL für die Verwaltung des geteilten Speichers verantwortlich. Die von AFL angepasste Version von QEMU ist für das Sammeln und Befüllen des geteilten Speichers verantwortlich [21].

3.3.2 Fuzzing einer Netzwerkapplikation

Das Fuzzen von Netzwerkapplikationen ist mit AFL nicht vorgesehen. Benutzt man hierbei einen herkömmlichen Netzwerkchannel im Gegensatz zu der vorgesehenen Herangehensweise aus der Sektion2.3, so muss mit Geschwindigkeitsverlusten von bis zu einem Faktor von 20 gerechnet werden. Um das Fuzzen von Netzwerkapplikationen zu ermöglichen, müssen einige Tricks verwendet werden. Dazu gehört die Anpassung des Sourcecodes, sodass die Applikation Eingaben über stdin oder über Filedeskriptoren statt über Netzwerksockets liest. Möchte man dies realisieren, so gibt es die Möglichkeit, mithilfe einer *preload library* den bereits implementierten Programmcode zu überschreiben. Von AFL existiert bereits eine rudimentäre Implementierung einer solchen Bibliothek [22], die für jedes Binary überschrieben werden muss. Dieser Bibliothek wurde in dieser Arbeit erweitert und auf die Anforderungen des zu untersuchenden Programmes angepasst. Die genauere Umsetzung der preload library wird in Abschnitt 5.2 beschrieben.

3.4 Risiken

Das Fuzzen von Applikationen bringt auch einige Risiken mit sich, die vor dem Start der Kampagne berücksichtigt werden sollten. Es handelt sich dabei um sehr leistungsintensive Anweisungen. Darunter fallen das Transpilieren – also Übersetzen – von Instruktionen auf eine andere Architektur, das Mutieren der Eingaben und das Verfolgen der abgedeckten Programmpfade, sowie das Lesen von Eingaben und das Schreiben von Output. Insbesondere bei der Parallelisierung der Kampagne und der Ausführung dieser auf mehreren CPU-Kernen ist auf die Temperatur des Testsystems zu achten. Die folgenden Absätze basieren auf der Dokumentation [23] von AFL++ und wurden zur besseren Veranschaulichung erweitert.

Lese- und Schreibinstruktionen, die eine große Belastung eines Systems darstellen, und das damit verbundene Caching können einen wesentlichen Faktor der Erhitzung der CPU

sein. Dies kann zur Drosselung der Leistung des Systems als Selbstschutzmechanismus und im Extremfall zum Systemabsturz oder zur Beschädigung der Hardware führen. Es wird daher empfohlen, eine spezialisierte Langzeitkampagne auf einem System mit ausreichender Kühlung durchzuführen, damit die im System verbauten Komponenten keinen Schaden davontragen.

Darüber hinaus besteht die Gefahr, dass es bei einer lang anhaltenden Kampagne zum Datenverlust kommen kann. Dieses Risiko wird durch die häufige Generierung vieler Dateien und deren Abspeicherung verursacht. Außerdem können viele Logdateien, wie Absturzprotokolle oder *core dumps*, zu einem hohen Speicherverbrauch während der Kampagne führen. Wenn dabei nicht genug Speicherplatz zur Verfügung steht, kann es passieren, dass Daten von dem von AFL erzeugten Output überschrieben werden. Daraus folgt, dass eine Langzeitkampagne – über mehrere Tage – nicht auf Systemen durchgeführt werden sollte, bei denen ein resultierender Datenverlust nicht tragbar ist. Daher sollte man immer feststellen, dass genug Speicherplatz für eine lange Fuzzing-Kampagne zur Verfügung steht.

Schließlich trägt das Fuzzing einer Applikation dazu bei, die Lebensdauer einer physischen Speichereinheit wie einer HDD (Hard Disk Drive) oder einer SSD (Solid State Drive), erheblich zu reduzieren. Dieser Beschädigung der Hardware ist auf sehr intensiven Lese- und Schreibaktivität auf dem Speichermedium zurückzuführen. Bei der Generierung von mutierten Daten und dem Lesen und Schreiben dieser Daten auf ein Speichermedium geschehen im Verlauf einer Kampagne Milliarden von Lese- und Schreibzyklen. Bei dem häufigen Schreiben und Lesen von Daten eines Speichermediums verschleißen die darin eingebauten Teile. Eine Möglichkeit, dem Verschleiß eines Speichermediums im gewissen Umfang entgegenzuwirken, ist das Caching der temporär erzeugten Daten im RAM (Random Access Memory).

Unter Abwägung aller Risiken ist es daher ratsam, anspruchsvolle Fuzzing-Kampagnen auf einem dedizierten System laufen zu lassen. Wenn diese Möglichkeit nicht besteht, sollte darauf geachtet werden, dass eine solche Kampagne nur für eine ausreichend kurze Zeit läuft. Des Weiteren sollte immer eine gute Backup-Strategie zur Hand sein, um einem möglichen Datenverlust entgegenzuwirken.

4 Testumgebung

Zur Informationsgewinnung der Funktionsweise und des Verhaltens des zu untersuchenden Binarys, muss eine Testumgebung implementiert werden. Hierzu wurden die Antworten, die Ausgaben und die durchgeführten Aktionen des Binarys untersucht.
Häufig wurde die Hardware zum Testen der Netzwerkschnittstelle verwendet. Um das Netzwerkprotokoll ortsunabhängig analysieren zu können, ohne die Hardware zur Hand zu haben, wurden mehrere Testumgebungen implementiert. Die konkrete Umsetzung der Testumgebungen wird in den folgenden Kapiteln erklärt.

4.1 Testen der Netzwerkschnittstelle über die Hardware

Um möglichst einfache Fehler bei der Emulation der Umgebung zu vermeiden, wurde die Hardware verwendet. Das IoT-Gerät verfügt über einen Ethernet-Anschluss, über den eine direkte Verbindung zur Datenübertragung hergestellt werden kann.
Um möglichst genaues Feedback über das Verhalten des Binarys zu erhalten, wurde sich über SSH (Secure Shell) mit dem Gerät verbunden. Danach wurde das Binary, welches für das Netzwerkprotokoll verantwortlich ist, beendet und in der Aktuellen SSH-Session neu gestartet. Beginnt man Daten an den Port 7142 zu senden, so erhält man auf der Konsole des IoT-Geräts folgende Ausgabe.

```
[-773231.-945]        [EvtThd_DevRecvFunc:1219] EvtRecv : TCP RECV
[-773231.-944] NOTICE [CMDIF_NecCmdExec:332] Network Standby Command
mmapp:  NOTICE [CMDIF_NecCmdExec:332] Network Standby Command

[-773231.-943]        [CMDIF_NecCmdExec:335] to networkstandby pjcmd exec
[-773231.-943] INFO   [ExecNSPJCmdFunc:4335] iNSIdx=3, iCmdIdx=41
mmapp:  INFO   [ExecNSPJCmdFunc:4335] iNSIdx=3, iCmdIdx=41

[-773231.-943] WARN   [NSPJCMD_037_InformationRequest:7096] DATA MANAGER Read failed.

mmapp:  WARN   [NSPJCMD_037_InformationRequest:7096] DATA MANAGER Read failed.


[-773231.-941]        [CMDIF_NecCmdExec:341] ExecNSPJCmdFunc OK
```

Abbildung 2: Beinhaltet die Ausgabe der Konsole nach händischem Starten des Binarys und Schicken eines Befehls an das Netzwerkprotokoll.

Die Ausführung der Anfrage erfolgt über das Testskript *binaryProt.py* [14]. Das Testskript führt eine in der `main()` Funktion übergebene Funktion aus. Es ist vorgesehen, dass alle Funktionen mit dem prefix *exec* ausgeführt werden können. Soll nur ein Befehl auf einem Gerät ausgeführt werden, so kann man die IP-Adresse über die Konstruktoren mitgeben. Die

dokumentierten Funktionen sind bereits in menschenlesbarer Form in der Klasse *Documented* enthalten. Die Kommunikation über die Netzwerkschnittstelle erfolgt über einen Socket. Hierzu wurde eine von Python bereitgestellte Bibliothek *socket* verwendet. Sie ermöglicht es dem Programmierer, unter Angabe einer IP-Adresse und eines Ports, eine TCP-Verbindung zu einem Gerät aufzubauen. Zur besseren Lesbarkeit wurde eine Hilfsfunktion geschrieben, welche den gesendeten Befehl in hexadezimaler Darstellung mit einer kurzen Beschreibung der Funktionalität des Befehls in der Konsole ausgibt. Zudem gibt das Programm die erhaltene Antwort des Befehls aus, die angibt, ob der gesendete Befehl erfolgreich auf dem Gerät ausgeführt wurde. Erkennbar ist die erfolgreiche Ausführung des Befehls, wenn das erste Byte der Antwort *0x20*, *0x21*, *0x22* oder *0x23* entspricht. Die Antworten sind ebenfalls zum Abgleich mit der Dokumentation des Protokolls [13] gegenzuprüfen.

Listing 5: Ausgabe des implementierten Python-Skripts *binaryProt.py*. Es gibt den gesendeten Befehl in hexadezimaler Schreibweise, sowie die IP-Adresse des Geräts und die vom Gerät gesendete Antwort aus.

```
COMMAND DOCUMENTED: [base model type request] with bytes: 00 BF 00 00
   ↪ 01 00 C0
CHECKING IP-ADDRESS: 127.0.0.1
Sending 00 BF 00 00 01 00 C0
COMMAND SUCCESSFUL: 20 BF 01 40 10 00 FF 22 50 35 30 32 48 4C 00 00 00
   ↪ 00 13 FF 00 DE with checksum DE
```

Die Antworten der Befehle geben einen Hinweis darauf, dass es verschiedene Handler gibt, welche die Anfragen auf diesem Protokoll bearbeiten. Pahl dokumentierte die jeweiligen Handler, welche die Befehle auf dem Gerät bearbeiten, und ihre dazugehörigen Befehle in einer Tabelle mittels des Python-Skripts [24]. Der Tabelle ist zu entnehmen, dass beim Untersuchen der Funktionen und ihrer Handler 53 dokumentierte Funktionen vorliegen, wovon nur 26 eine valide Form besitzen. Von diesen 26 Funktionen besitzen wiederum nur 15 einen Handler.

4.2 Aufbau der Testumgebung mit chroot

Das Programm chroot ermöglicht das Ändern des aktuellen Wurzelverzeichnisses innerhalb eines bereits laufenden Betriebssystems. Unter vielen Linux-Distributionen ist das Wurzelverzeichnis des Dateisystems unter dem Verzeichnispfad **/** zu finden. Die minimale Struktur eines Linux FS (File System) [25] beinhaltet die Verzeichnisse */boot, /dev, /etc, /bin, /sbin* und in einigen Fällen */tmp*. Hierbei befinden sich unter */boot* Informationen zum Starten des

Betriebssystems, unter */dev* Informationen zur verwendeten Hardware, unter */etc* Konfigurationsdateien für das Betriebssystem und unter */bin* und */sbin* ausführbare Dateien (Executables). Im Spezialfall des zu untersuchenden Binarys werden zum Start weitere Verzeichnisse benötigt. Dazu gehören die Verzeichnisse */proc* [26], mit den darin enthaltenen Informationen zum laufenden System und Kernel, */sys* [27] mit enthaltenen Datenstrukturen, welche vom Betriebssystem verwendet werden und */run* [28], welches Systeminformationen nach dem Bootprozess des Betriebssystems enthält.

Listing 6: Mounten der von chroot benötigten Verzeichnisse

```
$ sudo mount -t proc root/proc root/proc/
$ sudo mount -t sysfs root/sys root/sys/
$ sudo mount --rbind /dev root/dev/
$ sudo mount --rbind root/run root/run/
$ sudo mount --rbind /lib64 root/lib64/
$ sudo mount --rbind root/tmp root/tmp/
```

Das Mounten von */lib64* dient in diesem Script [29] nur als Komfort, um nicht alle Abhängigkeiten für AFL manuell kopieren zu müssen. Eine Alternative dazu wäre, das AFL-Executable `afl-fuzz` statisch zu linken, damit es ohne Abhängigkeiten ausführbar ist. Diese Möglichkeit ist bereits im Quellcode von AFL++ eingebaut.
Bevor das Programm gefuzzt werden kann, muss die Umgebung mit entsprechenden Umgebungsvariablen angepasst werden. Hierbei ist die *PATH*-Variable entscheidend. Diese ist dafür verantwortlich, dass Programme in der aktuellen BASH (Bourne Again Shell)-Session gefunden werden.
Anschließend erfolgt der Wechsel in die erstellte Verzeichnisstruktur. Die Syntax von chroot ermöglicht es, nach dem Wechsel in das neue Wurzelverzeichnis, einen BASH-Befehl auszuführen. In diesem werden alle – für AFL relevanten – Umgebungsvariablen gesetzt und mit passender Instrumentierung gestartet.

Listing 7: Wechseln in ein anderes Wurzelverzeichnis mit chroot und ausführen von AFL

```
$ sudo chroot root/
    bash -c 'export AFL_DEBUG=1; export AFL_PRELOAD=./sockfuzz.so;
    export QEMU_LD_PREFIX=/;
    export PATH="/usr/gnu/bin:/usr/local/sbin:/usr/local/bin:/bin:/sbin
       ↪ :/usr/bin:.";
    export QT_X11_NO_MITSHM=1; export DISPLAY=:10;
    ./afl-fuzz -Q -i in -o out -t 50000 -- /app/mmapp @@'
```

4.3 Aufbau der Testumgebung mit bwrap

Bubblewrap ist ein Low-Level Sandboxing-Tool. Es ermöglicht dem Nutzer, eine containerisierte Laufzeitumgebung ähnlich wie Docker, zu erstellen. Der Unterschied zu Docker ist, dass Bubblewrap [30] (oder kurz *bwrap*) es ermöglicht, auch unprivilegierten Benutzern Container – mithilfe von Linux Kernel-User-Namespaces – zur Verfügung zu stellen. Der Vorteil bei der Umsetzung einer Testumgebung mit einem Container liegt darin, dass es unwahrscheinlicher ist, das Host-System fälschlicherweise umzukonfigurieren. Wie bereits in Abschnitt 4.2 angesprochen, müssen auch mit bwrap die entsprechenden Verzeichnisse gemountet werden, sodass das Binary fehlerfrei funktioniert. Das muss jedoch nicht wie bei chroot auf dem Host-System erfolgen, sondern in einer Containerumgebung. Bwrap erstellt bei Aufruf zuerst ein leeres FS. Danach können alle benötigten Verzeichnisse in den Container gemountet werden 14. Diese Methode hat sich als die beste Methode für das hardwareunabhängige Testen erwiesen. Mit der Isolation der Applikation über Kernel-Namespaces bietet BWRAP (Bubblewrap) mit einer einfachen Syntax eine sichere Möglichkeit, Applikationen in eine Testumgebung zu packen. Aufgrund der Containerisierung ist es im Gegensatz zu chroot nur erschwert möglich, das eigene Betriebssystem mit dem falschen Mounten von Verzeichnissen zu beschädigen. Auch hat man bei der Konfiguration und Inbetriebnahme der Applikation keinen Overhead der Virtualisierung einer kompletten Plattform wie mit Docker, was bwrap zu einem geeigneten Tool zur Implementierung einer leichtgewichtigen Testumgebung macht.

5 Fuzzing-Kampagne

Eine Fuzzing-Kampagne folgt einer in der Regel festen Struktur.

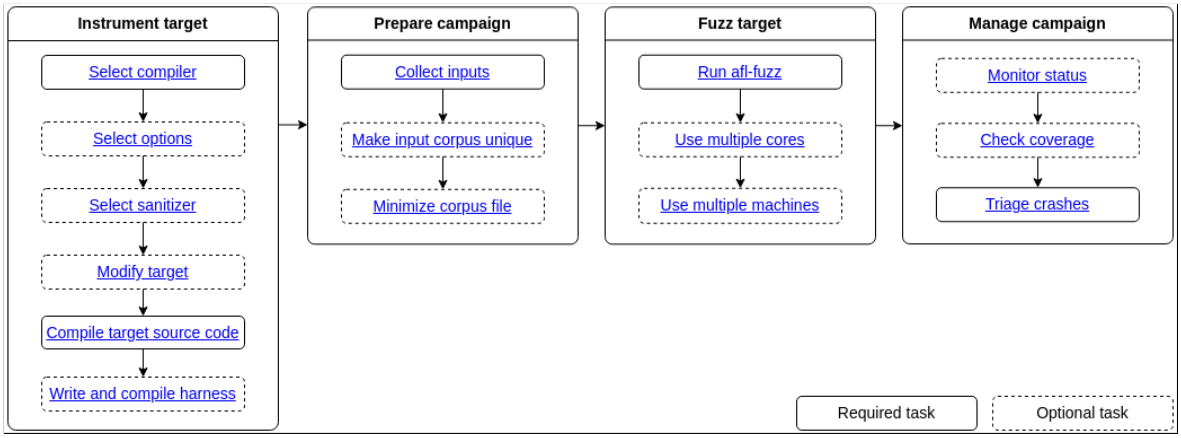

Abbildung 3: Visuelle Darstellung einer Fuzzing-Kampagne in mehreren Etappen, wenn der Quellcode eines Programms zur Verfügung steht [31].

Der erste Schritt besteht darin, das zu untersuchende Programm zu instrumentieren. Wenn der Quellcode der Applikation verfügbar ist, können in diesem Schritt Anpassungen am Quellcode gemacht werden. Dies geschieht, indem von AFL bereitgestellte Funktionen und Makros in den Code geschrieben werden. Dadurch ist es möglich, den Teil der Applikation zu isolieren, der untersucht werden soll.

Als Nächstes kommt die Sammlung möglicher Inputs für das instrumentierte Programm. Hierbei werden alle gültigen Eingaben gesammelt. Dabei ist zu beachten, dass die Dateien, die die Eingaben beinhalten, möglichst klein gehalten werden. Um möglichst wenig Zeit und Performance der Kampagne zu verlieren, sollte mit dem von AFL bereitgestellten Werkzeug `afl-cmin` sichergestellt werden, dass jede Eingabe für das Ablaufen eines anderen Codepfades verantwortlich ist.

Im Anschluss wird der Fuzzer gestartet und das Programm wird mit dem gelieferten Input getestet. Der Fuzzer mutiert den Input, nachdem alle Testfälle bereits mindestens ein Mal mit dem zu untersuchenden Programm geprüft wurden. Es wird empfohlen, dass die Mindestdauer einer Kampagne mindestens eine Stage beträgt. Eine Stage ist der Durchlauf der Kampagne, bis keine neuen Codepfade mehr entdeckt werden. Die benötigte Zeit für das Durchlaufen einer Stage hängt von der Komplexität des Programms ab. Sie kann von einem Tag bis zu einer Woche dauern.

Zuletzt werden die gefundenen Abstürze und Hänger – sofern sie gefunden wurden – untersucht und verifiziert.
Beim Fuzzing ohne Sourcecode fällt jedoch der Schritt der Instrumentierung weg. Dieser Schritt wird von dem QEMU Modus des Fuzzers übernommen.

Eine solche Kampagne stellt einen zyklischen Prozess dar. Nachdem eine Kampagne abgeschlossen wurde, ist es ratsam die bereits gefundenen Daten im Output-Verzeichnis wiederzuverwenden und beim Schritt der Vorbereitung der Kampagne von vorne anzufangen.

5.1 Instrumentation des Programms mit AFL++

Fuzzing mit AFL folgt einer klaren Vorgehensweise. Da es sich bei der Umsetzung der Fuzzing Kampagne um ein Closed-Source-Binary handelt, wird der von AFL bereitgestellte QEMU verwendet. QEMU ist eine Virtualisierungsumgebung, welche dafür verwendet wird, das Binary in einer emulierten Umgebung starten zu können. Aufgrund der hohen flexibilität von QEMU ist es auch möglich, plattformunabhängig Binarys einer anderen Architektur auszuführen. Somit ist es möglich, auf einer ARM Plattform Programme mit x86 Architektur und umgekehrt auszuführen. Die Plattformunterstützung muss beim Bau von AFL manuell mitgegeben werden. Das kann durch das Bauen des QEMU-Supports mithilfe des Komfortscripts *build_qemu_support.sh* in Kombination des Setzens einer Umgebungsvariable `CPU_TARGET=arm` 12 erreicht werden [32].
Dieser Modus ist beim Starten des Fuzzers mithilfe des `-Q` Flags benutzbar. Wie die QEMU Umgebung mit der Firmware vorbereitet wird, ist in Abschnitt 4.3 beschrieben.

5.2 Übergabe von Daten als Parameter mithilfe von LD_PRELOAD

Wie bereits in Abschnitt 3.3.2 erläutert, ist das Fuzzen einer Netzwerkapplikation in dem bereits implementierten Featureset von AFL nicht enthalten. Hierzu kann eine Bibliothek selbst implementiert und als shared object kompiliert werden, sodass diese zum Start der Applikation, mithilfe einer Umgebungsvariable namens `LD_PRELOAD`, in die Applikation geladen wird. Die Bibliothek beinhaltet eine eigene Implementierung der Syscalls `accept()`, `recv()`, `send()` und `socket()`, welche die Funktionalität der bereits in libc-Bibliothek enthaltenen Funktionen überschreibt. Um diese Funktionen effektiv für AFL nutzen zu können, müssen die genannten Syscalls so umgeschrieben werden, dass Input, welcher über einen Netzwerksocket empfangen werden würde, stattdessen über die Eingabe von Dateien als Startparameter übergeben wird. Dieser Vorgang wird in der Fuzzing Community auch *desocketing* [33] genannt. AFL besitzt bereits die Funktionalität ein shared object in Form einer *.so* Datei in das

zu untersuchende Programm zu laden. Unter AFL wird die Umgebungsvariable `AFL_PRELOAD` genannt.
Bei diesem Binary werden jedoch keine Standard-Syscalls verwendet, sondern eine vom Hersteller implementierte Abstraktionsschicht, welche die Funktionalität der Syscalls erweitert.

```
void debug(char* format, ...) {...}

int __libc_start_main(int (*main) (int, char * *, char **), int argc, char ** argv, void (*init) (void),
                      void (*fini) (void), void (*rtld_fini) (void), void (*stack_end)) {...}

int SOCK_Bind(sock_t* sock, const struct sockaddr *addr, socklen_t addrlen) {...}
int SOCK_Listen(sock_t *sock, int backlog) {...}
int SOCK_AssocEvt(sock_t *sock, int *evt, int flags) {...}
int SOCK_EnumEvt(sock_t *sock,int *evt, int *x, int *y) {...}
sock_t *SOCK_Open(int domain, int type, int protocol) {...}
sock_t *SOCK_Accept(sock_t *sock, struct sockaddr *addr, socklen_t *addrlen) {...}
int SOCK_Recv(sock_t *sock, void *buffer, size_t size) {...}
int SOCK_Send(sock_t *sock, void *buffer, size_t size) {...}
int SYS_ReadyThd(void) {...}
int SYS_AttachEvt(int *evt, int flag) {...}
int SYS_WaitEvts(int flag) {...}
int *SYS_CreateMsg(byte *old_msg, size_t size) {...}
int SYS_ReceiveMsg(int *x, int **y, int *z) {...}
int DoNothingThread(void) {...}
int SYS_CreateThd(int (*func)(void), thread_data_t *data, size_t data_size, const char *name) {...}
int SOCSVR_EventEnque(void *msg,uint msg_len,ushort seq_number,int socket,queue_t *queue) {...}
socsvr_t *SYS_GetMem(sys_mem_t *sys_mem) {...}
```

Abbildung 4: Zeigt die von Pahl überschriebenen Funktionen der preload Bibloiothek sock-fuzz.so. Sie bestehen aus der vom Hersteller implementierten Abstraktionen der Standardfunktionen der libc Bibliothek.

Das Ziel der Bibliothek ist es, das Überschreiben des TCP-Sockets auf Port 7142, auf dem der gewünschte Netzwerkservice läuft, sodass beim Fuzzen der Applikation keine Netzwerkanfragen lokal simuliert werden müssen. Wichtig zu erwähnen ist hierbei, dass das direkte Senden von Daten an das Programm über das C CLI eine Performancesteigerung um den Faktor der Größe 10 entspricht [22]. Dies ist darauf zurückzuführen, dass weniger Overhead über den TCP-Stack mitgegeben werden muss. Ausschlaggebend bei dem Überschreiben der Abstraktionsschicht sind alle Funktionen, die nach einem Socket verlangen, sowie die Funktion `__libc_start_main()`.
Die Funktion `__libc_start_main()` [34] ist der Einstiegspunkt eines ausführbaren C-Programms und sorgt für das Initialisieren der Laufzeitumgebung des Programms. Ebenfalls ist die Funktion dafür verantwortlich die `main()` Funktion mit den zum Starten des Programms benötigten Parametern (typischerweise `int argc` und `char *argv[]`) aufzurufen. Der Einsprungspunkt muss überschrieben werden, damit die von AFL generierten Eingaben an das Programm als Parameter zum Start des Programms übergeben werden können. Dazu

wird der `char *argv[]` Parameter der `main()` Funktion betrachtet. Da der erste übergebene Parameter in C immer das Programm selbst ist, ist der zweite Parameter `argv[1]` der ausschlaggebende Parameter. Dieser über die Konsole übergebene Parameter wird bisher nicht von dem Programm verwendet und kann somit zur Übergabe der Eingaben verwendet werden. Als Nächstes wird die übergebene Datei an das Programm weitergegeben und gelangt weiter in den Calltree. TCP-Sockets unter auf UNIX (Uniplexed Information Computing System) basierten Systemen arbeiten mit (Pseudo-)Dateien, welche einen Datenstrom repräsentieren. Somit müssen alle Syscalls, welche auf den Socket 7142 verweisen, angepasst werden. Sie sollen nicht mehr auf eine Datei, welche einen Socket repräsentiert, sondern auf die Datei, die von AFL erzeugt wird, zugreifen. Dafür soll die abstrahierte Version `SOCK_Bind()` des `bind()` Syscalls überschrieben werden. Er ist dafür verantwortlich, dass die Kommunikation über den richtigen Socket erfolgt.

5.3 Optimierungen

Aufgrund der langsamen Durchführung eines Fuzzing-Zyklus, muss die Kampagne angepasst werden. Je öfter ein Zyklus durchlaufen wird, desto höher ist die chance innerhalb eines Zeitintervalls Fehler im Programm zu finden. Ein Zyklus besteht aus dem Starten der Applikation, Weiterreichen der Daten bis zur gewünschten Stelle, der Terminierung des Programms und Mutation des Inputs anhand des erlangten Feedbacks.
Ein Schritt zur Beschleunigung der Kampagne wäre, den Netzwerkservice der Applikation mittels Reverse-Engineering-Techniken zu isolieren. Dieser Prozess ist jedoch sehr zeitintensiv und zum Zeitpunkt der Verfassung dieser Arbeit nicht umsetzbar.

5.3.1 Minimierung des Korpus

Ein wichtiger Bestandteil einer erfolgreichen Fuzzing-Kampagne ist der Korpus. Er ist das Mittel, mit dem AFL neue Codepfade (bzw. Call-trees) durchlaufen kann. Ein Korpus ist nur dann einzigartig, wenn alle darin enthaltenen Daten jeweils unterschiedliche Codepfade ablaufen und eine möglichst große Tiefe – also möglichst viele branch edges – im Code erreichen. Für den ersten test des Fuzzers werden alle dokumentierten Funktionen verwendet und getestet. Die Funktionen haben jeweils anders implementierte Handler für ihren speziellen Anwendungsfall. Zum Prüfen, ob es redundanten Input gibt, empfiehlt es sich, die Umgebungsvariable `AFL_DEBUG=1` zu setzen. Sie ermöglicht es, weitere Informationen, wie Debug-Instruktionen des Programms (alle print-Statements und alle `puts()` aufrufe) oder Informationen zur aktuell laufenden Kampagne anzuzeigen.
AFL bietet ein Tool namens `afl-cmin` [35] zum Minimieren der Testcases und zum Prüfen

```
[!] WARNING: No new instrumentation output, test case may be useless.
[!] WARNING: Instrumentation output varies across runs.
[*] Attempting dry run with 'id:000007,time:0,execs:0,orig:008'...
[D] DEBUG: calibration stage 1/7
```

Abbildung 5: Zeigt eine Warnung von AFL. Sie warnt vor einem oder mehrerer Testcases, welche duplizierte Codepfadabdeckung bezwecken.

des Korpus. Das Tool folgt einer ähnlichen Syntax wie `afl-fuzz` (siehe Listing: 3), wobei das Flag `-i` für den Ordner mit dem Korpus steht, der analysiert werden soll. Das Flag `-o` steht für den Ordner, in den der minimierte Korpus geschrieben werden soll. Die Verwendung des Tools hat den Vorteil, dass das Fuzzing und die Mutation des Inputs nach jeder Stage eines Fuzzing-Zyklus, aufgrund geringen Inputs, welcher mutiert werden muss und geringeren Inputs, welcher getestet werden muss, beschleunigt wird.

5.3.2 Verwendung des Persistent Modes

Als nächsten Performanceboost empfiehlt es sich, den bereits von AFL implementierten *Persistent Mode* zu verwenden. Der Persistent Mode erlaubt es AFL, ein Programm innerhalb zweier Speicheradressen mehrmals zu fuzzen, ohne dass der Prozess, auf dem der Fuzzer läuft, erneut geforkt wird. Hierzu muss das zu fuzzende Programm mit einem Hilfsprogramm wie dem GDB (GNU Debugger) des GNU (GNU's Not Unix)-Projekts untersucht werden, um eine geeignete Speicheradresse als Einsprungspunkt definieren zu können.
Hierzu wird exemplarisch eine selbst entwickelte Applikation [36] zu Demonstrationszwecken hergenommen und analysiert, welche ebenfalls einen TCP-Stack verwendet. Diese Applikation ist stark vereinfacht und bringt nicht die gleiche umfangreiche Funktionalität und Komplexität, wie das Ursprungsbinary *mmapp*, mit sich. Um sich der Analyse eines ARM-Binarys anzunähern, wurde das Beispielsbinary für ARM kompiliert. Dazu wurde ein Komfortskript [37] geschrieben, welches einen bereits für diesen Zweck gebauten Docker-Container [38] verwendet, um das Binary zu kompilieren. Die Besonderheit bei der Kompilierung des Binarys, um möglichst wenig Abweichung zu haben, ist die von dem Ursprungsbinary *mmapp* verwendete GCC (GNU Compiler Collection) Version *4.8.1*.
In Abbildung 6 ist der dazugehörige Disassembly Code des Beispielprogramms. Hier ist zu sehen, dass die `main()` Funktion einen Funktionsaufruf auf eine Funktion namens `simulateHeavyWorkload()` macht. Diese Funktion ist im übertragenen Sinne für das Initialisieren des IoT-Geräts verantwortlich. Diese Funktion sollte während des Fuzzens möglichst nur ein Mal aufgerufen werden, da das Initialisieren der gesamten Peripherie viel Zeit in Anspruch nimmt. Das kann erzielt werden, indem man AFL mittels eines *"persistent*

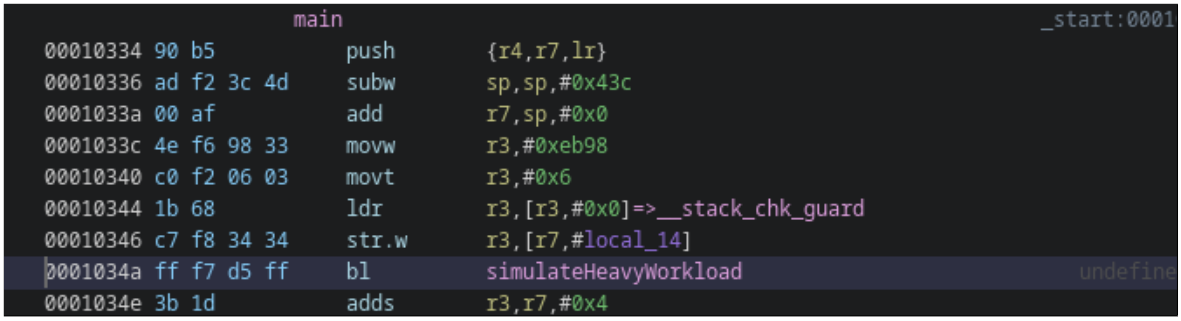

Abbildung 6: Zeigt den disassemblierten Code des selbst implementerten TCP-Servers. In der Abbildung ist eine Funktion `simulateHeavyWorkload` in der Speicheradresse `0x001034a` zu sehen. Das Ziel der Definitiion der Startadresse ist es, diese zeitintensive Funktion zu umgehen.

*loop"*s innerhalb zweier Speicheradressen instrumentiert. Hierfür muss eine Startadresse `AFL_QEMU_PERSISTENT_ADDR` an der gewünschten Speicheradresse `0x0001034e`, an der der Persistent-Loop anfangen soll, definiert werden. Da hier nur ein Teil der `main()` Funktion gefuzzt werden soll und das Programm nicht unter normalen Bedingungen terminieren sollte, muss eine Schlussadresse `AFL_QEMU_PERSISTENT_RET` [39] definiert werden. In diesem Fall soll der Persistent-Loop nach der Verarbeitung der empfangenen Daten enden.

Abbildung 7: Disassembly des selbst implementierten TCP-Servers. Diese Abbildung zeigt die für das Fuzzing wichtige Logik, in der die empfangenen Daten verarbeitet werden. Hierbei ist die Schlussadresse `0x000104ae` ausschlaggebend, in der die Verarbeitung der empfangenen Daten stattfindet.

Die letzte Instruktion, die ausgeführt werden soll ist somit in der Speicheradresse `0x000104ae` 7.

5.3.3 Verwendung mehrerer CPU-Kerne

Mit AFL ist es auch möglich, mehrere Instanzen des Fuzzers laufen zu lassen, um die Performance weiter zu verbessern. Dazu wurde eine Funktionalität implementiert, die es mehreren Instanzen ermöglicht, parallel zueinander zu fuzzen. Die Limitation besteht darin, dass nicht mehr Instanzen gestartet werden können, als CPU-Kerne auf dem System verfügbar sind. Außerdem ist es ratsam, die Anzahl der Kerne zu untersuchen, die zur Verwendung einer Kampagne nützlich sind ohne einen Perfomanceverlust zu bezwecken. Diese Zahl liegt in der Regel zwischen 32 und 64 Kernen [40].

Das Feature ist mit dem Setzen zweier Flags umsetzbar. Under Verwendung des Flags `-M` kann die Hauptinstanz definiert werden. Der Zweck eines Hauptfuzzers ist das Sammeln aller Testcases. Dadurch werden nach der Terminierung einer Fuzzer-Instanz die gefundenen einzigartigen Testcases in den Hauptfuzzer importiert.
Jeder Untergeordnete Fuzzer wird mit dem `-S` Flag deklariert.
Dadurch entsteht die daraus resultierende Syntax zum Starten mehrerer Instanzen:

Listing 8: Syntax des AFL fuzzing Befehls mit der Definition der Hauptinstanz mithilfe des `-M` Flags

```
$ afl-fuzz -Q -M main -i in/ -o out/ -- /app/mmapp @@
```

Listing 9: Syntax des AFL fuzzing Befehls mit der Definition einer Nebeninstanz mithilfe des `-S` Flags

```
$ afl-fuzz -Q -S qasan -i in/ -o out/ -- /app/mmapp @@
```

Das Fuzzing mittels mehrerer Kerne macht am meisten Sinn, wenn jede Instanz eine andere Strategie umsetzt. Es wird dabei empfohlen, andere Mutationsstrategien des Fuzzers zu verwenden, um möglichst viele mutierte Eingaben zu erzielen, die voneinander abweichen.

6 Auswertung der Ausgabe von AFL++

Um eine erfolgreiche Fuzzing-Kampagne zu entwickeln, ist es hilfreich, die Ausgaben des Fuzzers auswerten und deuten zu können. In diesem Kapitel werden alle Ausgaben des AFL erklärt und darauf eingegangen, wie anhand dessen zu einer möglichst effizienten und zielführenden Kampagne gelangt werden kann. Zu den von AFL bereitgestellten Ausgaben gehören die Fuzzingstatus Anzeige und die vom Fuzzer erzeugten Crashes, sofern diese aufgetreten sind.

6.1 Die Fuzzingstatus Anzeige

Der American Fuzzy Lop ist ausgestattet mit einer Statusanzeige, welcher den aktuellen Stand der Kampagne anzeigt.

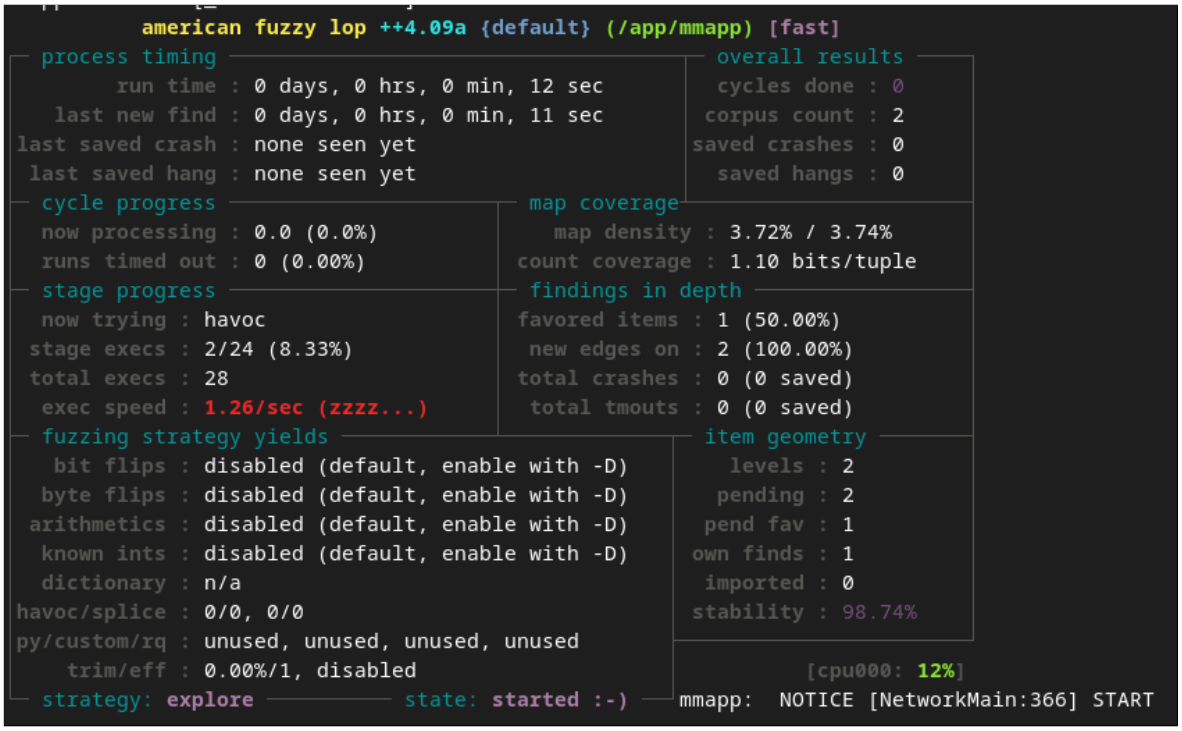

Abbildung 8: Statusbildschirm von AFL

Der obere Teil der Statusbildanzeige wird als Statusleiste bezeichnet. In ihr wird angezeigt, welche Version des AFL verwendet wird, sowie in welchem Modus (hier *{default}*) der Fuzzer gestartet wurde. Als Nächstes steht die Applikation, welche derzeit gefuzzt wird. Der Letzte, in der Statusleiste angezeigte, Wert beschreibt den Energiesparplan-Modus. Im Normalfall wird der Modus *fast* verwendet.

Desweiteren gibt es in der Statusanzeige ein Abteil mit dem Titel *process timing*. Ein besonderes Merkmal dieser Anzeige ist die Sektion *last new find*. Sie zeigt an, wann ein neuer Codepfad des Programms entdeckt wurde.

Die Sektion *overall results* beinhaltet eine Übersicht verschiedener Ergebnisse der Kampagne. Dazu gehört ein Zähler, welcher die Gesamtzahl der Iterationen aller Testcases beinhaltet. Er wird erst hochgezählt, wenn alle im Korpus enthaltenen Testcases durchlaufen sind. Darunter wird angegeben, wie viele Daten im Corpus vorhanden sind. Anhand dessen kann man sehen, wie viel neuer Input von AFL generiert wurde. Zudem gibt es zwei weitere Zähler, die die Anzahl der Crashes und Hangs, die der Input verursacht hat, festhalten.

Der Abschnitt *cycle progress* gibt Informationen über den derzeit verwendeten Input – innerhalb eines Fuzzing-Zyklus – mithilfe seiner ID (Identifier). Außerdem wird hier auch die Anzahl an Eingaben festgehalten, welche die Kommunikation mit dem Programm zu einem Timeout führen.

Als Nächstes gibt es den Teil *stage progress*. Dieser gibt Auskunft über das aktuelle Verhalten des Fuzzers, sowie die derzeitige Phase, in dem sich der Fuzzer gerade befindet. Das Feld *now trying* steht dabei für die derzeit verwendete Mutationsstrategie des Inputs. Das Schlüsselwort *havoc* beschreibt, dass der Korpus derzeit mittels bitflips und Überschreiben von Integern innerhalb des Testcases mutiert wird [41]. Die weiteren Felder beschreiben die Ausführgeschwindigkeiten. Sie zeigen ebenfalls auf, in welchem Zyklus sich der Fuzzer befindet und wie oft ein Durchlauf bereits ausgeführt wurde.

In der darauffolgenden Sektion *findings in depth* werden Metriken für besondere Entdeckungen festgehalten. Dazu gehören die Anzahl der Eingaben, die besonders gute Codepfadabdeckungen aufweisen (*favored items*) oder besonders weit in den Programmcode gelangen (*new edges on*). Außerdem werden die Anzahl der Inputs aufgezeichnet, die Crashes und Timeouts im Programm hervorrufen.

Die letzte – für die Validierung der Kampagne wichtige – Sektion *path geometry* beinhaltet eine generalisierte Sicht auf die Eingaben. In diesem Abschnitt werden weitere Daten der Kampagne zur Analyse der Richtigkeit der Ausführung zusammengefasst. Dazu gehört das Feld *levels*, welches die Tiefe der Kampagne veranschaulicht. In anderen Worten zeigt das Feld auf, wie viele Mutationszyklen die Eingaben bereits durchlaufen haben. Die weiteren

Felder *pending* und *pend fav* Zeigen auf welche Inputs von der Kampagne noch nicht zum Fuzzen verwendet wurden, wobei *fav* die von AFL als favorisiert markierten Inputs sind. Die favorisierten Inputs werden mit höherer Priorität von AFL abgearbeitet. Die nächsten beiden Felder *own finds* und *imported* zeigen die Anzahl der neu entdeckten Codepfade auf. Hierbei anzumerken ist, dass unter Setups, mit zueinander parallel laufenden Fuzzern, *own* die Codepfade der laufenden Instanz sind und *imported* die importierten Codepfade anderer Fuzzerinstanzen nach der Synchronisation derer miteinander aufzeigt. Das letzte Datenfeld *stability* misst die Zuverlässigkeit der Reaktion des Programms auf die vermittelten Eingaben. Es zeigt die Anzahl der Testcases, welche bei mehrfachem Testen die gleichen Effekte/Codepfade erreicht haben. Somit wird, wenn alle Testcases nach mehrfachem dasselbe Verhalten in einem Programm hervorgerufen haben, die Anzeige bei 100 % bleiben.

Die für den Tester interessanten Felder beschäftigen sich mit der Stabilität des Inputs, der Geschwindigkeit der Kampagne und den Crashes. Die Wahrscheinlichkeit, dass die Kampagne den gewünschten Effekt der Entdeckung eines Programmcrashs erzielt, hängt davon, ab wie gut die Eingaben sind und wie schnell der Fuzzer das Programm ausführen kann. Die Güte des Inputs wird daran bestimmt, ob ein Input eine möglichst hohe Programmpfadabdeckung hat und wie stabil das Programm auf den Input reagiert. Wenn ein Input beispielsweise bei jedem Durchlauf ein anderes Verhalten im Programm triggert, so ist dieser Input aufgrund seiner unverifizierbarkeit wertlos. Ein Input, der jedoch am Beispiel von mmapp einen Handler zur Verarbeitung dessen abläuft und möglichst weitere Programmfunktionen auslöst, wird als guter Input bewertet. Außerdem ist das Ziel eines Fuzzers Crashes zu finden, welche man potenziell ausnutzen kann.

6.2 Dokumentierte crashes, hangs und buggs

Bevor auf die Crashes eingegangen wird, werden die im output Verzeichnis angelegten Daten und Verzeichnisse genauer erläutert.
Die drei Hauptverzeichnisse bestehen aus *crashes/*, *hangs/* und *queue/*. Bei dem Verzeichnis *queue/* handelt es sich um den synthetischen Korpus. Dieser wurde bei der Mutationsphase von AFL mithilfe der bereitgestellten Testcases erstellt.
Die Verzeichnisse *hangs/* und *crashes/* existieren zur Dokumentation der durch Inputs verursachten Aufhängern und Programmabstürzen. Die jeweiligen Hangs und Crashes werden mit einer ID versehen, welche der numerisch sortierten chronologischen Reihenfolge der Crashes entspricht. Gefolgt werden sie von der Nummer des dafür verantwortlichen Inputs. Der Inhalt der darin enthaltenen Daten entspricht dem Input, welcher das Programm zum

```
out/
└── default/
    ├── .cur_input
    ├── .synced/
    ├── cmdline
    ├── crashes/
    ├── fuzzer_setup
    ├── fuzzer_stats
    ├── hangs/
    ├── hangs.2024-02-28-15:00:38/
    │   ├── id:000000,src:000001,time:69743,execs:110357,op:havoc,rep:11
    │   ├── id:000001,src:000001,time:209883,execs:110371,op:havoc,rep:3
    │   └── id:000002,src:000001,time:361580,execs:337735,op:havoc,rep:3
    ├── plot_data
    └── queue/
        ├── .state/
        │   ├── auto_extras/
        │   ├── deterministic_done/
        │   ├── redundant_edges/
        │   └── variable_behavior/
        ├── id:000000,time:0,execs:0,orig:test
        └── id:000001,time:0,execs:0,orig:cmd1
```

Abbildung 9: Verzeichnisstruktur des output Verzeichnisses von AFL

Absturz oder Aufhängen gebracht hat. Als Crash wird ein Fall bezeichnet, bei dem ein Input das Senden eines Signals wie `SIGSEGV` für *segmentation fault* und somit den Absturz des Programmes verursacht. Hangs hingegen sind Fälle, welche das von AFL definierte Zeitlimit von einer Sekunde oder der selbst definierten Zeitspanne – mithilfe des `-t` Flags konfiguriert – überschreiten. Es wird zwischen einzigartigen und nicht einzigartigen Crashes und Hangs unterschieden. Einzigartig sind diese nur, wenn der übermittelte Input einen zuvor unabgedeckten Codepfad überquert [42].
Zur Zeit der Verfassung dieser Arbeit wurden in dem zu untersuchenden Binary *mmapp* keine Crashes mithilfe des QEMU Modus gefunden.

Die Verifikation eines Crashes erfolgt durch das Ausführen des zu untersuchenden Programms mit dem Input, welcher den Crash erzeugt hat. Wenn der Crash mithilfe des Inputs reproduzierbar ist, so kann man sich den zugrundeliegenden *code.dump* genauer ansehen. Im Fall eines segmentation faults liegt dann eine potenzielle Memory-Corruption-Schwachstelle vor.

7 Probleme

Im Verlauf der Bearbeitung der Aufgabenstellung traten Probleme auf, deren Lösung im Folgenden aufgezeigt wird.

Das erste Problem, welches sich bereits am Anfang der Arbeit stellte, war die Erstellung eines Docker Containers. Dieser Docker Container sollte es ermöglichen, die in dieser Arbeit verwendeten shared library mithilfe einer alten, vom Binary genutzten GCC-Version, für ARM zu kompilieren.
Der ursprüngliche Ansatz der Aufgabe bestand daraus, die Instruktionen aus dem Dockerfile eines bereits vorhandenen ARM-Containers aus Dockerhub [38] herauszukopieren und diese auf die gestellte Anforderung der GCC-Version 4.8.1 anzupassen. Dieser Ansatz stellte sich jedoch schnell als sehr Zeitintensiv heraus, da das Nachvollziehen fremder Logik innerhalb eines Dockerfiles viel Zeit in Anspruch nimmt. Der nächste Ansatz war es, ein Installationsskript anhand der Dokumentation von GCC [43] selbst zu erstellen. Dieser Ansatz war jedoch erfolglos. Die Schwierigkeit bestand daraus, die von der Version des Compilers benötigten Dependencies zu erfüllen. Eine effiziente Lösung des Problems war es, ein altes Image zu verwenden, welches ein Betriebssystem aus dem Erscheinungsjahr der GCC-Version verwendet. Daraufhin konnte die Installation des Compilers begonnen werden. Da das Kompilieren von GCC sehr Zeit- und Ressourcenintensiv (ca. 20 Minuten) ist, wurden mehrere Instanzen des Docker Containers gestartet, um parallel an anderen Lösungsansätzen beim Auftreten eines Fehlers zu arbeiten. Diese Fehler waren jedoch der Inkompatibilität der auf dem Image verwendeten glibc Version geschuldet. Aufgrund der Komplexität und des hohen Zeitaufwands dieser Aufgabe wurde der Ansatz des Bauens eines eigenen Docker Image verworfen. Der letzte und schlussendlich eingesetzte Ansatz ist das Suchen eines bereits funktionierenden GCC Binaries, das für eine auf Debian basierte Linux Distribution gebaut wurde. Dafür wurde in den Archiven der Debian und Ubuntu Repositories [44] nach einer kompatiblen GCC-Version gesucht. Die kompatible GCC-Version wurde dann innerhalb eines Shellscripts in einem für ARM gebauten Container installiert und konnte dann verwendet werden.

Bei der Entwicklung des Testsystems kam es ebenso zu einigen schwerwiegenden Problemen. Eines der Probleme beinhaltet die Entwicklung der chroot Umgebung. Bei dem Mounten der verschiedenen Verzeichnisbäume in eine isolierte Umgebung kann es zu unerwünschten Beierscheinungen, wie dem Neudefinieren von für das Betriebssystem notwendigen Verzeichnissen. Ebenfalls hat das chroot Skript eine Bereinigungsfunktion, welche den Zustand des im Gitlab [45] enthaltenen rootfs berücksichtig. Diese Funktion löscht alle neu angelegten Dateien,

welche nicht im Git Repository nachverfolgt werden, um möglichst nah am Ausgangszustand der Firmware zu bleiben. Somit kann es passieren, dass falsch gemountete Verzeichnisse gelöscht werden und damit für das Betriebssystem wichtige Dateien verloren gehen. Es ist also ratsam, eine virtuelle Maschine zu Testzwecken des chroot Skripts zu verwenden, damit das eigene Betriebssystem unversehrt bleibt.

8 Fazit

In dieser Praxisarbeit zum Thema des Fuzzings eines Netzwerkprotokolls eines IoT Binarys wurde die grundlegende Herangehensweise zum Fuzzen und zum Entwickeln einer verwendbaren Testumgebung vermittelt. Quereinsteiger in das Thema Fuzzing haben so die Möglichkeit, möglichst schnell zu einem erwünschten Ziel der Entwicklung einer lauffähigen Fuzzingumgebung zu kommen. Zudem wurden Erkenntnisse in die Funktionsweise von AFL dargestellt, welche die Einarbeitung in das Thema Fuzzen mit AFL vereinfachen sollte.
Des Weiteren wurden Methodiken zum Optimieren der Fuzzing Kampagne, wie desocketing einer Netzwerk basierenden Applikation beleuchtet, welche es neuen Nutzern von AFL ermöglicht, möglichst performante Kampagnen zu starten. Das sollte die Wahrscheinlichkeit der Findung neuer Bugs und Schwachstellen erhöhen.
Als Nächstes wurden die Möglichkeiten zur Auswertung einer erfolgreichen Kampagne erläutert. Diese ermöglichen es dem Nutzer des AFL, die Ausgaben der Kampagne zu deuten und potenzielle Schwachstellen mithilfe gezielter Instrumentation zu verifizieren.
Als Letztes wurden aufgetretene Probleme angesprochen und deren Lösungswege offengelegt. Das Kapitel Probleme soll es der Leserschaft der Arbeit ermöglichen bei ähnlichen Problemen möglichst schnell zu einem Lösungsweg zu kommen und als Hilfestellung dienen.

9 Ausblick

Aufgrund der beliebig steigenden Komplexität der Herangehensweisen beim fuzzen gibt es bereits erste Baukastensysteme zum Bauen eigener Fuzzer. Von AFL++ selbst gibt es eine in RUST geschriebene Bibliothek LibAFL [46] zum Bauen modularer Fuzzer. Diese Art des Fuzzen erlaubt es dem Tester, noch schnellere Ausführgeschwindigkeiten zu erzielen. Mithilfe dieser Frameworks ist es möglich, Fuzzing komplett Plattformunabhängig durchzuführen. Zudem ist es möglich, den mit LibAFL gebauten fuzzer in ein gewünschtes Binary zu integrieren und somit natives Fuzzing zu ermöglichen.
Nachdem sich in dieser Arbeit mit AFL++ auseinandergesetzt wurde, soll sich in der auf dieser Arbeit aufbauenden Bachelorarbeit mit den alternativen Ansätzen des Fuzzing beschäftigt werden. Dazu gehören die Abwägungen von Netzwerk basierenden Fuzzern wie boofuzz und modernen Baukastensystemen wie der bereits genannten Fuzzing-Bibliothek LibAFL.

In einer zukünftigen Arbeit soll ein – an das untersuchte Binary – angepasster Fuzzer mithilfe des bereits angesprochenen Frameworks LibAFL umgesetzt werden. Hierbei soll die Performance der bereits existierenden Kampagne weiter optimiert werden.

Anhang A Listings

Listing 10: `example.pdf`: Synthax einer PDF Datei

```
%PDF-1.7
1 0 obj % entry point
<<
  /Type /Catalog
  /Pages 2 0 R
>>
endobj
%%EOF
```

Listing 11: Exemplarisches C Programm, welches das Mitgeben eines Startparameters benötigt

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
    if (argc != 2) {
        printf("Usage: %s <string>\n", argv[0]);
        return 1;
    }

    // Print the input string passed as a command-line argument
    printf("You entered: %s\n", argv[1]);

    return 0;
}
```

Listing 12: Bauen des QEMU-Supports von AFL für ARM Binarys

```
$ cd AFLplusplus/qemu_mode
$ CPU_TARGET=arm ./build_qemu_support.sh
```

Listing 13: Zeigt die Ausgabe des `ldd` commands auf mmapp. `ldd` ist ein Programm zur Ausgabe von benötigten Bibliotheken eines Binarys.

```
ldd mmapp
        libX11.so.6 => not found
        libXmuu.so.1 => not found
        libappmsg.so => not found
        librt.so.1 => /usr/lib/librt.so.1 (0x3f7e0000)
        libMoviePlayer.so => not found
        libieumultiM.so => not found
        libdatamgr.so => not found
        libIEUSDK.so => not found
        libcmn.so => not found
        libudev.so.0 => not found
        libQtWebKit.so.4 => not found
        libQtGui.so.4 => not found
        libQtCore.so.4 => not found
        libpthread.so.0 => /usr/lib/libpthread.so.0 (0x3f7c0000)
        libstdc++.so.6 => /usr/lib/libstdc++.so.6 (0x3f5c0000)
        libgcc_s.so.1 => /usr/lib/libgcc_s.so.1 (0x3f590000)
        libc.so.6 => /usr/lib/libc.so.6 (0x3f3f0000)
        /lib/ld-linux-armhf.so.3 => /usr/lib/ld-linux-armhf.so.3 (0
            ↪ x40000000)
        libm.so.6 => /usr/lib/libm.so.6 (0x3f390000)
```

Listing 14: Einstellen der Containerumgebung mit bwrap

```
$ bwrap \
    --clearenv \
    --die-with-parent \
    --bind root/ / \
    --setenv DISPLAY :10 \
    --setenv QT_X11_NO_MITSHM 1 \
    --setenv PATH "/usr/gnu/bin:/usr/local/sbin:/usr/local/bin:/bin:/
        ↪ sbin:/usr/bin:." \
    --proc /proc \
    --dev /dev \
    --dev-bind /tmp/ttyACM0 /dev/ttyACM0 \
    --uid 0 \
    --gid 0 \
    --chdir / \
    --unshare-all \
    --share-net \
    --cap-add CAP_SYS_ADMIN \
    --ro-bind /lib64 /lib64 \
    --setenv AFL_PRELOAD ./sockfuzz.so \
    /afl-fuzz -Q -i in/ -o out/ -t 50000 -- /app/mmapp @@
```

Literatur

[1] Kaspersky. *Übersicht über IoT Ausnutzung*. URL: https://securelist.com/iot-threat-report-2023/110644/ (besucht am 11. 03. 2024).

[2] Greenbone. *Greenbone Schwachstellenscanner*. URL: https://www.greenbone.net/en/products/ (besucht am 11. 03. 2024).

[3] iisys. *System and Network Security*. URL: https://www.iisys.de/forschung/forschungsgruppen/system-and-network-security/ (besucht am 11. 03. 2024).

[4] OWASP. *Path Traversal*. URL: https://owasp.org/www-community/attacks/Path_Traversal (besucht am 11. 03. 2024).

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

[6] Michal Salewski. *AFL*. URL: https://lcamtuf.coredump.cx/afl/ (besucht am 11. 03. 2024).

[7] 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).

[8] dataprot. *IoT Statistiken 2024*. URL: https://dataprot.net/statistics/iot-statistics/ (besucht am 11. 03. 2024).

[9] TIOBE. *Index der Popularität der Programmiersprachen*. URL: https://www.tiobe.com/tiobe-index/ (besucht am 11. 03. 2024).

[10] mend.io. *Statistiken über die Sicherheit von Programmiersprachen*. URL: https://www.mend.io/most-secure-programming-languages/ (besucht am 11. 03. 2024).

[11] NVD. *Kategorisierung der Schwere einer Sicherheitslücke mittels CVSS*. URL: https://nvd.nist.gov/vuln-metrics/cvss (besucht am 11. 03. 2024).

[12] Bruno Dorsemaine. *Internet of things: a definition & taxonomy*. 2015. URL: https://www.researchgate.net/profile/Bruno-Dorsemaine/publication/282218657_Internet_of_Things_A_Definition_Taxonomy/links/560833be08ae8e08c0946052/Internet-of-Things-A-Definition-Taxonomy.pdf (besucht am 11. 03. 2024).

[13] Sharp NEC. *NEC Control Command Manual*. URL: https://assets.sharpnecdisplays.us/documents/miscellaneous/pj-control-command-codes.pdf (besucht am 11. 03. 2024).

[14] Sebastian Peschke. *binaryProt.py Script zur Analyse der Ausgaben des Netzwerkprotokolls*. URL: https://github.com/ItsMagick/Praxis-Bachelor-Listings/blob/main/binaryProt.py (besucht am 11. 03. 2024).

[15] Sebastian Peschke. *Netzwerk Protokoll Befehle und Antworten*. URL: https://gitlab.iisys.de/isb/beam-me-up-scotty/-/blob/fuzz-test/docs/BinaryCommandsAndResponses.md?ref_type=heads (besucht am 11. 03. 2024).

[16] 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).

[17] AFLplusplus. *AFL input Dateiendung*. URL: https://github.com/AFLplusplus/AFLplusplus/blob/stable/src/afl-fuzz.c#L2094-L2098 (besucht am 11. 03. 2024).

[18] 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).

[19] The QEMU Project. *System und User Space Emulation*. URL: https://www.qemu.org/docs/master/about/index.html (besucht am 11. 03. 2024).

[20] Michal Zalewski. *Funktionsweise des AFL QEMU Modus*. URL: https://github.com/google/AFL/blob/master/docs/technical_details.txt#L490-L524 (besucht am 11. 03. 2024).

[21] Gal Tashma. *Codepfadinformationen im AFL QEMU Mode*. URL: https://galtashma.com/posts/how-fuzzing-with-qemu-and-afl-work (besucht am 11. 03. 2024).

[22] AFL. *Fuzzing a network service*. URL: https://aflplus.plus/docs/best_practices/#fuzzing-a-network-service (besucht am 11. 03. 2024).

[23] AFLplusplus. *Risiken des Fuzzing*. URL: https://github.com/AFLplusplus/AFLplusplus/blob/stable/docs/fuzzing_in_depth.md#common-sense-risks (besucht am 11. 03. 2024).

[24] Sebastian Pahl. *Tabelle der Funktionen des Netzwerkprotokolls und ihre Handler*. URL: https://gitlab.iisys.de/isb/beam-me-up-scotty/-/blob/fuzz-test/scripts/table.py?ref_type=heads (besucht am 11. 03. 2024).

[25] The Linux Information Project. *Root Filesystem Definition*. URL: https://www.linfo.org/root_filesystem.html (besucht am 11. 03. 2024).

[26] ArchWiki. *procfs*. URL: https://wiki.archlinux.org/title/Procfs (besucht am 11. 03. 2024).

[27] kernel.org. *sysfs*. URL: https://www.kernel.org/doc/html/latest/admin-guide/sysfs-rules.html (besucht am 11. 03. 2024).

[28] The Linux Foundation. *Filesystem Hierarchy Standard*. URL: https://refspecs.linuxfoundation.org/FHS_3.0/fhs-3.0.html#runRuntimeVariableData (besucht am 11. 03. 2024).

[29] Sebastian Peschke. *chroot-fuzz-bin*. URL: https://gitlab.iisys.de/isb/beam-me-up-scotty/-/blob/fuzz-test/fuzz/bin/chroot-fuzz-bin?ref_type=heads (besucht am 11. 03. 2024).

[30] Containers. *bubblewrap*. URL: https://github.com/containers/bubblewrap (besucht am 11. 03. 2024).

[31] AFLplusplus. *Fuzzing Prozess*. URL: https://github.com/AFLplusplus/AFLplusplus/tree/stable/docs (besucht am 11. 03. 2024).

[32] AFLplusplus. *Benutzen des QEMU Mode*. URL: https://github.com/AFLplusplus/AFLplusplus/tree/stable/qemu_mode (besucht am 11. 03. 2024).

[33] lolcads. *Whas ist Desocketing*. URL: https://lolcads.github.io/posts/2022/02/libdesock/ (besucht am 11. 03. 2024).

[34] linuxbase. *Libc Entrypoint*. URL: https://refspecs.linuxbase.org/LSB_3.1.1/LSB-Core-generic/LSB-Core-generic/baselib---libc-start-main-.html (besucht am 11. 03. 2024).

[35] AFLplusplus. *Sicherstellung der Einzigartigkeit des Korpus mit afl-cmin*. URL: https://github.com/AFLplusplus/AFLplusplus/blob/stable/docs/fuzzing_in_depth.md?plain=1#L418-L443 (besucht am 11. 03. 2024).

[36] Sebastian Peschke. *Exemplarischer TCP Server*. URL: https://github.com/ItsMagick/Praxis-Bachelor-Listings/blob/main/example-tcp-server.c (besucht am 11. 03. 2024).

[37] Sebastian Peschke. *Skript zum Cross-kompilieren eines C-Programms*. URL: https://github.com/ItsMagick/Praxis-Bachelor-Listings/blob/main/compile-example.sh (besucht am 11. 03. 2024).

[38] docker hub. *ARM32v7 GCC Container*. URL: https://hub.docker.com/r/arm32v7/gcc/ (besucht am 11. 03. 2024).

[39] AFLplusplus. *QEMUs persistent mode*. URL: https://github.com/AFLplusplus/AFLplusplus/blob/stable/qemu_mode/README.persistent.md (besucht am 11. 03. 2024).

[40] AFL plus plus. *Verwendung mehrerer CPU-Kerne*. URL: https://github.com/AFLplusplus/AFLplusplus/blob/stable/docs/fuzzing_in_depth.md#c-using-multiple-cores (besucht am 11. 03. 2024).

[41] AFLplusplus. *Strategien der Input Mutation*. URL: https://github.com/AFLplusplus/AFLplusplus/blob/stable/docs/afl-fuzz_approach.md#stage-progress (besucht am 11. 03. 2024).

[42] AFLplusplus. *Interpretation der Outputs von AFL*. URL: https://github.com/AFLplusplus/AFLplusplus/blob/stable/docs/afl-fuzz_approach.md#interpreting-output (besucht am 11. 03. 2024).

[43] GNU. *Installation von GCC*. URL: https://gcc.gnu.org/install/ (besucht am 11. 03. 2024).

[44] Ububtu. *Ubuntu Archiv GCC*. URL: http://archive.ubuntu.com/ubuntu/pool/universe/g/ (besucht am 11. 03. 2024).

[45] S. Pahl. *Gitlab des Projekts*. URL: https://gitlab.iisys.de/isb/beam-me-up-scotty (besucht am 11. 03. 2024).

[46] 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).

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 Arbeit *Fuzzing von IoT Binarys für Ruhm und Ehre* beschäftigt sich mit dem Fuzzing eines Netzwerkprotokolls und der damit verbundenen Analyse eines bereits kompilierten Programms. Nach der Einordnung und Kategorisierung des zu untersuchenden Programms werden grundlegende Begriffe und Herangehensweisen der Programmanalyse und des Fuzzing mit *AFL* erarbeitet.

Folgend wird eine detaillierte Übersicht über den in dieser Arbeit verwendeten Fuzzer *AFL++* gegeben. Hierzu werden die verwendete Strategie und die verbauten Technologien des Fuzzers beschrieben. Anand der Funktionsweise des Fuzzers werden die mit ihm verbundenen Risiken offengelegt.

Im folgenden praktischen Teil der Arbeit werden die entwickelten und verwendeten Testumgebungen erklärt. Dazu werden die Unterschiede der beiden entwickelten Ansätze aufgezeigt und ein Fazit zur Benutzbarkeit der Testumgebungen gezogen.

Im Anschluss wird die Vorgehensweise zum Fuzzen eines Netzwerkprotokolls auf Applikationsebene erläutert. Dazu gehört das Implementieren einer Preload-Library zum Überschreiben von Syscalls, die für die Kommunikation über das Netzwerk verantwortlich sind. Es werden drei Ansätze zur Leistungssteigerung beschrieben, indem die von AFL++ bereitgestellten Funktionen zur Minimierung der Eingaben, zur Verwendung des Persistent Modes und zur Skalierung der Kampagne über mehrere CPU-Kerne genutzt werden.

Abschließend werden die vom Fuzzer erzeugten Ausgaben der Statusanzeige des laufenden Fuzzers erklärt. Die erzeugten Dateien und deren Inhalt werden anschließend ausgewertet und der Erfolg des Fuzzingprozesses ermittelt.

Zum Abschluss werden die während der Arbeit aufgetretenen Probleme offengelegt und die Arbeit reflektiert. Der verwendete Fuzzer kann als Erweiterung dieser Arbeit durch die Bibliothek *LibAFL* erweitert und den Anforderungen der Arbeit angepasst werden.