OSTBAYERISCHE
TECHNISCHE HOCHSCHULE
REGENSBURG

BACHELORARBEIT

Stefan Schönhärl

Automatisiertes Reverse Engineering von Nachrichten aus automobilem CAN-Traffic

16. März 2020

Fakultät:	Informatik und Mathematik
Studiengang:	Bachelor Technische Informatik
Betreuung:	Prof. Dr. Rudolf Hackenberg
Zweitbegutachtung:	Prof. Dr. Frank Hermann

Inhaltsverzeichnis

Abkürzungsverzeichnis

UNECE United Nations Economic Commission for Europe

ISO International Organization for Standardization

GRVA Working Party on Automated/Autonomous and Connected Vehicles

ASIL Automotive Safety Integrity Level

CAN Controller Area Network

CPU Central Processing Unit

DoIP Diagnostics over Internet Protocol

ECUs Electronic Control Units

LSB Least Significant Bit

DLC Data Length Code

V Volt

1 Einleitung

Historisch gesehen waren Automobile lange Zeit größtenteils mechanische Produkte. Dies veränderte sich besonders in den letzten Jahren rapide wegen den vielen Entwicklungen und Verbesserungen im Informatik Sektor. Da aufgrund des geringen Platzes in Automobilen keine modernen Standard PCs verbaut werden können, ist besonders das Fachgebiet der „Embedded Systems“ von Bedeutung.

In diesem Gebiet werden eigenständige kleine Rechner, sogenannte Mikrocontroller, eingesetzt, die eine bestimmte Aufgabe, wie zum Beispiel das Auslesen eines Sensors, gut lösen können. Heute werden viele verschiedene Mikrocontroller in ein Auto eingebaut, um die diversen Aufgabenstellungen zu bewältigen. Die Ergebnisse der einzelnen Mikrocontroller werden miteinander durch verschiedene Protokolle ausgetauscht.

Ein Beispiel für eine moderne Entwicklung bei Fahrzeugen ist das „vernetzte Fahren“. Das Bundesministerium für Verkehr und digitale Infrastruktur definiert dieses als „Kommunikation zwischen Fahrzeugen […] sowie zwischen Fahrzeug und Infrastruktur“ [BMV]. Dabei werden Daten von einem Verkehrsteilnehmer erfasst und anschließend an die anderen Teilnehmer weitergeleitet, damit diese zum Beispiel auf erhöhtes Verkehrsaufkommen, oder Unfälle hingewiesen werden können.

Sowohl der Datenaustausch zwischen den Verkehrsteilnehmern, als auch zwischen einzelnen Komponenten im Fahrzeug muss aus der Sicht der IT Security zwangsläufig sicher sein, da unsichere Konzepte, oder Implementierungen im Fahrzeug mitunter schwerwiegende Folgen für die Sicherheit der Insassen haben können.

1.1 Bedrohungen

Im Folgenden werden Beispiele gezeigt, die verdeutlichen, welche Auswirkungen aus der Sicht der IT Security unsichere Komponenten im Fahrzeug haben können.

Beim vernetzten Fahren könnte ein Angreifer zum Beispiel die Nachricht abfangen und verändern, welche von einem Fahrzeug versendet wurde. Dieses Vorgehen bezeichnet man allgemein als „man-in-the-middle“ Angriff. Die ursprüngliche Nachricht könnte zum Beispiel beinhalten, dass hinter einer scharfen Kurve ein Unfall passiert ist. Wenn dieser Inhalt nun verändert wird und nicht mehr auf die Unfallstelle hingewiesen wird, könnte ein anderes Auto ungebremst in die Unfallstelle einfahren.

Des weiteren könnten Angreifer private Daten auslesen, indem eine der zahlreichen Infotainment Schnittstellen, wie USB, manipuliert wird. Viele dieser Schnittstellen sind vorgesehen, um sie mit dem Mobiltelefons des Fahrers zu verbinden. Wenn diese nun mit einer manipulierten Schnittstelle verbunden werden, könnten persönliche Daten, die auf dem Mobiltelefon gespeichert sind, ausgelesen und an den Angreifer weitergeleitet werden. So eine Schwachstelle stellt zwar keine unmittelbare Bedrohung der körperlichen Unversehrtheit der Insassen dar, aber es können beispielsweise Kreditkarten Informationen gestohlen werden.

Ein Beispiel für einen Angriff, der sich in einem Fahrzeug intern abspielt, ist, dass der Angreifer die Kommunikationswege zwischen Mikrocontroller und Sensoren blockiert. Dies nennt man einen „Denial-of-Service“ Angriff. Wird der Kommunikationsweg vollständig lahmgelegt, werden sicherheitskritische Informationen von Sensoren nicht mehr an die zuständigen Mikrocontroller weitergegeben. Dies ist besonders gefährlich bei Technologien, wie dem Notbremsassistenten, der hier nicht mehr auf Gefahren reagieren kann.

1.2 Sicherheitstests und Normierung

Die im vorigen Kapitel genannten Beispiele von Bedrohungen der IT Sicherheit machen deutlich, dass es moderne Sicherheitslücken gibt, die geschlossen werden müssen. Das Problem, welches hier auftritt, ist, dass erst herausgefunden werden muss, wo eine Schwachstelle auftritt, die ausgenutzt werden kann. Deshalb sind Sicherheitstests von Soft- und Hardware äußerst wichtig.

Besonders wichtig bei diesen Test ist, dass diese verlässlich und umfangreich durchgeführt werden. Dafür sind international gültige Standards notwendig.

Wichtige Organisationen, die solche Standardisierungen festlegen, sind die United Nations Economic Commission for Europe (UNECE) und die International Organization for Standardization (ISO).

Der Teil der UNECE, der sich unter anderem um die Sicherheit in vernetzten Fahrzeugen kümmert, heißt „Working Party on Automated/Autonomous and Connected Vehicles (GRVA)“. Von dieser wurde beispielsweise das Dokument „Draft Recommendation on Cyber Security of the Task Force on CyberSecurity and Over-the-air issues of UNECE WP.29 GRVA“ [GRV] herausgegeben. In diesem wird behandelt, welche verschiedenen Angriffspunkte bei vernetzten Fahrzeugen vorhanden sind. Des weiteren werden Möglichkeiten aufgeführt, um die beschriebenen Gefahren zu verhindern, oder zumindest zu erkennen.

Die ISO hingegen entwickelt Normen in allen wichtigen Bereichen. Eine der wichtigsten Normen im Automotive Bereiche ist „ISO 26262:2018 Road vehicles — Functional safety“ [ISOb]. Diese beschreibt die Funktionale Sicherheit in Fahrzeugen. Funktionale Sicherheit ist das Erkennen von potentiell gefährlichen Zuständen, was zur Aktivierung von schützenden oder korrigierenden Vorrichtungen oder Mechanismen führt, um das Auftreten von gefährlichen Situationen zu

verhindern, oder um eine Milderung der Konsequenzen der gefährlichen Situationen bereitzustellen [IEC].

Besonders wichtig bei dieser Norm ist die Festlegung der Automotive Safety Integrity Level (ASIL). Bedrohungen werden in diese ASIL eingeordnet, indem sie in drei Unterpunkten bewertet werden. Diese sind:

- Severity
- Controllability
- Exposure

Die Bewertung der Severity bezieht sich auf die Auswirkung, die die jeweilige Bedrohung darstellt, falls sie eintritt. Controllability bewertet, wie schwer es ist die Gefahr wieder unter Kontrolle zu bringen und der Punkt Exposure beschreibt, wie oft die Bedrohung eintritt. Aus diesen Kriterien wird der jeweiligen Bedrohung ein ASIL Level von A-D zugeordnet.

Eine weitere ISO Norm ist die „ISO/SAE DIS 21434:2020 Road vehicles — Cybersecurity engineering“ [ISOc]. Diese ist noch nicht offiziell erschienen, befindet sich aber gerade in Entwicklung. Wie im Namen bereits beschrieben, befasst sich diese Norm nicht mit der Funktionellen Sicherheit allgemein, sondern speziell mit der Cybersecurity. Die finale Version soll noch im Jahr 2020 erscheinen.

1.3 Automatisiertes/Manuelles testen

Diese im vorherigen Kapitel beschriebenen Normen definieren Sicherheitstests, die manuell oder automatisiert durchgeführt werden.

Die generelle Ausgangslage ist, dass Sicherheitstest manuell durchgeführt werden müssen. Allerdings ist damit ein hoher Zeitaufwand für Entwickler oder Tester verbunden. Des weiteren werden bei diesen oft Fehler wegen repetitiven Abläufen gemacht.

Eine weitaus geschicktere Lösung ist es Tests zu automatisieren. Dabei kann oft nicht nur die Geschwindigkeit erhöht werden, mit der getestet wird, sondern es werden auch die menschliche Fehler vermieden. Falls ein automatischer Test fehlerhaft ist, kann dieser schneller gefunden werden, da alle Testergebnisse falsch sind und deshalb schneller auffallen, als bei einem einzelnen Fehler. Für die Automatisierung müssen Kriterien entwickelt werden, die im Zuge der Automatisierung bewertet werden.

Für das Entwickeln von automatischen Tests muss anfangs Entwicklungszeit investiert werden, was es manchmal kosteneffektiver macht nur manuell zu testen. Da aber in der Automobil Branche Komponenten so sicher wie möglich sein müssen, zahlt sich diese initiale Investition oft aus.

Allerdings kann eine sehr hohe Komplexität der Testautomatisierung dazu führen, dass sie von den Entwicklern nicht erreicht werden kann.

1.4 Schwachstellenanalyse und Penetrationstests

Penetrationstests sind eine bestimmte Art von Sicherheitstests. Verschiedene Penetrationstests können manuell, oder automatisch durchgeführt werden. Speziell im Softwarebereich sind Penetrationstests ein sehr gutes Werkzeug, um zu testen, wie sicher die produzierte Software ist.

Vor einem Penetrationstest wird meist eine Sicherheitsanalyse durchgeführt. Diese ist dafür da, um potentielle Schwachstellen im anzugreifenden System zu finden. Mit den Ergebnissen aus diesem Schritt können auf die gefundenen Schwachstellen zugeschnittene Penetrationstests entwickelt und ausgeführt werden.

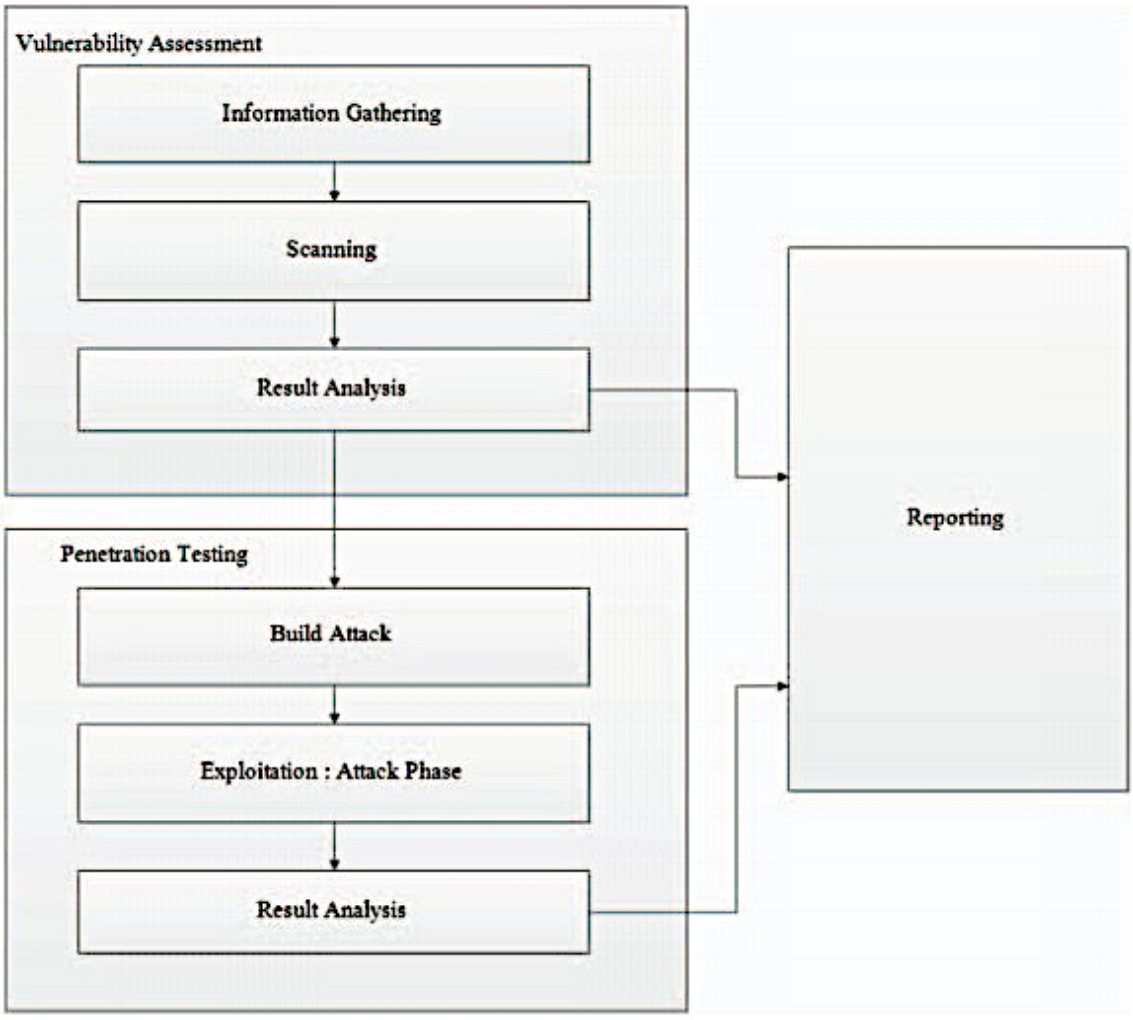

Abbildung 1: Schritte einer Schwachstellenanalyse und eines Penetrationstests [SA16]

In Abbildung 1 ist der Ablauf einer Schwachstellenanalyse mit anschließendem Penetrationstest dargestellt.

Im Abschnitt der Schwachstellenanalyse werden zuerst über das zu untersuchende Ziel Informationen gesammelt. Anschließend wird in der „Scanning“ Phase das Ziel auf unsichere Komponenten, beispielsweise offene Ports, überprüft. Mit den gefundenen Ergebnissen der vorgehenden Schritte wird schlussendlich die Ergebnisanalyse angefertigt, welche den Abschluss der Schwachstellenanalyse darstellt.

Mit den Ergebnissen der Ergebnisanalyse kann begonnen werden angemessene Angriffe zu erstellen und durchzuführen. Die Ergebnisse aus der Schwachstellenanalyse und des Penetrationstests werden abschließend dokumentiert und an den Auftraggeber weitergeleitet.

1.5 Penetrationstest mithilfe von Reverse Engineering

Ein sehr nützliches Werkzeug für einen Penetrationstest kann das Reverse Engineering sein. Dabei kann Reverse Engineering in die „Scanning“ Phase der Sicherheitsanalyse eingeordnet werden.

Mit einem erfolgreichen Ergebnissen aus dem Reverse Engineering wird die Phase der Entwicklung des Penetrationstests vereinfacht, da das System, das hinter dem Angriffsziel steckt, vollkommen nachvollzogen werden kann. Wird beispielsweise der Code einer Applikation vollständig durch Reverse Engineering zurückgewonnen, kann mit bloßer Analyse nach Schwachstellen im Code selbst nach Fehlen gesucht werden, anstatt zufällig nach Sicherheitslücken zu suchen.

Wird ein Penetrationstest ohne Wissen über den Quellcode durchgeführt, wird dies „Black-Box Testing“ genannt. Das Gegenteil dazu, wenn der Quellcode des zu testenden Programmes vollständig vorliegt, wird analog „White-Box Testing“ genannt. Folglich ermöglicht das Reverse Engineering einem Penetrationstester das Verwandeln eines Black Box Tests in einen White Box Test.

1.6 Motivation

Ziel dieser Arbeit ist es eines der im Automobil am weitesten verbreiteten Bussysteme, nämlich Controller Area Network (CAN), zu untersuchen. Es soll automatisiert herausgefunden werden, welche Felder in einer CAN Nachricht zusammen gehören. Daraus folgt der Titel:„Automatisiertes Reverse Engineering von Nachrichten aus automobilem CAN-Traffic“

2 Grundlagen

2.1 Kommunikation in Fahrzeugen

Für die Kommunikation in modernen Fahrzeugen werden verschiedene Technologien eingesetzt.

Besonders Ethernet hat in den vergangenen Jahren im Automotive Bereich viele Einsatzbereiche hinzugewonnen. Der größte Vorteil von Ethernet gegenüber anderen Technologien ist die großen Datenrate von bis zu 1GB pro Sekunde [HMVV13].

Diese Entwicklung begann mit der Einführung von Diagnostics over Internet Protocol (DoIP). DoIP ist definiert in ISO 13400 Road vehicles — Diagnostic communication over Internet Protocol (DoIP) [ISOa].

Der nächste Einsatz für Ethernet wurde mit Erweiterung und Einführung von Infotainment- und Fahrerassistenzsystemen geschaffen. Dazu gehören zum Beispiel Kameras, die zur Einpark- oder Rückfahrhilfe benutzt werden. Ebenso gehören dazu Services, die das Internet nutzen, wie Navigationssysteme.

Eine der neusten Entwicklungen ist Ethernet als Backbone für das gesamte Fahrzeug zu verwenden. Dies bedeutet, dass alle anderen Protokolle mit Ethernet verbunden werden [HMVV13].

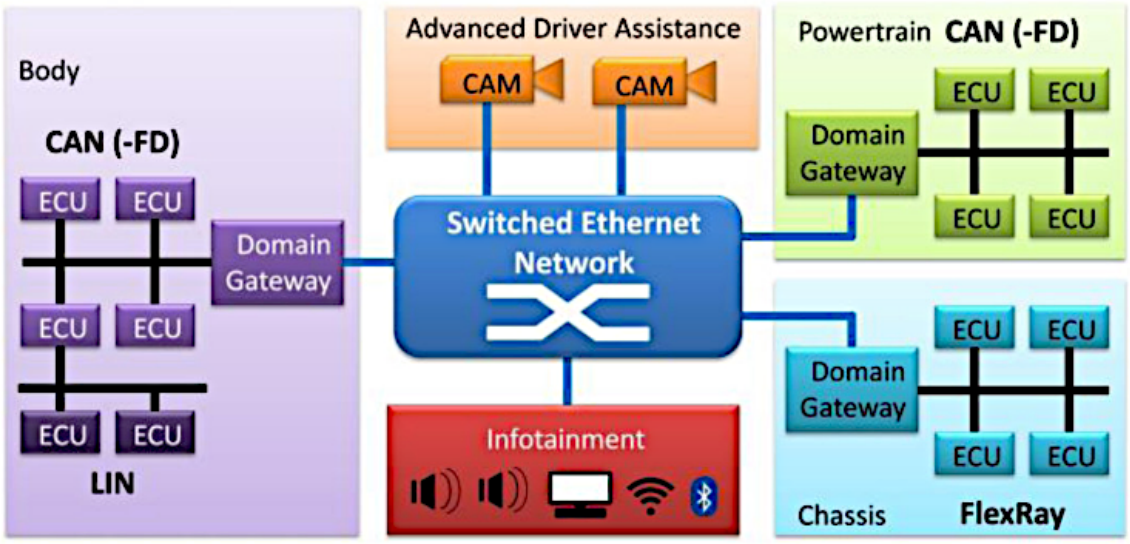

Abbildung 2: Beispielhaftes Fahrzeugnetzwerk mit Ethernet als Backbone [ATE]

In Abbildung 2 ist ein solches Fahrzeugnetzwerk mit Ethernet als Backbone zu sehen.

Eine weitere Technologie, die in Automobilen sehr viel Anwendung findet ist FlexRay. Diese überzeugt durch eine relativ hohe Datenrate von bis zu 10MB pro Sekunde je Kanal. Zwei Kanäle sind vorhanden, die entweder verschiedene Daten verschicken können, um die Datenrate auf 20MB pro Sekunde zu erhöhen oder gleiche Daten, um zusätzliche Redundanzen einzubauen. Ferner kann es als Protokoll in Echtzeitsystemen eingesetzt werden, da Prioritäten durch Einteilung in Zeitslots vergeben werden können [SJ08].

Diese Vorteile machen FlexRay zu einem sehr guten Protokoll für Sicherheitskritische Systeme und modernen Technologien, wie „Steer-by-Wire“. Bei diesem wird die Lenkung in einem Fahrzeug, welches normalerweise mechanisch oder hydraulisch abläuft, durch ein elektrisches System ersetzt [BBM07].

Ein Ziel von FlexRay war es das 1983 von Bosch entwickelte CAN Protokoll abzulösen. Allerdings wird CAN immer noch wegen der geringen Kosten in Systemen eingesetzt, deren Anforderungen nicht so hoch sind, wie für FlexRay.

Mit CAN werden Datenraten, je nach Konfiguration, von bis zu 1MB pro Sekunde erreicht. Wie auch bei FlexRay ist auch CAN ein Protokoll, das in Echtzeitsystemen eingesetzt werden kann, da eine Priorisierung von Nachrichten möglich ist.

Des weiteren ist CAN sehr resistent gegenüber Störeinflüsse. Das liegt daran, dass der CAN Bus aus zwei Leitungen besteht, CAN High und CAN Low. CAN High kann zum Beispiel eine Spannungen von 2.5 bis 5 Volt (V) übertragen, wobei 5V für eine „0“ und 2.5V für eine „1“ steht. CAN Low dagegen überträgt Spannungen von 0 bis 2.5V, wobei 0V für eine „0“ und 2.5V für eine „1“ steht.

Um die übertragen Zahl zu interpretieren wird nun die Differenz aus den beiden Spannungen errechnet, also für eine „0“ ist die Differenz 5 und für eine „1“ ist sie 0. Stellt man sich nun vor ein elektrisches Feld wirkt sich auf beide Leitungen aus und erhöht die Spannung in diesen um je 1V wird sich das nicht auf die Differenz auswirken.

2.2 Penetration Testing

Ein Penetrationstest ist ein Test, der Firmen Sicherheitslücken in ihren Produkten/-Systemen aufzeigt. Dies wird erreicht, indem diese Sicherheitslücken von beauftragten Testern gefunden und ausgenutzt werden.

Verschiedene Parameter bestimmen, welche Penetrationstests angewendet werden können. Zum Beispiel sind mit physikalischem Zugriff auf das System mehrere Seitenkanalangriffe möglich. Diese sind eine Art Angriff, die nicht Ein-/Ausgaben des Systems direkt angreifen, sondern Nebeneffekte analysieren. Beispiele für Seitenkanalangriffe auf Software können Ausführungszeit des Programms, Energieverbrauch und Temperaturveränderungen sein.

Ein weiterer Parameter ist das Wissen des Penetrationstesters über das zu testende System. Folgende Arten von Software Tests sind mit bestimmtem Wissen möglich:

- Black Box
- White Box
- Grey Box

Beim Black Box Testing muss der Penetrationstest durchgeführt werden, ohne Wissen über die inneren Vorgänge des Systems. Um so ein System zu testen werden nur fundamentale Aspekte überprüft. Vorteile dieser Testart sind beispielsweise, dass ein Tester über keine Programmierkenntnisse besitzen muss. Des Weiteren können sehr große Codesegmente effizient getestet werden [KK+12].

Ein Beispiel für einen Black-Box Test ist das „Fuzzing“. Dieses ist ein automatischer Test, der einem System zufällige Daten als Eingabe übergibt und die Ausgaben auf fehlerhafte Ergebnisse überprüft.

Für einen White Box Test müssen alle Informationen über das zu Testende System bekannt sein. In der Softwareentwicklung bedeutet dies, dass beispielsweise der Quellcode des Programms vorliegen muss. Ein Tester kann nun im Quellcode selbst nach Fehlern und Schwachstellen suchen. Vorteilhaft beim White Box Testing ist, dass mit der Analyse des Quellcodes durch einen erfahrenen Penetrationstester eine sehr hohe Anzahl an Schwachstellen gefunden werden kann [KK+12].

Beispiele für Fehler bei der Implementierung, die bei der Analyse des Quellcodes gut erkennbar sind, sind Buffer Overflows und Format String Schwachstellen.

Ein Buffer Overflow ist eine Sicherheitslücke, die auftritt, wenn Nutzereingaben benutzt werden um einen Buffer zu füllen und dabei nicht überprüft wird, ob über die definierte Grenze des Buffers hinaus geschrieben wird. Ein Buffer Overflow hat im besten Fall nur ein Abstürzen des Programms zur Folge. Im schlimmsten Fall kann vom Angreifer gewünschter Code ausgeführt werden.

Ebenfalls kann eine Format String Schwachstelle auftreten, wenn das Programm Nutzereingaben erwartet und diese Eingaben anschließend ungeprüft wieder ausgibt. Dies kann dazu führen, dass der Angreifer den Inhalt des Stacks auslesen kann.

Das Grey Box Testing stellt einen Mittelweg zwischen Black und White Box Testing dar. Bekannt ist nicht der gesamte Quellcode, sondern nur fundamentale Aspekte des Systems, wie interne Datenstrukturen und Algorithmen [KK+12].

2.3 Reverse Engineering

Allgemein ist Reverse Engineering ein Prozess, bei dem ein konstruiertes Objekt so dekonstruiert wird, dass die innersten Details, wie Design und Architektur, aufgedeckt werden [Eil13a]. Reverse Engineering ist nicht auf Hard- oder Software beschränkt und findet somit in verschiedenen technischen Bereichen, wie Maschinenbau, Informatik, Elektrotechnik, etc., Anwendung.

In der Softwareentwicklung speziell wird mit Reverse Engineering der Fall beschrieben aus einem Programm, für das kein Quellcode vorhanden ist, eben diesen Quellcode zurückzugewinnen.

Um dies zu erreichen kann versucht werden das gewünschte Programm nachzubauen. Jedoch ist dies ein sehr aufwändiger Prozess. Deshalb gibt es spezielle Programme, mit denen der Vorgang des Reverse Engineering vereinfacht wird. Die Aufgabenbereiche dieser Programme können unterteilt werden in:

- Disassembler
- Debugger
- Decompiler

Disassembler können aus der Binärdatei eines Programms den zugehörigen Assembler Code rekonstruieren. Dies wird erreicht, indem die Instruktionen, die die Central Processing Unit (CPU) aus der Binärdatei erhält, dekodiert und, anstatt diese Instruktionen auszuführen, in eine textuelle Darstellung gebracht werden [Eil13b].

Debugger sind Programme, die den Assembly Code eines Programms benötigen. Durch diesen kann schrittweise iteriert werden, um den Ablauf des Programms nachvollziehen zu können. Viele Debugger zeigen dem Entwickler währenddessen die Inhalte der CPU Register, den Inhalt des Speichers und den aktiven Stack Bereich [Eil13c].

Zuletzt gibt es noch Decompiler. Sie sind vergleichbar mit Disassembler, jedoch ist die Programmiersprache nach dem Decompile Prozess nicht Assembler, sondern eine Hochsprache, wie C. Allerdings gibt es Informationen, die beim ursprünglichen Kompilieren des Programms verloren gehen, wie Namen von Variablen, die nicht wiederhergestellt werden können [Eil13d]

Neben dem Anwendungsbereich für Penetrationstests gibt es weitere Beispiele für Software Reverse Engineering. Bei der Abwehr von Schadsoftware, wie Trojaner oder Viren, ist es wichtig zu verstehen, wie das jeweilige Programm funktioniert. Ist das Reverse Engineering der Software erfolgreich, können Wege gefunden werden, wie diese wieder restlos entfernt werden kann, oder auch welche Auswirkungen noch nicht identifiziert wurden. Zu diesen Auswirkungen zählt beispielsweise die selbständige Verbreitung des Virus im Netzwerk des infizierten Geräts.

3 Herleitung der Metriken

Mit dieser Arbeit wird versucht zu Reverse Engineeren, welche Teile einer CAN Nachricht zusammengehören. Wichtig ist dies für Penetrationstester, um zu bewerten, ob der CAN-Bus ein Angriffsziel für wirklich Angreifer sein kann, die Informationen aus diesem nutzen, um die Sicherheit im Fahrzeug zu gefährden. Die Automatisierung dieser Aufgabe ist wünschenswert, da damit die Verarbeitung der Ergebnisse sehr effizient ist.

Die eingesetzten Kriterien, mit denen diese Aufgabe bewältigt werden soll, sind die Metriken und werden in diesem Kapitel erklärt. Diese Metriken werden aus den einzelnen Feldern einer CAN Nachricht gewonnen. Auch diese werden im Anschluss beschrieben.

3.1 CAN

3.1.1 Protokoll

Für die Analyse der Datenfelder ist es nötig, dass der Aufbau einer CAN Nachricht bekannt ist.

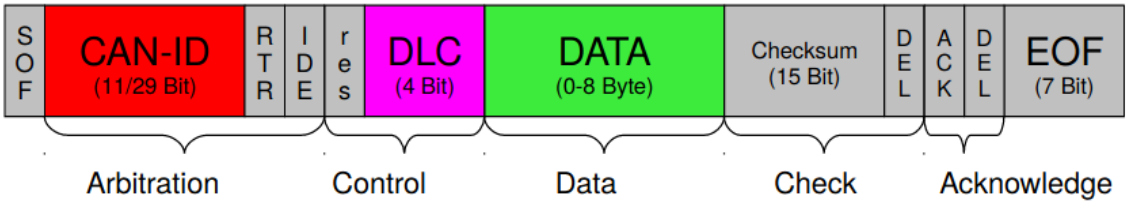

Abbildung 3: Standard CAN Frame (siehe [Har])

In Abbildung 3 ist der Frame einer CAN Nachricht zu sehen. Es werden anschließend nur die wichtig Abschnitte des Frames erklärt.

3.1.1.1 CAN-ID

Die CAN-ID hat einige Aufgaben, zum Beispiel kann man damit Identifizieren, welches Gerät den vorliegenden Frame verschickt hat, oder welches Gerät zuerst seine Nachricht senden darf. Diese Vorgehensweise wird „Arbitrierung“ genannt.

Wie in Abbildung 3 zu sehen ist, wird die ID am Anfang des Frames verschickt.

Wenn nun mehrere Komponenten gleichzeitig versuchen eine Nachricht zu verschicken und zur selben Zeit ihre ID auf die Leitung legen, werden die höherwertige IDs überschrieben. Dies wird umgesetzt, indem alle IDs, die versuchen eine Nachricht zu senden, mit einem logischen Und verknüpft werden.

Anschließend an die Arbitrierung können alle Komponenten prüfen, welche ID auf dem Bus vorhanden ist. Falls es nicht die eigene ist, wird aufgehört zu senden und auf den nächsten möglichen Zeitpunkt gewartet.

Diese Art der Kollisionskontrolle macht CAN zu einem echtzeitfähigen Protokoll, da eine niedrige ID gleichbedeutend mit einer hohen Priorität ist.

Da für die ID 11 Bits vorgesehen sind ist es möglich 2^11 verschiedene IDs zu vergeben. Soll CAN in einem System eingesetzt werden, welches mehr IDs benötigt, kann eine erweiterte Adressierung benutzt werden, die die Stellen auf 29 Bits erhöht.

3.1.1.2 DLC

Im Data Length Code (DLC) Feld des CAN Frames steht die Information, wie viele der maximal 8 Byte an Daten im Datenfeld benutzt werden.

3.1.1.3 DATA

In diesem Feld werden Nutzdaten übertragen, deren Länge im DLC Feld angegeben ist.

Es wird meist mehr als nur ein Datenpaket pro Nachricht verschickt. Dabei ist zwischen den Paketen keine Kennzeichnung gegeben, wo ein Paket endet und wo ein anderes beginnt.

Daten, die hier versendet werden, kommen oft von Sensoren, die kontinuierlich Daten übermitteln. Zum Beispiel könnte ein Steuergerät im Auto die Radumdrehungen pro Minute versenden, oder die Temperatur im Motor, etc..

Da ein Steuergerät oftmals mehr als nur einen Wert gleichzeitig versenden muss, können auch Multiplexer Felder eingebaut werden, um den geringen Platz von nur 8 Byte geschickter zu nutzen, ohne mehr Platz zu verbrauchen. Solche Multiplexer sind oft relativ kurze Felder, die am Anfang des Datenfeldes zu finden sind und dem Empfänger der Nachricht mitteilen, wie der restliche Inhalt des Datenfeldes zu interpretieren ist.

3.1.2 Candump

Candump ist eine Funktion, die es Nutzern von Linux Distributionen erlaubt CAN Traffic eines bestimmten Interfaces auszulesen.

Diese Funktion wurde im CarSec Labor der OTH Regensburg benutzt, um den CAN Traffic eines Mercedes aufzuzeichnen und in der Datei „candump-2019-04-05_104847.log“ abzuspeichern.

```
(1554454127.044719) can0 1F3#00FF008600F072F3
(1554454127.044720) can0 122#8E41050A02000040
(1554454127.045838) can0 096#18AD882780029920
(1554454127.045839) can0 0B1#413B791A04980728
(1554454127.045840) can0 1EE#0F0F1F0000FCFF00
(1554454127.047055) can0 02F#A32C961F0020FF3F
(1554454127.048676) can0 31A#FFE7010B00000000
(1554454127.048677) can0 320#03E8010300000000
(1554454127.052051) can0 0A1#A1A7000000000000
(1554454127.052052) can0 50B#000B000200000000
(1554454127.053381) can0 0A3#4AA7000000000000
(1554454127.055632) can0 2B9#FE840246717A0700
(1554454127.055633) can0 094#C9BE800D2A7E8900
(1554454127.055633) can0 307#1F0000000F020000
(1554454127.057019) can0 02F#B12D961F0020FF3F
(1554454127.057020) can0 3E0#F4E8302A68300000
(1554454127.058352) can0 098#9A1800000000207F
(1554454127.058383) can0 03D#370B00000000FFFF
(1554454127.058384) can0 0AE#6FD30000000068FF
(1554454127.062216) can0 087#E9585E025E021038
(1554454127.063332) can0 369#2004200000000000
(1554454127.063333) can0 0B3#5B38000000070058
(1554454127.065648) can0 147#00005FFF00711A04
(1554454127.065649) can0 149#6E01
(1554454127.065650) can0 096#7BAF882780029920
(1554454127.065651) can0 0B1#483D791A04960728
```

Abbildung 4: candump-Auszug eines Mercedes

In Abbildung 4 ist ein Ausschnitt dieser Datei zu sehen.

Die Zahl in Klammern sind die Sekunden seit dem 1.Januar.1970 (Unix Zeit), „can0“ beschreibt das ausgewählte CAN Interface und das letzte Feld ist der übertragene CAN Frame. Dabei stehen die ersten drei Stellen vor dem # für die ID und alles dahinter für den Inhalt des DATA Feldes.

Restliche Felder des CAN Frames, wie die „Checksumme“, werden nicht mehr aufgeführt, da diese Frames schon auf eine fehlerfreie Übertragung geprüft wurden.

3.2 Metriken

3.2.1 Bitflips

Bitflips sind eine Metrik, die aus der Binärdarstellung der Payload Daten gewonnen wird. Bitflips werden errechnet, indem man die Daten einer Nachricht mit der darauffolgenden vergleicht. Dabei wird an jeder Stelle, an der ein Bit ’kippt’, also von 1 auf 0, oder von 0 auf 1 springt, ein Zähler erhöht. Dies geschieht von der ersten bis zur letzten Nachricht.

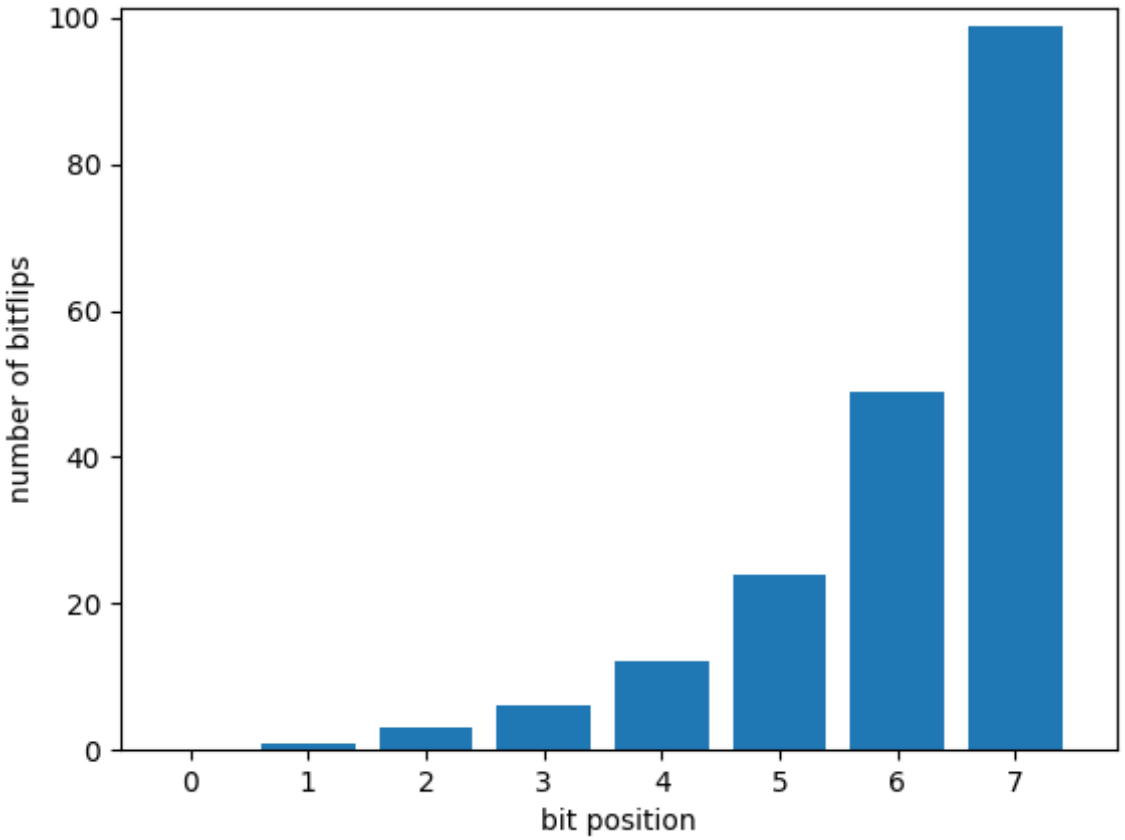

Abbildung 5: Bitflips beim Zählen von 0 bis 99

In Abbildung 5 ist zu sehen, wie oft die einzelnen Bits einer 8 Bit Zahl kippen, wenn von 0 bis 99 gezählt wird. Das Besondere bei dieser Metrik ist, dass es keinen Unterschied macht, wo angefangen wird zu zählen. Das heißt, dass von 0 aus gezählt werden kann oder auch von einer anderen beliebigen Zahl aus. Es wirkt sich ebenfalls nicht auf die Bitflips aus, ob nach oben oder unten gezählt wird.

Mit dieser Abbildung wird gezeigt, wie sich die Bitflips beim konstanten Erhöhen oder Verringern eines Zählers verhalten. Es ist zu sehen, dass das Least Significant Bit (LSB) doppelt sooft kippt, wie das nächst höherwertige Bit. Dieses wiederum kippt doppelt so oft, wie das nächst höherwertige.

Diese Eigenschaft macht die Bitflips Metrik sehr gut im Erkennen von Feldern, die einen Zähler beinhalten, da beim Aufzeichnen eine Parabolische Kurve entsteht, wie in Abbildung 5 zu erkennen ist.

Dies macht die Bitflips zu einem wichtigen Werkzeug CAN Daten zu analysieren. Doch es gibt auch einige Fälle, in denen Bitflips nur schwer eine Aussage darüber treffen können, ob Felder zusammengehören. Diese werden im Anschluss erläutert.

3.2.1.1 Fehlerquellen

Ein Fall, der mit den Bitflips schwer zu erkennen ist, macht sich bemerkbar, wenn bei einem Datenfeld der Zähler pro CAN Nachricht um >1 erhöht wird.

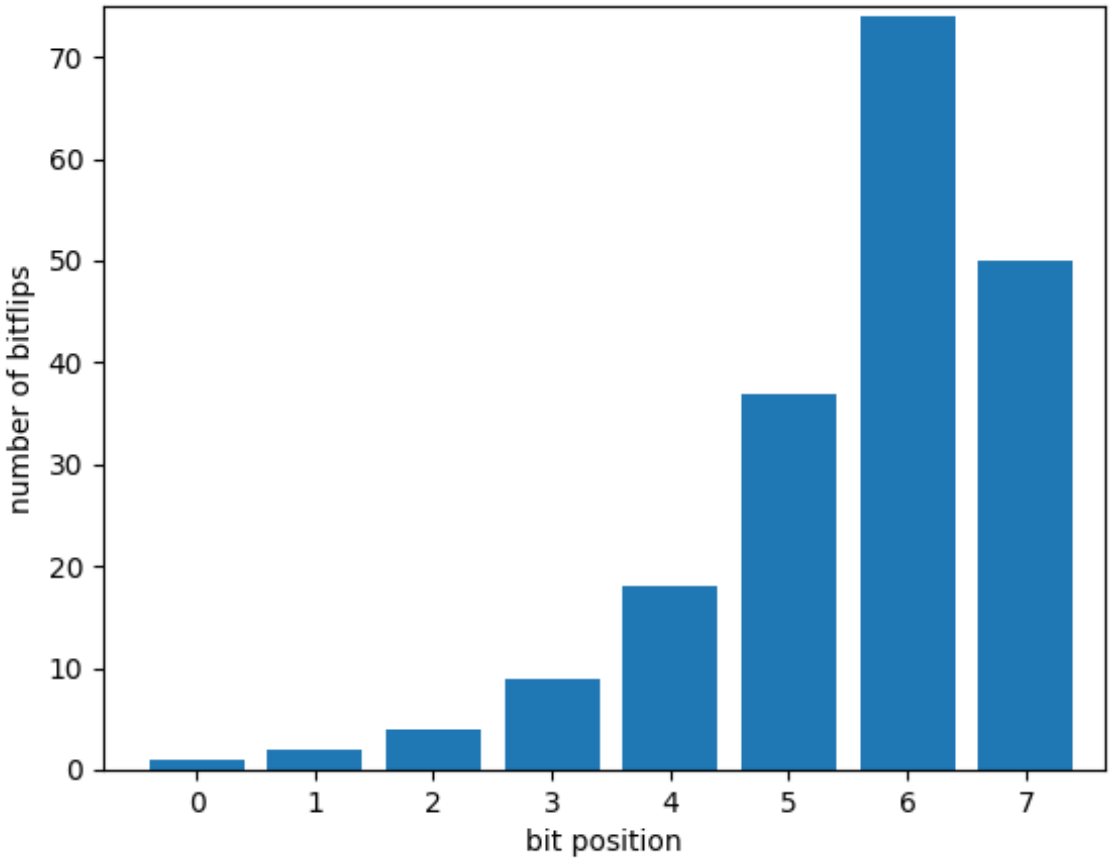

Abbildung 6: Nicht monoton ansteigende Bitflips

In Abbildung 6 ist zu sehen was passiert, wenn nicht bei jeder neuen CAN Nachricht der Zähler nur um 1 erhöht wird, sondern um 1,5. In anderen Worten wird der Zähler bei jeder zweiten Nachricht um 2 erhöht, anstatt nur um 1. Das hat zur Folge, dass sich das LSB nicht verändert und so auch die Bitflips an dieser Stelle nicht erhöht werden.

Ein weiterer Fall, den die Bitflips nur sehr schlecht erkennen können, ist das Vorkommen von Multiplexer Feldern, anstatt von Zählern. Bei diesen ist nämlich ein monotones Ansteigen der Bitflips bis zum LSB nicht gegeben, da die Werte keinem bestimmtem Muster folgen.

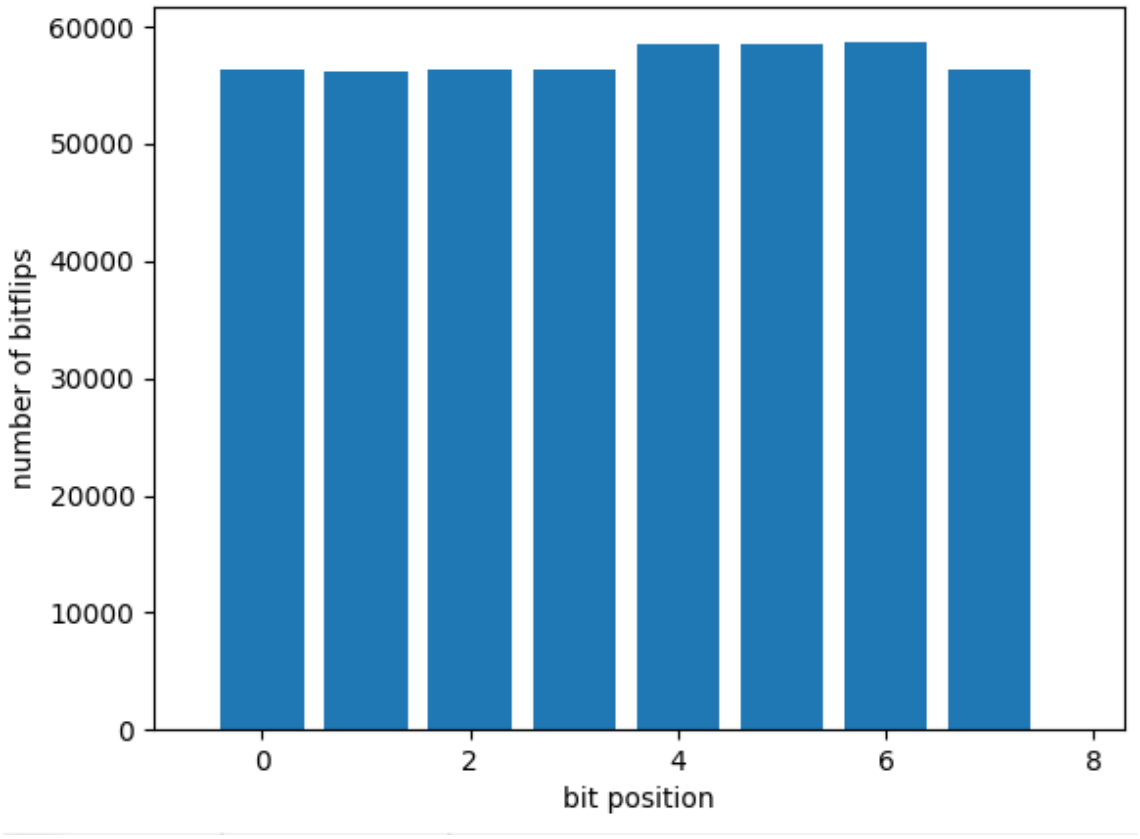

Abbildung 7: Bitflips Multiplexer

Die Abbildung 7 wurde erstellt, indem im oben aufgeführten candump Log (Abbildung 4) nach ID 161 gefiltert wird und die Bitflips der ersten 8 Bit gezählt werden. Es ist zu sehen, dass ein (aus der Sicht eines Angreifers) zufälliges Ändern eines Multiplexer Feldes bitweise eine relativ konstante Bitflips Metrik zur Folge hat.

3.2.2 Quotient

Die Quotienten Metrik ist mit den Bitflips eng verwandt, da diese benutzt werden, um die Quotienten Metrik zu errechnen. Bei dieser wird geprüft, in wie viel Prozent der Fälle, zwei aneinander anliegende Bits gleichzeitig kippen.

In Abbildung 8 ist die Quotienten Metrik beim Zählen von 0 bis 99 zu sehen. Dabei steht die jeweilige Position für den Quotienten zwischen dem Bit an dieser Stelle und dem nächsten Bit. Es fällt auf, dass alle Werte konstant auf „1“ liegen. Dies erschließt sich auch, da in 100% der Fälle, wenn ein Bit kippt, auch das Bit an der nächst höheren Stelle kippen muss. Bei Bit Position 0 ist die Metrik allerdings „0“,

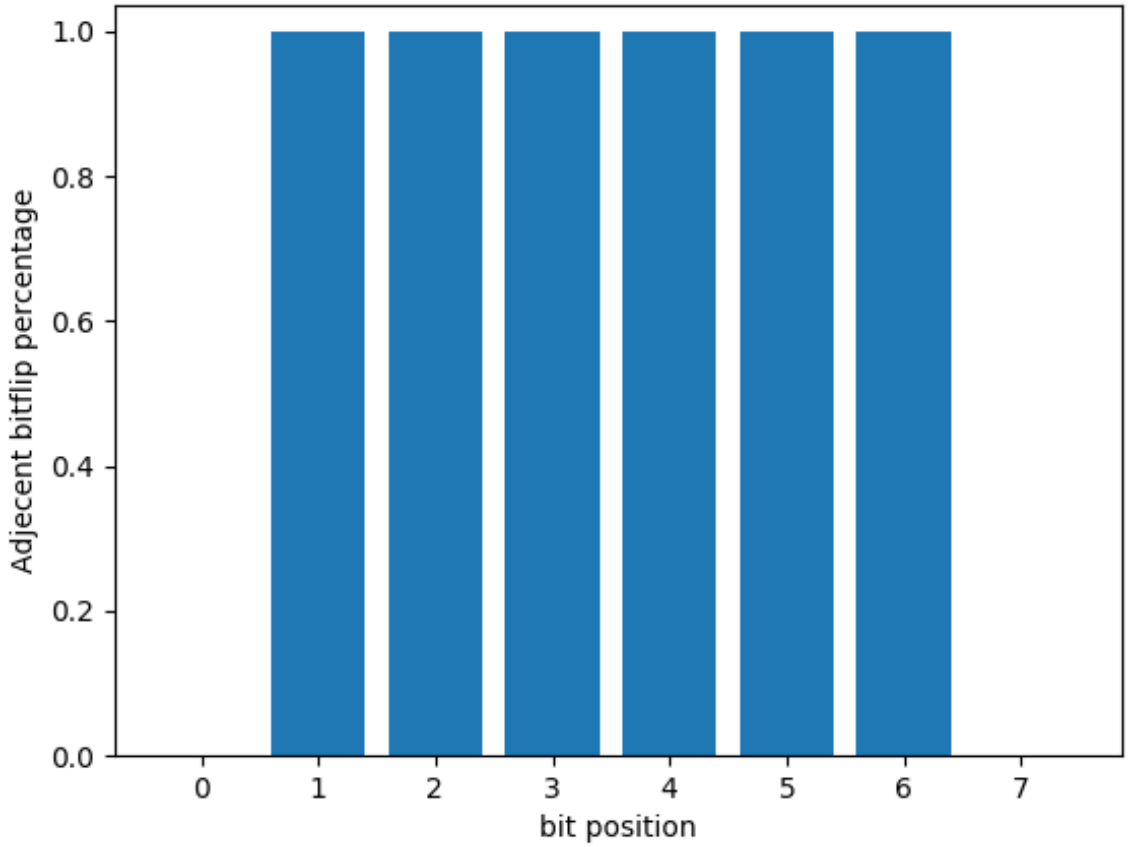

Abbildung 8: Quotienten Metrik beim Zählen von 0 bis 99

was aber daran liegt, dass an Bit 0 keine Bitflips vorkommen, da so weit nicht gezählt wird. Ebenso ist Bit Position 7 „0“, da Bit 8 nicht mehr Teil der Nachricht ist.

3.2.2.1 Fehlerquellen

Auch bei dem Quotienten kann es zu einer schwierigeren Interpretation der Daten führen, wenn ein Zählers mit einem Faktor größer als 1 erhöht wird.

Abbildung 9 wurde mit den selben Daten wie Abbildung 6 erstellt. Auffällig ist die Position 6 mit Wert „0,5“. Dieser kommt zustande, da das Bit an Position 7 nur in 50% der Fälle gleichzeitig mit Bit 6 kippt, da Bit 7 jeden zweiten Bitflip überspringt.

Des Weiteren kann es zu Fehlern bei dieser Metrik kommen, indem zwei Felder direkt nebeneinander liegen, wenn das vordere Feld ein Zähler ist, dessen LSB nah an die Anzahl maximal möglicher Bitflips kommt. Dies bedeutet, dass das gleichzeitige Kippen immer vorkommt, wenn gerade das Bit kippt, welches nicht immer kippt.

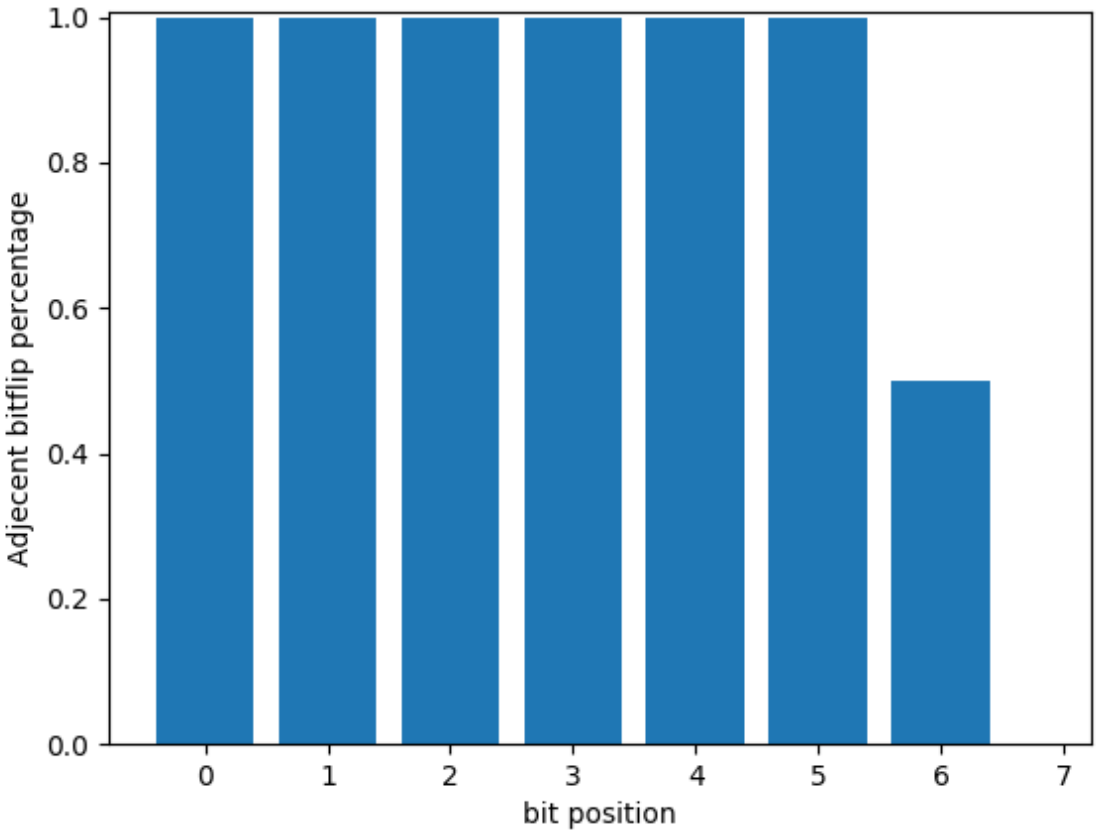

Abbildung 9: Quotienten Metrik bei Erhöhen von Zähler mit Faktor 1,5

In Abbildung 10 sind die Bitflips dieses Szenarios dargestellt. An der Stelle 7 betragen die Bitflips 99, was bei 100 Nachrichten das Maximum darstellt, und bei Bit 8 beginnt bereits ein neues Feld. In Abbildung 11 sind die Quotienten dargestellt. An Stelle 7 ist nun nicht zu sehen, dass hier ein neues Feld beginnt, sondern es hat den Anschein, das selbe Feld wird fortgeführt.

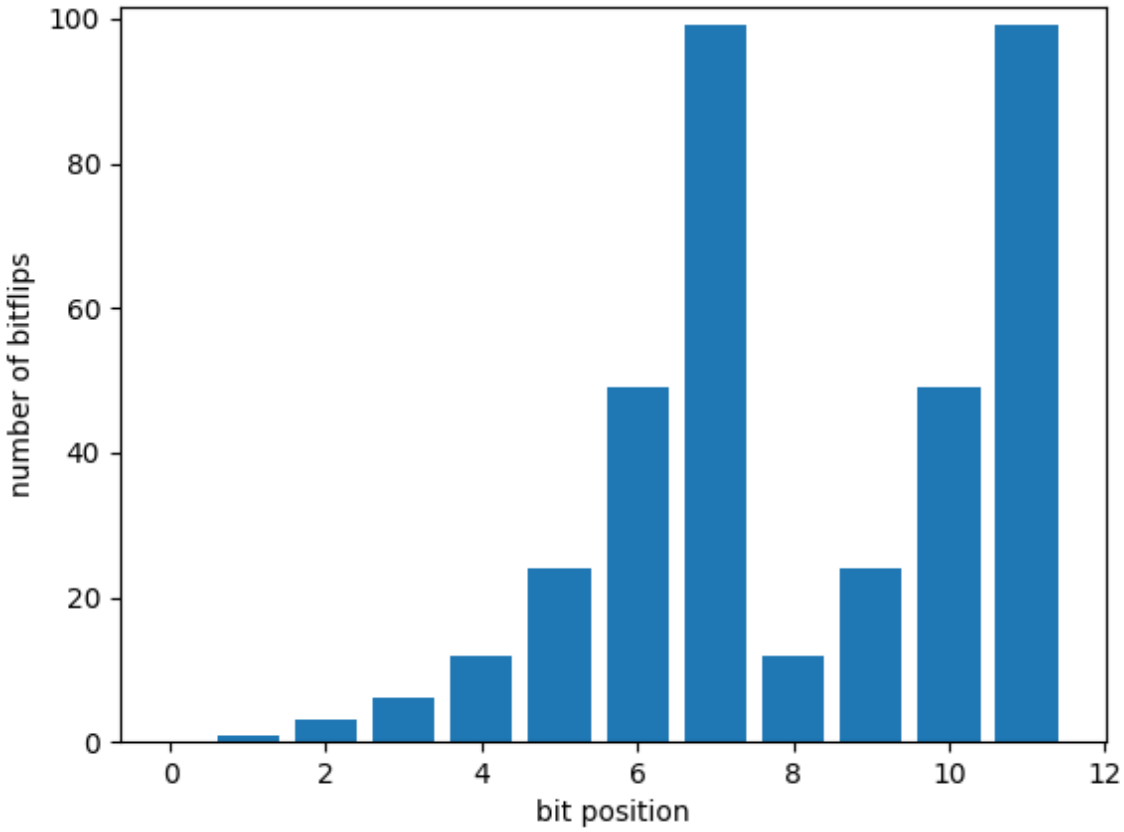

Abbildung 10: Bitflips bei zwei aneinander anliegenden Feldern

4 Implementierung

4.1 Python & NumPy

Alle erstellten Programme wurden in Python3 implementiert. Grund für diese Entscheidung war, dass Python generell einfach zu erlernen und gleichzeitig relativ schnell ist. Da es in Python keine Arrays, sondern nur Listen gibt, wurde mit der „NumPy“ Bibliothek gearbeitet, die die deutlich performanteren Arrays einführt. Dies ist eine enorme Erleichterung gegenüber den Python Listen, da es bei diesen vorgesehen ist, dass die Länge nach der Deklarierung veränderbar ist. Dies ist für die Implementierung dieser Metriken allerdings nicht erforderlich und es werden Arrays mit mehreren Millionen Einträgen benötigt, was sehr lange Laufzeiten zur Folge hätte. Somit wurde die schlechte Performance beim Verarbeiten von großen Datenmengen erheblich verbessert.

Ebenfalls in NumPy enthalten ist ein Zufallszahlen Generator, der für die Simulation in Kapitel 4.2 gebraucht wird.

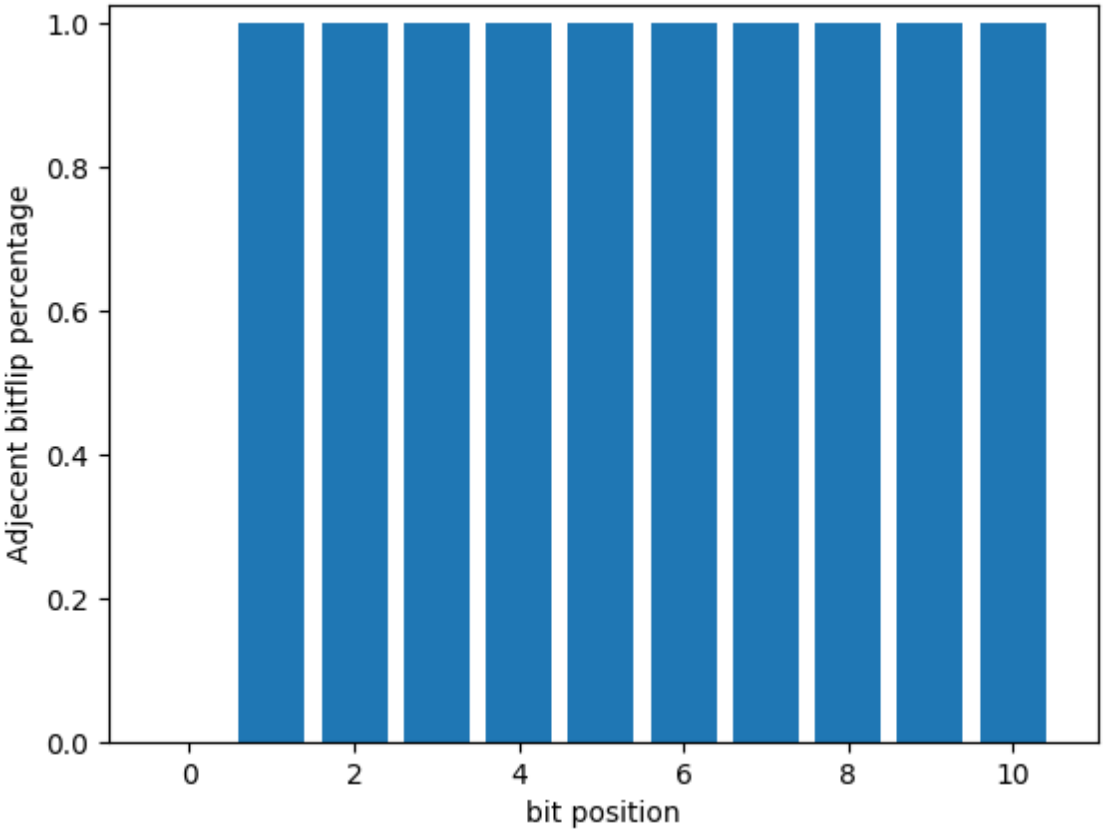

Abbildung 11: Quotienten bei zwei aneinander anliegenden Feldern

4.2 CAN-Trace Simulation

Eines der Probleme war die Verfügbarkeit von Candumps, deren wirkliche Daten bekannt sind. Solche Daten sind wichtig, um die Korrektheit des entwickelten Programms zu testen. Da aber nur Candumps von echten Fahrzeugen vorhanden sind, von welchen weder bekannt ist welche Steuergeräte genau verbaut sind, noch der Quellcode von diesen, können nicht zweifelsfreie Aussagen über die echten versendeten Daten getroffen werden.

Deshalb wurde entschieden ein weiteres Programm zu erstellen, welches es ermöglicht eine Candump Logdatei zu simulieren.

In diesem Programm sollte es möglich sein Datenpakete aus verschiedenen Daten zusammenzubauen und auch die Position dieser festzulegen. Diese Daten sollten dann noch mit steigender Nachrichtennummer verändert werden, damit eine Analyse stattfinden kann. Ebenfalls muss noch eine ID, das gewünschte CAN Interface und der Zeitstempel hinzugefügt werden.

In Abbildung 12 ist eine simulierte Logdatei aufgeführt. Die Verschiebung bei manchen Nachrichten folgt daraus, dass beim Zeitstempel kein Padding mit Nullen an den hinteren Stellen stattfindet.

```
(1579855478.3417382) can0 02f#0b000014000000ff
(1579855478.3418536) can0 02f#0a00001500000102
(1579855478.3419025) can0 02f#0200001700000106
(1579855478.341943) can0 02f#030000180000010a
(1579855478.3419611) can0 02f#0000001a0000010e
(1579855478.3419814) can0 02f#0200001b00000112
(1579855478.3420002) can0 02f#0a00001d00000116
(1579855478.3420155) can0 02f#0a00001e0000011a
(1579855478.3420308) can0 02f#020000200000011e
(1579855478.342046) can0 02f#0b00002100000122
(1579855478.3420606) can0 02f#0900002300000126
(1579855478.342075) can0 02f#0d0000240000012a
(1579855478.3420894) can0 02f#0a0000260000012e
(1579855478.3421037) can0 02f#0100002700000132
(1579855478.3421185) can0 02f#0c00002900000136
(1579855478.3421328) can0 02f#0500002a0000013a
(1579855478.342147) can0 02f#0e00002c0000013e
(1579855478.3421617) can0 02f#0400002d00000141
(1579855478.3421755) can0 02f#0800002f00000145
(1579855478.3421898) can0 02f#0000003000000149
(1579855478.342204) can0 02f#030000320000014d
(1579855478.3422186) can0 02f#0400003300000151
(1579855478.342233) can0 02f#0100003500000154
(1579855478.342247) can0 02f#0e00003600000158
(1579855478.3422606) can0 02f#0c0000380000015c
```

Abbildung 12: Simulierte candump.log Datei

Das ausgewählte CAN Interface ist „can0“ und die ID bei allen Nachrichten ist 47. Die Anzahl der Nachrichten ist 100000.

Der Inhalt der Nachrichten besteht erstens aus einem Multiplexer Feld, welches 4 Bit lang ist und an Position 4 beginnt. Um ein Multiplexer Feld zu simulieren wurde eine zufällige Zahl von 0-15 generiert. Der Zufallszahlengenerator, der hierfür verwendet wurde, ist „numpy.random.default_rng()“. Mit diesem Aufruf wird ein neuer Generator erstellt, mit welchem man dann die Funktion „integers()“ nutzen kann.

Anschließend folgt ein 3 Byte langer Zähler, der von 20 aus nach oben zählt und bei jeder zweiten Nachricht um 2 erhöht wird, wie in Abbildung 6 und 9. Implementiert ist dies durch die Addition von 20 und dem Produkt aus der Laufvariable „i“ und 1,5. Anschließend muss das Ergebnis noch in eine Ganzzahl umgewandelt werden.

Die letzten 4 Byte sind ebenfalls ein Zähler, allerdings zählt dieser nicht konstant nach oben, sondern er bewegt sich immer zwischen 0 und 510. Erreicht wird dies, indem mithilfe der „math“ Bibliothek der Sinus an der Stelle „i/64“ errechnet wird. Dies ergibt natürlich ein Ergebnis von -1 bis 1. Dieses Ergebnis wird noch mit 255 multipliziert und auf 255 addiert.

Das Teilen der Laufvariable durch 64 ist von Vorteil, da beim Sinus langsamer die Extremwerte -1 und 1 erreicht werden.

4.3 CAN-Trace Analyse

Der Großteil der Implementierung stellt die Analyse der CAN Daten selbst dar. Alle wichtigen Funktionen sind Teil der Klasse „DataContainer“.

```
if __name__ = '__main__':
        main()
```

Dieser Code bewirkt, dass beim Aufrufen des Skriptes nur die Funktion „main()“ ausgeführt wird. Vorrangig dient dies der Lesbarkeit des Codes, da der Programmfluss klarer erkennbar wird.

Die „main()“ Funktion erstellt zuerst einen Parser mithilfe des Paketes „argparse“. Mit diesem können Argumente beim Aufruf des Skriptes übergeben werden. Die benötigten Argumente sind:

- input
- id
- type

Das „input“ Argument erwartet eine Datei im „candump.log“ Format. Benötigt wird dies, da verschiedene Logdateien vorhanden sein können und immer die gewünschte ausgewählt werden soll. Dieses Argument muss immer gegeben sein.

Das ID Argument wird genutzt, um in eben dieser Logdatei alle Nachrichten mit der gewünschten ID zu analysieren. Diese ID muss als Dezimalzahl angegeben werden. Falls dieses Argument fehlt, wird eine Anzeige mit allen vorhandenen IDs per Konsole ausgegeben.

Mit „type“ wird spezifiziert, welche Metriken zum Finden der Datenfelder benutzt wird. Um die Daten nur mit den Bitflips zu analysieren, muss „bf“ übergeben werden. Will man stattdessen die Ergebnisse des Quotienten erhalten, muss man das Programm mit „q“ aufrufen. Es ist ebenfalls möglich die Daten mit beiden Metriken gleichzeitig zu analysieren. Dazu muss „b“ übergeben werden. Auch dieses Argument muss immer vorhanden sein.

Ein Beispiel eines korrekten Aufrufs des Programms via Bash sieht also so aus:

```
python3 ba_classify.py -i candump-2019-04-05_104847.log --id 47 -t q
```

In Abbildung 13 ist ein Aktivitätsdiagramm dargestellt, das den Ablauf des gesamten weiteren Programms nochmals darstellt. Dabei stehen die einzelnen Aktivitäten für Funktionen. Diese werden im Anschluss einzeln erläutert.

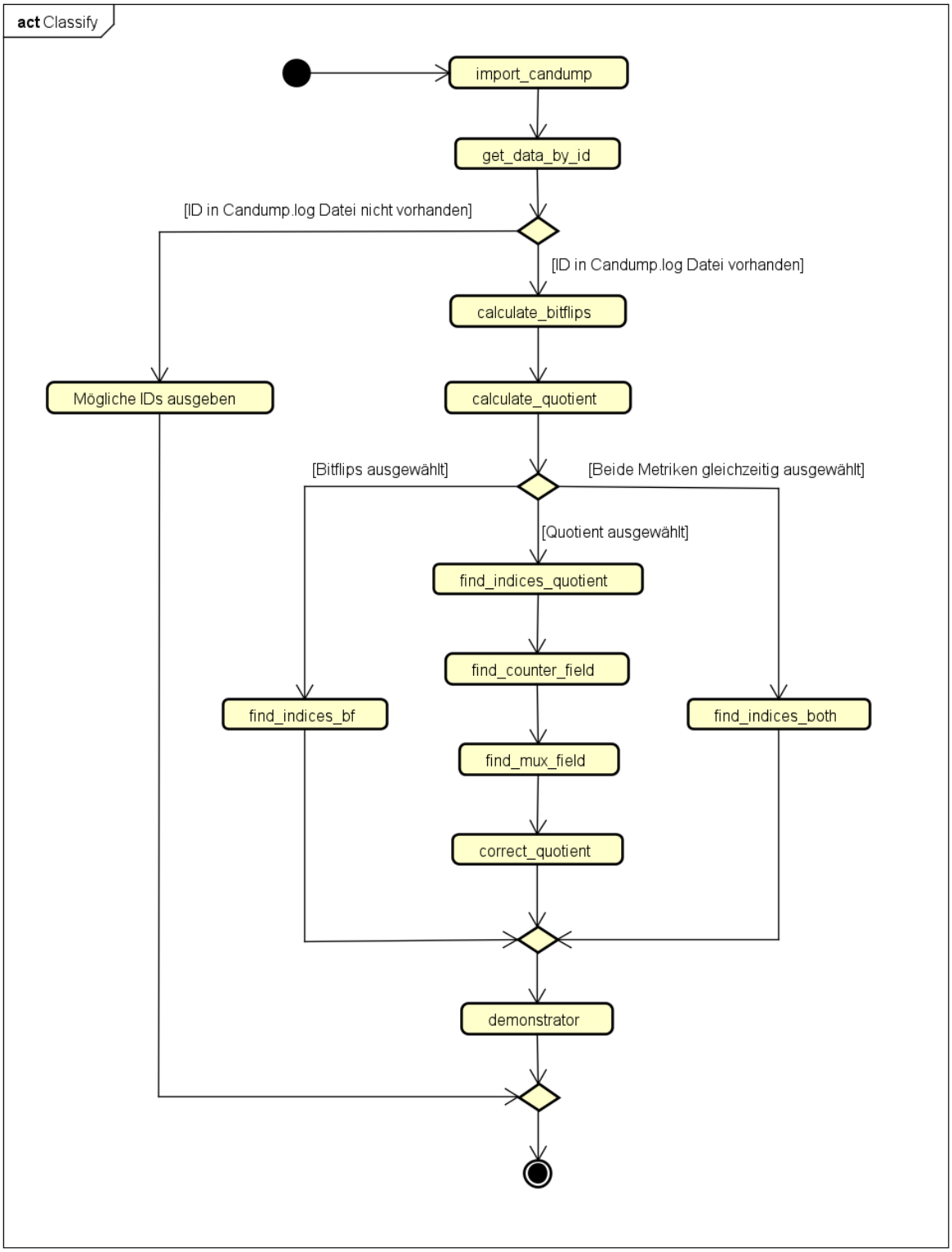

Abbildung 13: Aktivitätsdiagramm, das den Ablauf des „ba_classify.py“ Programms beschreibt

4.3.1 import_candump

Die erste Funktion, die aufgerufen wird ist „import_candump“. Diese nimmt das „input“ Argument und öffnet die gegebene Logdatei. In dieser wird nun jede Zeile aufgeteilt in:

- timestamp
- identifier
- data

Diese Informationen werden anschließend separat in Numpy Arrays gespeichert.

Diese Funktion war bereits am Anfang der Bachelorarbeit gegeben und wurde nicht im Zuge dieser erstellt.

4.3.2 get_data_by_id

Nachdem das Importieren der Daten abgeschlossen ist, kann mit dem sortieren der Daten fortgefahren werden.

Hierzu wird zuerst geprüft, ob die gewünschte ID überhaupt in der Logdatei vorhanden ist. Ist dies nicht der Fall, wird per Konsole eine Fehlermeldung und die Liste der möglichen IDs ausgegeben. Anschließend wird die Ausführung des Programms beendet. Ist die ID jedoch vorhanden, kann das Programm weiter ausführen.

Nun werden alle Nachrichten nacheinander überprüft, ob deren ID gleich der übergebenen ID ist. Wenn dies so ist, wird das Datenfeld dieser Nachricht in die Binärform umgewandelt.

Es ist wichtig, dass bei dieser Umwandlung die führenden Nullen, die häufig vorkommen, nicht abgeschnitten werden. Der Code dazu wird hier kurz erklärt.

```
b_data = np.binary_repr(line["data"])
b_str = "0" * 64
b_data = b_str + b_data
b_data = b_data[len(b_data)-64:]
```

In Zeile 1 des Codes geschieht die Umwandlung nach Binär mit der Numpy Funktion „binary_repr“. Der Rückgabetyp dieser Funktion ist ein String. Zeile 2 erstellt einen String mit 64 0en. Diese beiden Strings werden nun konkateniert. In Zeile 4 betrachtet man nun nur die hinteren 64 Ziffern des konkatenierten Strings.

Der entstandene String wird anschließend in das „bin_data“ Array bitweise kopiert. Dieses Array hat zwei Dimensionen. Die erste Dimension steht für je eine neue Nachricht und die zweite steht für je ein Bit des Datenfeldes.

4.3.3 calculate_bitflips

Sobald alle Datenfelder der gewünschten ID binär abgespeichert sind kann begonnen werden die Bitflips Metrik zu erstellen. Da das Abspeichern der Daten in Arrays geschieht, ist dieser Schritt sehr einfach.

Dafür wird in dem „bin_data“ Array durch alle Nachrichten iteriert und der Wert jedes Bits wird verglichen mit dem Wert, den es in der vorhergehenden Nachricht hatte. Falls sich diese unterscheiden wird in einem neuen Array, genannt „bf“, ein Zähler an genau dieser Bit Position erhöht. Dieses Array hat eine Länge von 64, damit für jede Bit Position genau ein Zähler vorhanden ist.

	Bit #0	Bit #1	Bit #2	Bit #3
Nachricht #1	0	0	1	1
Nachricht #2	0	1	0	0
Nachricht #3	0	1	0	1
Nachricht #4	0	1	1	0

Tabelle 1: bin_data

	bf[0]	bf[1]	bf[2]	bf[3]
nach Nachricht #1	0	0	0	0
nach Nachricht #2	0	1	1	1
nach Nachricht #3	0	1	1	2
nach Nachricht #4	0	1	2	3

Tabelle 2: bf

In Tabelle 1 ist ein Beispiel für das „bin_data“ Array verkürzt und abstrahiert abgebildet. Nach jeder Nachricht wird bei jedem Bit geprüft, ob ein Bitflip vorkam. Ist dies der Fall wird an dieser Stelle im „bf“ Array der Zähler erhöht.

4.3.4 calculate_quotient

Das erstellen der Quotienten Metrik gestaltete sich ein wenig problematischer.

Dafür muss zuerst wieder das „bin_data“ Array betrachtet werden. Hier wird bei jeder Nachricht gespeichert, ob bei einem Bit ein Bitflip vorgekommen ist. Nun wird gezählt, wie oft ein Bit in einer Nachricht und auch das nächste Bit gleichzeitig einen Bitflip haben. Diese Information wird im „adj_bf“ Array gespeichert.

Mit diesem Zähler kann nun die Quotienten Metrik erstellt werden. Dafür wird verglichen, welche der beiden Stellen eine kleinere Bitflip Anzahl hat. Dies wird einfach aus „bf“ übernommen. Zuletzt wird noch jede Stelle von „adj_bf“ durch das korrespondierende Minimum geteilt, sodass Maximal ein Wert von 1 entstehen kann.

Die folgende Formel verdeutlicht diese Implementierung nochmal.

$$quotient[x] = \frac{adj\_bf[x]}{Min(bf[x],bf[x+1])}$$

	adj_bf[0]	adj_bf[1]	adj_bf[2]	adj_bf[3]
nach Nachricht #1	0	0	0	0
nach Nachricht #2	0	1	1	0
nach Nachricht #3	0	1	1	0
nach Nachricht #4	0	1	2	0

Tabelle 3: adj_bf

Tabelle 4 bezieht sich auf die Nachrichten aus Tabelle 1. Wenn man die Werte aus „adj_bf“ und „bf“ in die Formel einsetzt, ergeben sich die folgenden Werte.

	quotient[0]	quotient[1]	quotient[2]	quotient[3]
nach Nachricht #1	0	0	0	0
nach Nachricht #2	0	1	1	0
nach Nachricht #3	0	1	1	0
nach Nachricht #4	0	1	1	0

Tabelle 4: quotient

Hiermit sind die Berechnung der benötigten Metriken abgeschlossen und es kann begonnen werden mit diesen Felder zu identifizieren.

4.3.5 find_indices_bf

Mithilfe der „find_indices“ Funktionen werden Anfang und Ende eines Feldes ermittelt. Die verschiedenen Metriken können hierbei verschiedene Parameter einsetzen, um ein gewünschtes Ergebnis zu erzielen. Ebenfalls macht es natürlich einen Unterschied, ob man nur eine Metrik benutzt, oder beide.

In der „find_indices_bf“ Funktion geschieht das Finden der Indices ausschließlich mit den Bitflips. Deshalb muss nur durch das „bf“ Array iteriert werden.

Dabei wird ein Startpunkt gesetzt und von diesem ausgehend überprüft, an welcher Stelle die Bitflips nicht mehr monoton steigen. Zudem muss noch festgelegt werden, dass nur Werte berücksichtigt werden, die über 0 liegen. Anschließend werden Start- und Endpunkte als Element einer Liste abgespeichert. Diese hat den Namen „indices_bf“.

In Abbildung 14 sind die Bitflips der Nachrichten mit ID 47 aus „candump-2019-04-05_104847.log“ zu sehen.

Wenn auf diesen Datensatz die „find_indices_bf“ Funktion angewendet wird, ergeben sich die folgenden Feldgrenzen:

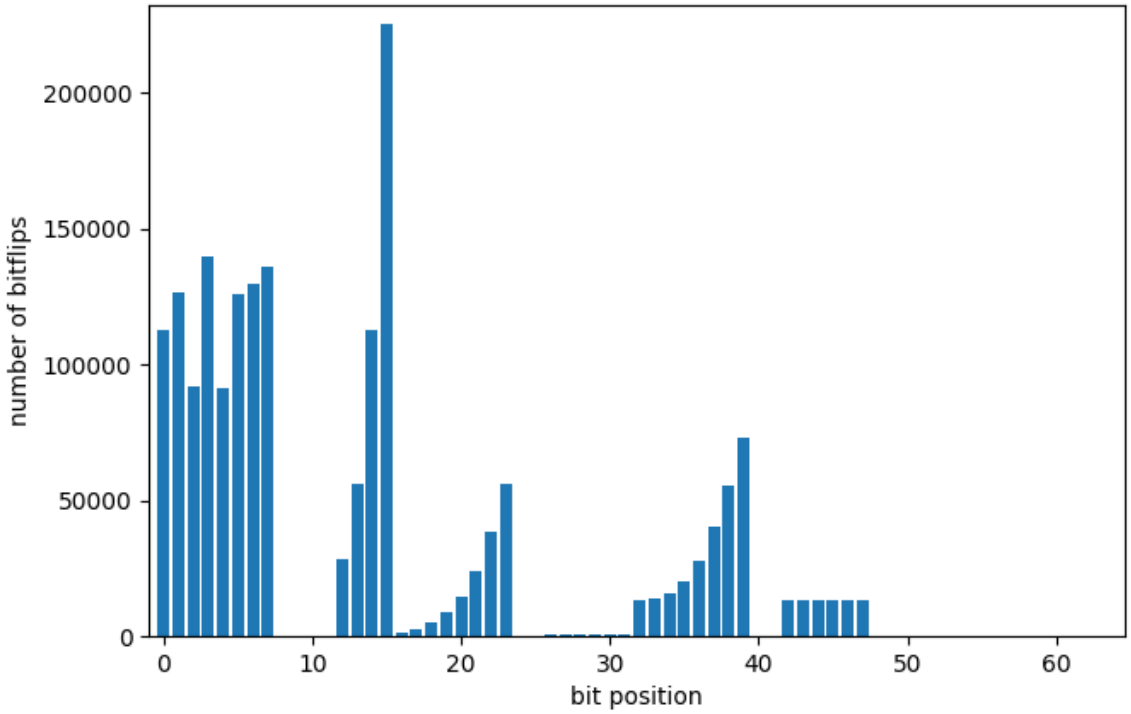

Abbildung 14: Bitflips von ID 47 aus „candump-2019-04-05_104847.log“

- 0-1
- 2-3
- 4-7
- 12-15
- 16-23
- 25-39
- 42-47

Da Bitflips keine Felder erkennen können, die nicht monoton steigen, wird das erste Feld von 0-7, welches vermutlich ein Multiplexer ist, sehr schlecht erkannt. Es werden nur Einzelteile, bei denen ein Anstieg an Bitflips zwischen zwei Stellen vorhanden ist, als Feld erkannt.

Das nächste gefundene Feld liegt bei 12-15, was auch korrekt ist.

Auch das Feld bei 16-23 wird korrekt erkannt.

25-39 ist kein korrekt erkanntes Feld, da 26-31 ein Feld und 31-39 ein anderes Feld ist.

Zuletzt wird richtig erkannt, dass 42-47 ein Feld ist.

Da die Bitflips der Bit Positionen von 26 bis 31 im Vergleich sehr klein sind, werden diese nochmal in Abbildung 15 vergrößert dargestellt.

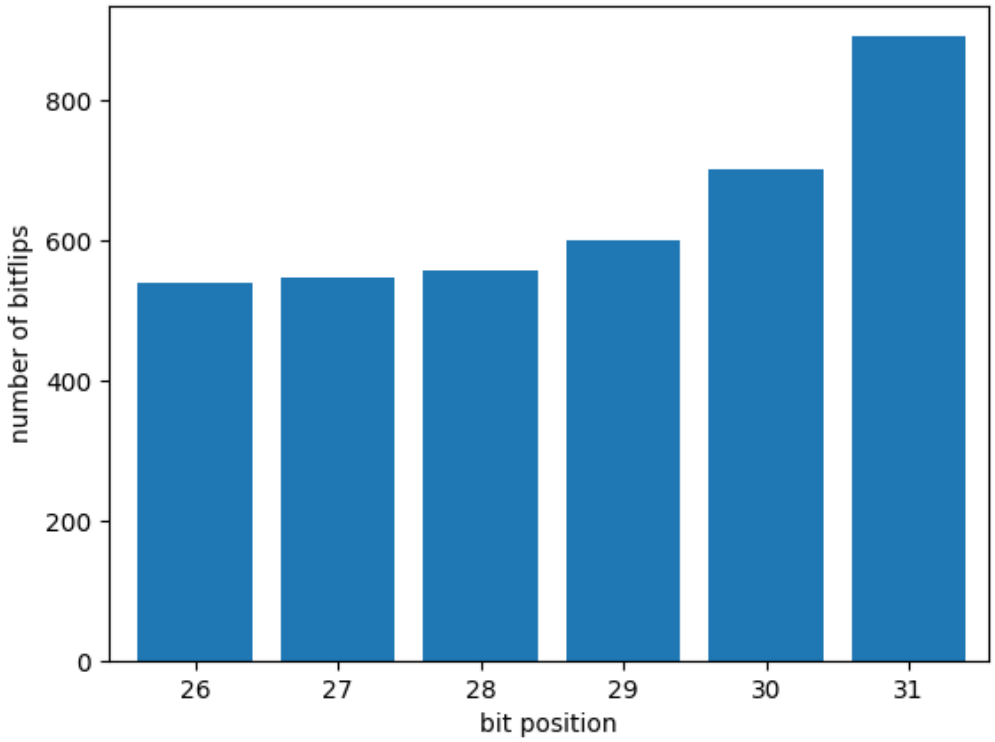

Abbildung 15: Nahaufnahme der Bitflips von Position 26-31 aus Abbildung 14

4.3.6 find_indices_quotient

Die nächste „find_indices“ Funktion versucht die Feldgrenzen mit dem Quotienten zu finden.

Dazu wird geprüft, wie lange die Quotienten Metrik über 0,3 liegt. Die Ergebnisse werden wieder in eine List abgespeichert, die hier „indices_q“ genannt ist.

Dieser Parameter wurde so gewählt, da er niedrig genug ist, damit Felder, wie z.b bei Abbildung 9, trotzdem als zusammengehörend interpretiert werden. Gleichzeitig ist es unwahrscheinlich, dass zwei aneinander anliegende Felder einen Quotienten haben, der über 0,3 liegt.

Wie schon bei Kapitel 4.3.5, sind auch in Abbildung 16 die Nachrichten der ID 47 dargestellt, jedoch hier mit dem Quotienten.

Die gefundenen Felder sind:

- 0-7
- 12-23
- 26-31

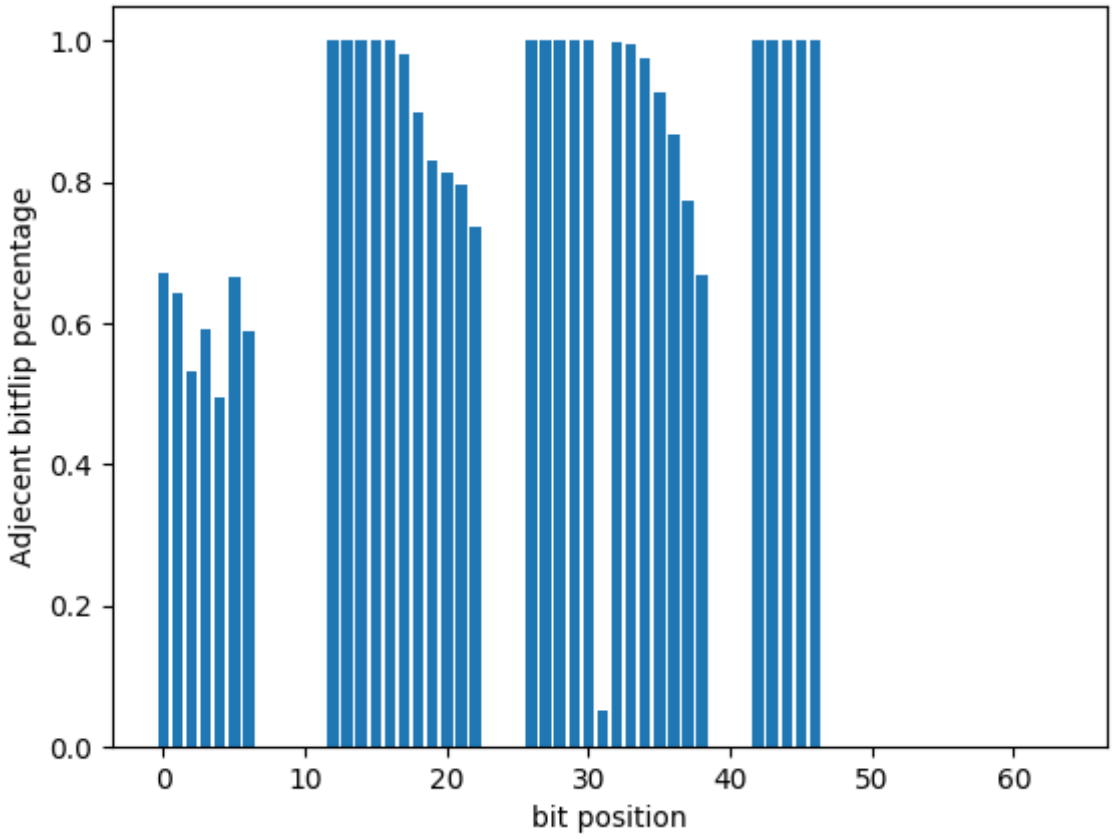

Abbildung 16: Quotienten von ID 47 aus „candump-2019-04-05_104847.log“

- 32-39
- 42-47

Das erste Feld, das von 0-7 geht und zuvor nur teilweise erkannt wurde, kann mit dieser Metrik erfolgreich identifiziert werden.

Als nächstes wird ein Feld bei 12-23 erkannt. Hierbei kommt es zu einem Fehler, der bereits in Kapitel 3.2.2.1 beschrieben wurde. Es liegen zwei Felder nebeneinander, die nicht voneinander unterschieden werden können.

Das Feld in 26-31, welches zuvor nicht korrekt erkannt wurde und in Abbildung 15 dargestellt ist, kann mit dieser Metrik korrekt bestimmt werden.

Ebenso wird das vorletzte Feld bei 32-39, welches mit „find_indices_bf“ nicht erkannt wurden, hier richtig identifiziert.

Auch das letzte Feld von 42-47 wird hier richtig erkannt

Mit dieser Implementierung ist es also bereits möglich fast alle zusammenhängenden Felder der Nachricht, mit Ausnahme von 12-15 und 16-23, fehlerfrei zu erkennen.

4.3.7 find_indices_both

Im nächsten Schritt werden nun beide Metriken gleichzeitig genutzt, um die korrekten Feldgrenzen zu finden.

Um dies zu erreichen wird gleichzeitig geprüft, ob an zwei folgenden Stellen der Quotient über 0,3 liegt und ob die Bitflips an der hinteren Stelle höher sind, als an der Vorderen. Die Liste, in der die Daten abgespeichert werden heißt „indices_both“

Auch hier wird wieder die selbe Nachricht wie zuvor in Abbildung 14 und 16 untersucht. Dabei werden die folgenden Felder gefunden:

- 0-1
- 2-3
- 4-7
- 12-15
- 16-23
- 26-31
- 32-39
- 42-47

Mit dieser Implementierung werden nur Einzelteile des ersten Feldes von 0-7 erkannt. Diese werden voneinander abgeschnitten, da die Bitflips an Stelle 2 und 4 kleiner sind, als an Stelle 1 und 3.

Bei den nächsten beiden Feldern, 12-15 und 16-23, wird korrekt erkannt, dass es sich um zwei verschiedene Felder handelt.

Ebenso richtig erkannt werden die restlichen Felder von 32-39 und von 42-47.

Mit dieser Funktion werden ebenso, wie schon mit „find_indices_quotient“, fast alle Felder richtig erkannt.

4.3.8 find_counter_fields

In dieser und der folgenden Funktion wird versucht aus den Feldern, die mit „find_indices_quotient“ gefunden wurden, weiterzuarbeiten und die Ergebnisse zu verbessern.

Vorrangig soll der Fehler, der bereits in Kapitel 3.2.2.1 Absatz 3 beschrieben wurde, beseitigt werden. Um dies zu erreichen wurde versucht die von „find_indices_quotient“ gefundenen Felder zu unterteilen in Zähler und Multiplexer.

Dabei übernimmt diese Funktion das Finden von Zählern.

Dazu werden die Feldgrenzen von „indices_q“ zunächst übernommen. Anschließend wird vom ersten Element aus gestartet und überprüft, wie lange die Bitflips monoton ansteigen. Sobald dies nicht mehr der Fall ist wird der Index dieser Stelle gespeichert. Ist dieses neue Feld nun mindestens 4 Stellen lang, wird das neue Feld in der Liste „counter_list“ abgespeichert. Daraufhin wird getestet, ob das von „indices_q“ vorgegebene Feld zu ende ist, oder ob noch mehr Stellen vorhanden sind. In diesem Fall wird überprüft, ob weitere Zahler in dem ursprünglichen Feld vorhanden sind.

Dies wird für alle Elemente aus der „indices_q“ Liste wiederholt, bis diese vollständig getestet wurde.

Mit dieser Veränderung werden die folgenden Felder gefunden:

- 4-7
- 12-15
- 16-23
- 26-31
- 32-39
- 42-47

Wie auch bei Abschnitt 4.3.5 und 4.3.7 wird das erste Feld nicht korrekt erkannt, da die Anzahl der Bitflips pro Stelle nicht monoton steigt.

Die nächsten beiden aneinander anliegenden Felder werden wieder richtig erkannt.

Ebenso werden die restlichen Felder wieder korrekt interpretiert.

4.3.9 find_mux_fields

Die „find_mux_fields“ Funktion versucht analog zu „find_counter_fields“ mithilfe der Einträge aus „indices_q“ Multiplexer Felder zu finden.

Der Ausgangspunkt ist, wie schon zuvor, die „indices_q“ Liste. Jedoch wird hier vom ersten Element aus getestet, bis zu welcher Stelle die Anzahl der Bitflips das 0,9 bis 1,1 - fache des Vorgängers betragen. Auch hier muss das neue Feld mindestens 4 Stellen lang sein, um berücksichtigt zu werden. Ist dies der Fall wird das restliche Feld ebenfalls getestet, ob ein weiterer Multiplexer folgt.

Dasselbe wird mit jedem Element aus der „indices_q“ Liste wiederholt.

Aus dieser Funktion entstehen dann die folgenden Felder:

- 26-29
- 42-47

Man sieht, dass hier nur wenig Felder erkannt werden. Zudem ist das Feld von 26-29 falsch erkannt, da sich dieses eigentlich von 26-31 erstreckt.

Das Feld von 42-47 hingegen ist korrekt erkannt.

Diese Funktion bietet keinen großen Fortschritt an, da sie die Art Multiplexer erkennt, die über alle Stellen eine relativ gleich große Anzahl an Bitflips hat. Diese Art Multiplexer wird allerdings bereits ohne große Probleme erkannt. Andere Multiplexer, die von Stelle zu Stelle sehr unterschiedliche Bitflips haben, werden auch mit dieser Funktion nicht erkannt.

Wegen diesen Tatsachen wird diese Funktion in der neuesten Version des Programms nicht mehr berücksichtigt.

4.3.10 correct_quotient

Mit dieser Funktion werden die gefundenen Felder aus Kapitel 4.3.6 und 4.3.8 vereint.

Dies geschieht, indem zuerst in beiden Arrays „indices_q“ und „counter_list“ gesucht wird, ob sie irgendwo eine gleiche Anfangsstelle eines Feldes haben. Anschließend wird geprüft, ob in „counter_list“ ein anderes Ende an dieser Stelle steht, als in „indices_q“. Dies bedeutet im Rückschluss, dass in „counter_list“ eine Unterteilung in mehrere Felder stattgefunden hat. Ist dies der Fall, wird der Wert aus „indices_q“ gelöscht und mit den unterteilten Feldern aus „counter_list“ ersetzt.

Mit dieser Verbesserung werden die folgenden Felder gefunden:

- 0-7
- 12-15
- 16-23
- 26-31
- 32-39
- 42-47

beim direkten Vergleich mit den Feldern aus Kapitel 4.3.6 fällt auf, dass alle Felder gleich geblieben sind, bis auf 12-23. Dieses wurde in 12-15 und 16-23 aufgeteilt.

Es scheint, als könne man mit dieser Implementierung diese Nachricht vollständig automatisch erkennen.

Diese Funktion ändert nur etwas an den Daten, wenn Anfang und Ende eines Feldes aus „indices_q“ in verschiedenen Feldern von „counter_list“ enthalten sind.

Dies hat den Nachteil zur Folge, dass nichts verändert wird, wenn Felder vorkommen, wie in Abbildung 6, da hier der Schlussindex falsch gesetzt werden wird. Positiv ist jedoch, dass dafür Felder, wie 0-7 aus 14, nicht verändert werden.

4.3.11 demonstrator

Zuletzt wurde noch eine Funktion erstellt, die alle gefundenen Felder auf dem Bildschirm aufzeichnet. Dazu wird das Paket „matplotlib“ verwendet. Wird diese Funktion aufgerufen, werden die Bitflips und der Quotient jedes Feldes in einem sogenannten „subplot“ ausgegeben.

Des Weiteren sind Funktionen vorhanden, die die gesamte Nachricht ausgeben. Diese Plots kann man miteinander vergleichen, um die Korrektheit der ausgewählten Felder zu testen.

5 Ergebnisse

Abschließend werden noch die Ergebnisse anderer IDs aus der Datei „candump-2019-04-05_104847.log“ mit der aktuellen Implementierung des Quotienten gezeigt.

5.1 ID 61

Wie der Names dieses Kapitels bereit voraus nimmt, wird hier die Nachricht mit ID 61 auf die vorhandenen Felder überprüft.

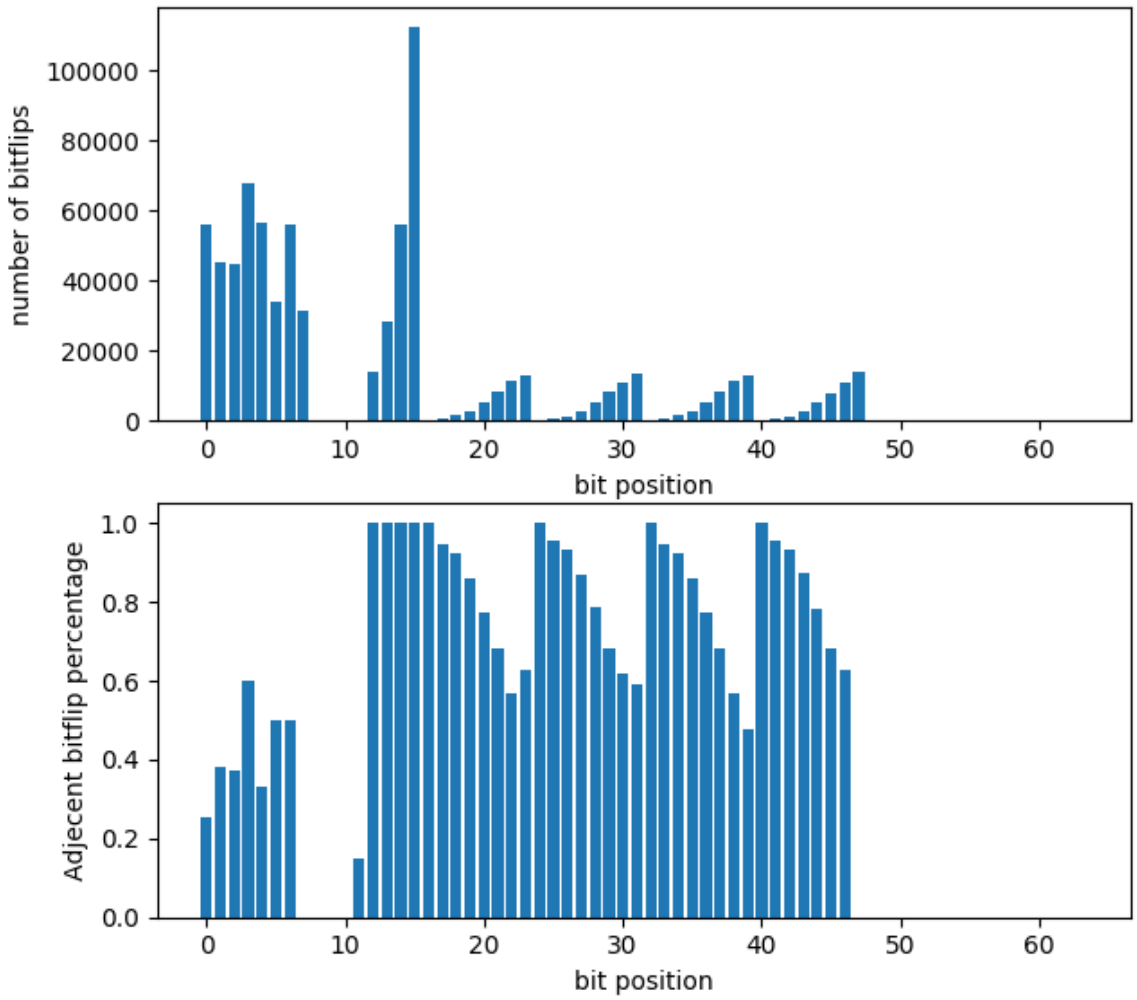

Abbildung 17: Metriken von ID 61 aus „candump-2019-04-05_104847.log“

In Abbildung 17 sind die Werte der beiden Metriken aufgezeichnet. Man vermutet Felder bei:

- 0-7
- 12-15
- 16-23
- 24-31
- 32-39
- 40-47

In der Grafik der Bitflips ist es kaum zu erkennen, doch die Startpositionen der Felder bei 16, 24, 32 und 40 sind nicht Null, sondern auf dieser Skala nur sehr klein dargestellt.

Mit der aktuellen Implementierung des Quotienten werden die folgenden Felder gefunden:

- 1-7
- 12-15
- 16-23
- 24-31
- 32-39
- 40-47

Das einzige Feld, welches nicht komplett richtig erkannt wird, ist das Feld 0-7, da bei diesem der Quotient an der Stelle 0 bei 0,25 liegt, sodass das erste Bit nicht in das Feld mit aufgenommen wird.

5.2 ID 150

In Abbildung 18 sind Bitflips und Quotient von ID 150 dargestellt. Die Felder dieser ID vermutet man bei:

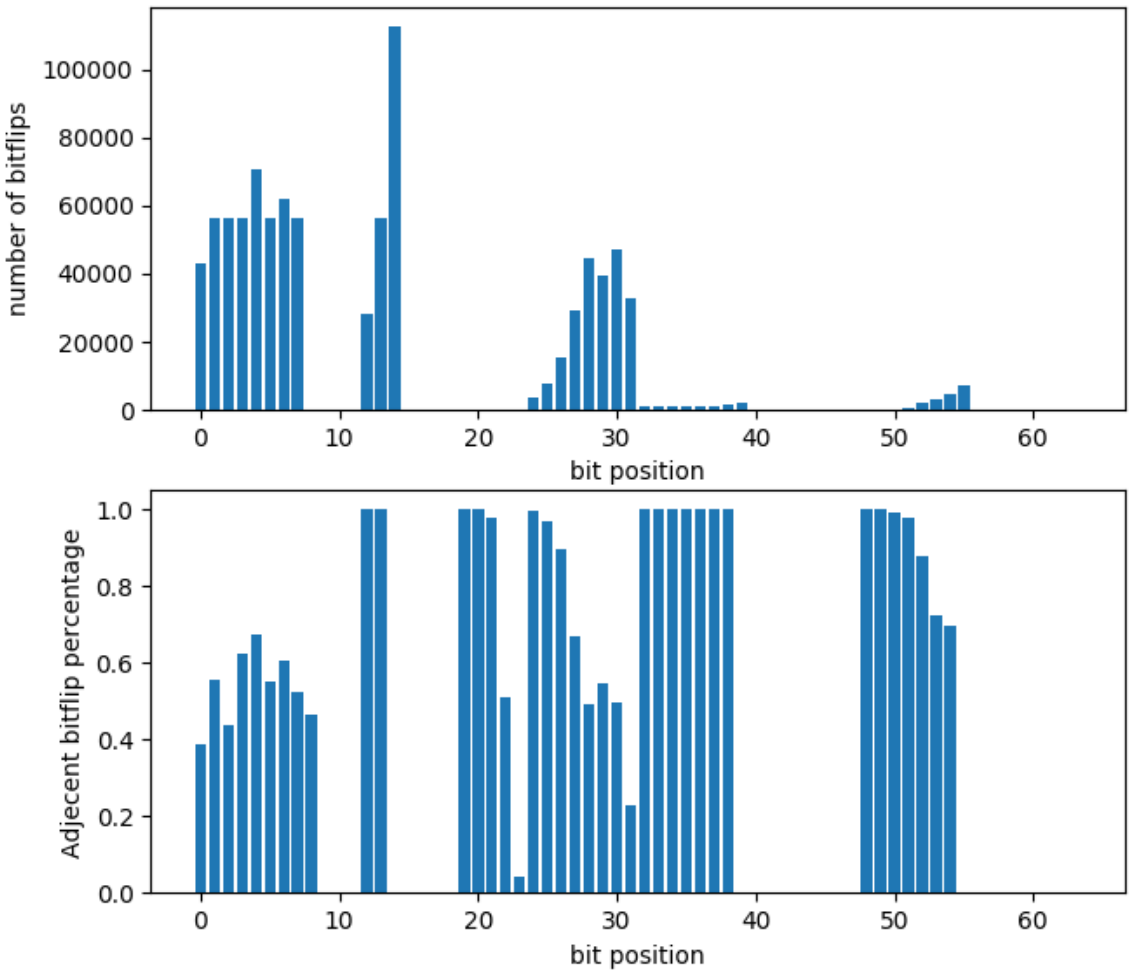

Abbildung 18: Metriken von ID 150 aus „candump-2019-04-05_104847.log“

- 0-7
- 8-9
- 12-14
- 19-23
- 24-31
- 32-39
- 48-55

Wie bereits im letzten Kapitel gibt es hier bei den Bitflips einige Stellen, die im Vergleich sehr klein und deshalb schlecht sichtbar sind. Diese sind 8-9, 19-23 und 48-50.

Mit dem Quotienten werden folgende Felder gefunden:

- 0-9
- 12-14
- 19-23
- 24-31
- 32-39
- 48-55

Auch hier werden wieder alle Felder richtig erkannt, außer das Erste. Diesmal kommt es hier zu einem Fehler, da der Quotient an Stelle 7 über 0,3 liegt und es wird zusätzlich von „find_counter_fields“ nicht erkannt, dass 8-9 ein eigenes Feld ist, da es nur 2 Bits lang ist.

6 Abschluss

6.1 Weitere Verbesserungsmöglichkeiten

Obwohl das Erkennen der Felder bereits gut funktioniert, gibt es trotzdem noch einige Stellen, an denen Verbesserungen vorgenommen werden können.

Eine dieser Stellen ist das Erkennen von speziellen Feldern. Im CAN Datenfeld kann es zum Beispiel den Fall geben, dass ein Datenpaket nicht lückenlos abgespeichert ist. Das bedeutet konkret, dass zum Beispiel ein Datenpaket von Bit 8 bis 15 und von 21 bis 23 reicht, während zwischen diesen ein anderes Paket versendet wird. Um dies zu Erkennen müsste allerdings zuerst die Quotienten Metrik erweitert werden, damit nicht nur der Quotient von benachbarten, sondern auch von beliebigen Stellen erstellt werden kann.

Des Weiteren könnten die Ergebnisse des Programms nicht nur visuell ausgeben, sondern auch als Datenpaket exportiert werden. Dies könnte besonders bei Scapy Anwendung finden. Scapy ist ein open source Projekt, welches unter anderem als Penetrationstest Werkzeug im Automotive Bereich verwendet werden kann.

6.2 Fazit

Das Ziel dieser Bachelorarbeit war das automatisierte Finden von zusammengehörenden Datenpaketen im CAN Datenfeld. Es wurde versucht dieses Ziel mit der Implementierung von einem Programm zu erfüllen, welches zwei Metriken benutzt, um die CAN Daten zu analysieren. Diese Metriken sind die Bitflips und der Quotient. Die Analyse der Daten kann sowohl mit nur einer dieser Metriken, als auch mit beiden durchgeführt werden. Das beste Ergebnis wurde erzielt, indem zuerst mit dem Quotienten Felder gesucht werden und anschließend mit den Bitflips zusätzlich getestet wird, ob die von dem Quotienten gefundenen Felder aus mehreren kleinen Feldern bestehen.

Trotz alledem gibt es einige Fälle, in denen Felder nicht optimal erkannt werden, was verschiedene Gründe haben kann. Besonders problematisch ist, dass bei den Parametern der Metriken Fehler auftreten können, wenn bei einem spezifischen Wert Felder nicht erkannt werden, obwohl diese zusammen gehören. Gleichzeitig werden mit den selben Parametern andere Felder, die nicht zusammen gehören, fälschlicherweise als zusammenhängend angesehen.

Daraus folgt, dass auf die automatisierte Erkennung zusätzlich ein manuelles Überprüfen, ob diese Datenfelder auch wirklich zusammen gehören können, folgen muss.

Abbildungsverzeichnis

Literatur

[ATE] Philip Axer, Daniel Thiele, and Rolf Ernst. Requirements on real-time-capable automotive ethernet architectures. https://www.sae.org/publications/technical-papers/content/2014-01-0245/preview/. [Online; besucht am 08.03.2020].

[BBM07] M. Bertoluzzo, G. Buja, and R. Menis. Control schemes for steer-by-wire systems. *IEEE Industrial Electronics Magazine*, 1(1):20–27, Spring 2007.

[BMV] BMVI. Automatisiertes und vernetztes fahren. https://www.bmvi.de/DE/Themen/Digitales/Automatisiertes-und-vernetztes-Fahren/automatisiertes-und-vernetztes-fahren.html. [Online; besucht am 25.02.2020].

[Eil13a] Eldad Eilam. *Reversing: secrets of reverse engineering*, page xxiv. Wiley, 2013.

[Eil13b] Eldad Eilam. *Reversing: secrets of reverse engineering*, page 110 f. Wiley, 2013.

[Eil13c] Eldad Eilam. *Reversing: secrets of reverse engineering*, page 116 f. Wiley, 2013.

[Eil13d] Eldad Eilam. *Reversing: secrets of reverse engineering*, page 129 f. Wiley, 2013.

[GRV] GRVA. Draft recommendation on cyber security of the task force on cybersecurity and over-the-air issues of unece wp.29 grva. https://www.unece.org/fileadmin/DAM/trans/doc/2018/wp29grva/GRVA-01-17.pdf. [Online; besucht am 02.03.2020].

[Har] Oliver Hartkopp. The can subsystem of the linux kernel. https://wiki.automotivelinux.org/_media/agl-distro/agl2017-socketcan-print.pdf. [Online; besucht am 10.01.2020].

[HMVV13] P. Hank, S. Müller, O. Vermesan, and J. Van Den Keybus. Automotive ethernet: In-vehicle networking and smart mobility. In *2013 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1735–1739, March 2013.

[IEC] IEC. Functional safety. https://www.iec.ch/functionalsafety/explained/. [Online; besucht am 03.03.2020].

[ISOa] ISO. Iso 13400-1:2011 road vehicles — diagnostic communication over internet protocol (doip) — part 1: General information and use case definition. https://www.iso.org/standard/53765.html. [Online; besucht am 08.03.2020].

[ISOb] ISO. Iso 26262-1:2018(en) road vehicles — functional safety — part 1: Vocabulary. https://www.iso.org/obp/ui/#iso:std:iso:26262:-1:ed-2:v1:en. [Online; besucht am 03.03.2020].

[ISOc] ISO. Iso/sae dis 21434(en) road vehicles — cybersecurity engineering. https://www.iso.org/obp/ui/#iso:std:iso-sae:21434:dis:ed-1:v1:en. [Online; besucht am 03.03.2020].

[KK+12] Mohd Ehmer Khan, Farmeena Khan, et al. A comparative study of white box, black box and grey box testing techniques. *Int. J. Adv. Comput. Sci. Appl*, 3(6), 2012.

[SA16] P. S. Shinde and S. B. Ardhapurkar. Cyber security analysis using vulnerability assessment and penetration testing. In *2016 World Conference on Futuristic Trends in Research and Innovation for Social Welfare (Startup Conclave)*, pages 1–5, Feb 2016.

[SJ08] R. Shaw and B. Jackman. An introduction to flexray as an industrial network. In *2008 IEEE International Symposium on Industrial Electronics*, pages 1849–1854, June 2008.

Erklärung

1. Mir ist bekannt, dass dieses Exemplar der Bachelorarbeit als Prüfungsleistung in das Eigentum der Ostbayerischen Technischen Hochschule Regensburg übergeht.
2. Ich erkläre hiermit, dass ich diese Bachelorarbeit selbstständig verfasst, noch nicht anderweitig fÃijr Prüfungszwecke vorgelegt, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie wörtliche und sinngemäße Zitate als solche gekennzeichnet habe.

Ort, Datum und Unterschrift

Vorgelegt durch:	Stefan Schönhärl
Matrikelnummer:	3101180
Studiengang:	Bachelor Technische Informatik
Betreuung:	Prof. Dr. Rudolf Hackenberg
Zweitbegutachtung:	Prof. Dr. Frank Hermann