universitätbonn

Rheinische
Friedrich-Wilhelms-
Universität Bonn

Bachelorarbeit

Heuristische CPU Architekturerkennung anhand architekturspezifischer Charakteristika

Marcel Johann Aniol

2542907

17.11.2016

Erstgutachter: Prof. Dr. Michael Meier
Zweitgutachter: Prof. Dr. Peter Martini

Institut für Informatik IV

Mathematisch-Naturwissenschaftliche Fakultät

Selbstständigkeitserklärung

Hiermit versichere ich, die vorliegende Bachelorarbeit ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Bonn, den 17.11.2016

Marcel Johann Aniol

Danksagung

Ich bedanke mich bei meiner Familie, die mich im Laufe meines Studiums immer unterstützt und zu mir gehalten hat. Dann bedanke ich mich noch bei meinen Betreuern Peter Weidenbach und Johannes vom Dorp, die mich bei der Erstellung dieser Arbeit fachlich begleitet haben.

Inhaltsverzeichnis

1 Einleitung

Mit den NSA Enthüllungen im Jahr 2013 von Edward Snowden wurde ein Zeitalter internationalem Misstrauens gegenüber Geheimdiensten eingeleitet. Es war anzunehmen, dass Geheimdienste Mittel zur Überwachung besitzen, allerdings wurde nicht damit gerechnet, dass davon global in solchem Umfang Gebrauch gemacht wird [10]. Die Beschaffung der Informationen kann dabei vielfältig verlaufen. Neben der Auswertung von Daten ausgewählter Konzerne, ist auch das Verteilen von manipulierter Software oder Hardware nicht auszuschließen.

Am 16 Februar 2015 veröffentlichte Reuters einen Artikel, welcher dokumentiert, dass russische Wissenschaftler von Kaspersky Lab herausgefunden haben, dass Mitarbeiter der NSA es geschafft haben Festplatten von namhaften Herstellern mit Spionage Software zu infizieren. Laut Kaspersky wurde damit ein technologischer Durchbruch geschafft, da die Schadsoftware sich nun nicht mehr im Speicher der Festplatte befindet, sondern in der Firmware der Hardware selbst [28].
Mit der steigenden Bedeutung von eingebetteten Systemen, ist diese Erkenntnis ein Alarmsignal für Sicherheitsforscher. Laut einer Publikation der Bitkom [8] erkennt diese, Eingebettete Systeme in der Industrie 4.0 als ein strategisches Wachstumsfeld für Deutschland [9]. Allerdings zeigen Artikel wie von *"/dev/ttys0"*, dass selbst bei namhaften Herstellern, die Sicherheit in der Firmware nicht immer garantiert werden kann [15]. Das macht Möglichkeiten zur Analyse von Firmware auch für Dritte interessant.

Bei der Analyse von Firmware treten in der Regel andere Fragestellungen auf, als bei der Analyse von Software. So weiß man zu Beginn der Analyse häufig nicht in welchem Format sich die Datei befindet oder auf welcher CPU Architektur das Programm in der Datei ausführbar ist. Es ist möglich, dass bei einigen Geräten die technische Dokumentation von den Herstellern absichtlich verschleiert wird, um solche Analysen gar nicht erst zu ermöglichen. Zudem wird in Kapitel 2 gezeigt, dass insbesondere bei Eingebetteten Systemen der Einsatz unterschiedlicher CPU Architekturen allgegenwärtig ist. Statische Analyse Werkzeuge, wie zum Beispiel *IDA* [20] sind größtenteils abhängig von der Kenntnis der CPU Architektur. Ohne die Kenntnis der CPU können auch keine dynamischen Analysen auf Programmen ausgeführt werden, was den Einsatz von Debuggern nicht möglich macht. Im Gegensatz zu gängiger Windows und Linux Software trifft man in Firmware häufig auf Formate, die weitgehend unbekannt sind und die CPU Architektur dementsprechend nicht direkt offensichtlich ist. Dieser Punkt wird in Kapitel 2 genauer erläutert. Diese Arbeit widmet sich deshalb der Erkennung von CPU Architekturen. Ziel ist dabei die Entwicklung eines Verfahrens, welches eine beliebige Binärdatei einliest und so anhand ausgewählter Kriterien die Architektur, auf der die Datei ausführbar ist, erkennen kann.

Grundlage zur Architekturerkennung bietet der *Codescanner* [32], welcher zur Erkennung von *x86 Code* in Dateien eingesetzt wird. Das Verfahren operiert dabei auf Byte

Ebene und ist so unabhängig von Meta Informationen in der Datei. Diese Eigenschaft sowie die präzise Erkennungsrate, machen die verwendeten Konzepte in dem Werkzeug auch für die Architekturerkennung interessant. Die Idee zur Architekturerkennung besteht darin, potenziellen Code in der Datei einer CPU Architektur zuzuordnen.

Es wurden bereits Verfahren entwickelt, die dem Problem der Architekturerkennung entgegenkommen, allerdings konnte diesbezüglich die Zuverlässigkeit noch nicht belegt werden. Ziel ist deshalb eine hohe Erkennungsrate zu gewährleisten und das Werkzeug anschließend mit existierenden Tools zu vergleichen. Tools die dabei in Frage kommen sind unter anderem *binwalk* [14] und ein Ansatz unter Verwendung von maschinellem Lernen [11]. Für die Evaluation wird eine große Datenmenge aus einem Linux Repository erzeugt. Die erzeugte Datenmenge soll mehrere Architekturen abdecken und nach Durchlauf, die Erkennungsrate der Verfahren feststellen. Damit man eine Erkennung anhand von Metadaten ausschließen kann, werden die Programme jeweils so modifziert, dass ein signifikanter Teil, der Metadaten enthalten könnte abgeschnitten wird. In der Evaluation werden auch Schwellwerte und Referenzwerte, welche die Erkennungsrate des Verfahrens beeinflussen, generiert. Bei Fehlklassifizierungen wird anschließend untersucht, was die Ursache für die Fehlklassifizierung sein könnte. Abschließend werden alle Werkzeuge aus ökonomischer Sicht betrachtet.

In Kapitel 2 werden zunächst einige *Grundlagen* über die Firmware Analyse näher gebracht. Außerdem werden Kriterien, sowie Architekturen die das Verfahren beeinflussen genauer betrachtet. In Kapitel 3 werden *verwandte Arbeiten* vorgestellt, sowie bewertet. In Kapitel 4 wird das *Konzept*, welches Grundlage zur Architekturerkennung bietet, genauer erläutert. Darauf aufbauend wird in Kapitel 5, die *technische Umsetzung* des Konzepts unter anderem in Pseudocode erläutert. In Kapitel 6 und 7 wird die Güte des Verfahrens mit anderen Ansätzen verglichen und ausgewertet. In Kapitel 8 werden die Ergebnisse zusammengefasst und daraus ein Fazit gezogen. Abschließend wird in Kapitel 9 ein Ausblick über das Verfahren gegeben.

2 Grundlagen

In diesem Kapitel werden Grundlagen vermittelt, die für das weitere Verständnis der Arbeit von Bedeutung sind. Unter anderem wird zuerst genauer in das Forschungsfeld der Firmware Analyse eingeführt, um die Bedeutung der Architekturerkennung noch einmal hervorzuheben und die Reichweite, sowie gängige Praktiken genauer darzustellen. Dabei werden auch die untersuchten Architekturen in ihren Gemeinsamkeiten und Unterschieden untersucht und Umstände, die eine Erkennung beeinflussen genauer betrachtet.

2.1 Firmware

IEEE definiert Firmware als eine "Read-Only-Memory", kurz *ROM* basierte Software Komponente, die in der Zeit zwischen dem An und Ausschalten bis zur Kontrollübernahme des primären Betriebssystems, die Kontrolle über den Computer hält. Zu den Aufgaben gehört unter anderem das Testen, sowie die Konfiguration der Hardware, die Möglichkeit zum Debuggen, das Management über den Bootvorgang und vieles mehr [1]. Allerdings ist die Bedeutung von Firmware weitreichender als die Definition von IEEE einem suggeriert. So kann die Firmware auch die grundlegende Betriebssoftware des Gerätes sein und in der Hardware eingelötet sein [37]. Firmware kann sehr vielfältig sein, wie in diesem Abschnitt gezeigt wird. Zunächst soll allerdings der Zusammenhang zwischen Firmware und Eingebetteten Systemen hergestellt werden.

2.1.1 Eingebettete Systeme

Eingebettete Systeme sind Computersysteme, welche mittels eigenem Mikroprozessor, sowie weiteren Hardware Komponenten und Kommunikationsschnittstellen, Aufgaben in größeren oder kleineren Installationen übernehmen können [36]. Auf Eingebetteten Systemen kommt Firmware zum Einsatz, die die interne primäre Steuerung der Schnittstellen übernimmt. In der Industrie kommen Eingebettete Systeme zum Beispiel in Anlagen oder Maschinen vor [25]. Aber auch im nicht industriellen Umfeld sind Eingebettete Systeme allgegenwertig, weitere Beispiele sind [41]:

- Haushaltsgeräte: Kühlschränke, Waschmaschinen
- Multimediageräte: MP3 Player, DVD Player, Kameras, Handys, Smartphones
- Steuereinheiten: Autos, Flugzeuge, industrielle Anlagen
- Überwachungsanlagen: IP-Kameras, Alarmanlagen

Die Liste lässt sich fortsetzen, allerdings wird darauf verzichtet, da hierdurch klar werden soll, welchen Stellenwert Eingebettete Systeme in unserem Umfeld erreicht haben. Dabei haben die Einsatzszenarien unterschiedliche Anforderungen, was Einfluss auf die installierte Firmware der Systeme hat. Hardware Komponenten die zum Einsatz kommen sind unter anderem Prozessoren, flüchtige Speichermedien, persistente Speichermedien, Datenbüsse, Kommunikationsschnittstellen wie Ethernet oder Bluetooth und

vieles mehr. Die Firmware hat die Aufgabe mit den verbauten Schnittstellen zu kommunizieren und gegebenenfalls Treiber bereitzustellen, um die Schnittstellen in Betrieb zu nehmen [41]. Bei individuellen Systemen können die Anforderungen an die Firmware also unterschiedlich sein. So haben sich zum Beispiel *Arduino* [5] mit ihren eigenen Microcontrollern und die *Raspberry Pi* [30], welche ARM Prozessoren verbaut haben, zunehmend in der Bastlerszene beliebt gemacht.

2.1.2 Firmware Dateien

Die Firmware Datei die man umgangssprachlich auch als *Image* bezeichnet, kann in unterschiedlichen Formaten zur Verfügung gestellt werden. Häufig wird das Firmware Image von den Herstellern als *BLOB* (Binary large object) bereit gestellt. Ein *BLOB* ist eine große Binärdatei, welche nicht primär nach einem bestimmten Format charakterisiert wird [34]. In diesem Zustand wird das Firmware Image dann initial auf das Speichermedium eingespielt. Speichermedien können dabei Festplatten, SD-Karten aber auch fest gelötete ROM Speicher sein [41].

Der Inhalt des Image unterscheidet sich, je nach Einsatzszenario. Firmware kann für komplexere Aufgaben ein ganzes Betriebssystem wie Linux zur Verfügung stellen, aber auch nur ein auf die Anwendung zugeschnittenes System enthalten. Komplexere Firmware Images können dabei unter anderem mit Bootloadern, Kerneln, Dateisystemen oder Servern für bestimmte Aufgaben ausgestattet sein, sodass ein großer Bestandteil der Firmware, Programme enthalten könnte. Weniger komplexe Firmware Images können dagegen zum Beispiel auch nur eine Auswahl an gekapselten Anwendungen implementieren um die Anforderungen des Einsatzzwecks zu erfüllen. So können Firmware Images auch Dateisysteme implementieren, einige Beispiele sind [41]:

- SquashFS
- CramFS
- NTFS
- YAFFS
- ext Dateisysteme

Softwarekomponenten der Firmware, sowie die Firmware selbst, können auch in komprimierten Archiv Formaten verpackt sein und sich mithilfe eines Tools beim Installieren selbst entpacken. Häufig ist deshalb auch der Einsatz von Kompressionswerkzeugen notwendig. So sind Archiv Formate wie *Ar, Tar, Gzip, RPM, DEB* und weitere auch in Firmware wiederzufinden [41].

Firmware Images können auf verschiedenen CPU Architekturen ausführbar sein, so sind für manche Szenarien mächtige Prozessoren, wie *Intels* x86 Prozessor nicht notwendig und ein *ARM* Microcontroller könnte zum Beispiel aus Gründen wie Energie Effizienz

besser geeignet sein. Mittlerweile existiert eine Vielzahl von verschiedenen Mikrocontrollern und Prozessoren. Nur einige Beispiele sind [27]:

- Alpha
- ARM
- Burroughs B5000/B6000/B7000 series
- IA-64
- MIPS
- Motorola 68k
- IBM 700/7000 series
- PowerPC
- ...

Dabei implementieren die meisten Prozessoren unterschiedliche Befehlssätze und haben alle ihre Eigenschaften. Auf einige dieser Prozessoren wird in Kapitel 2.3 genauer eingegangen. Die hohe Anzahl an verschiedenen Prozessorarchitekturen in Kombination mit unbekannten Firmware Formaten zeigt, dass der Einsatz von Klassifizierungsverfahren durchaus wichtig ist.

2.2 Statische Firmware Analyse

Da die Architekturerkennung, wie sie in dieser Arbeit angestrebt wird als statisches Analyse Verfahren umgesetzt wird, soll in diesem Abschnitt auf Aspekte in der statischen Analyse von Firmware eingegangen werden, die auch für die CPU Architekturerkennung von Bedeutung sind. Bei der Firmware Analyse kann man mit Dateiformaten an Meta Informationen gelangen, welche wichtige Informationen über die Datei preisgeben können. So kann man anhand von Dateiformaten, wie *PE* oder *.ELF* die Architektur, den Einstiegspunkt in das Programm, die *Endianness*, sowie weitere grundlegende Informationen erhalten [29] [12].

Sollten vor der Analyse keine Meta Informationen vorliegen, so kann es sein das vor der eigentlichen Architekturerkennung einige Vorraussetzungen erfüllt sein sollten, da zum Beispiel das Verfahren angewendet auf verschlüsselte oder komprimierte Container, falsche Ergebnisse erzielen könnte [41]. Mithilfe von Verfahren aus der *Informationstheorie*, kann man allerdings erste Antworten auf die Frage bekommen in welchem Initial Zustand sich die Datei befinden könnte ohne bekannte Dateiformate zu kennen. Diese Konzepte gehen allerdings auch weiter und können auch zur Architekturerkennung eingesetzt werden, weshalb darauf genauer eingegangen wird.

2.2.1 Informationsgehalt

In der Praxis angewendet kann so beim Einlesen von Byte Blöcken einer bestimmten Größe, die Entropie berechnet werden, um Rückschlüsse über bestimmte Abschnitte in der Datei zu ziehen. Beispielsweise können so Regionen erkannt werden, die keine Informationen enthalten, wie Blöcke die immer wieder gleiche Byte Sequenzen enthalten. Es können aber auch Regionen erkannt werden, die ein hohes Maß an Informationen besitzen und so unterschiedlich charakterisiert werden. Dabei kommen Blöcke in Frage die zum Beispiel Code enthalten oder verschlüsselte Signaturen. Mit dieser Vorgehensweise kann man die Datei in verschiedene Regionen unterteilen, welche so sukzessiv weiter untersucht werden können. Abbildung 1 zeigt beispielhaft verschiedene Ergebnisse die durch Anwendung der Shannon Entropie auf bestimmte Regionen in einer Datei entstehen. So ist durch den Graphen gut zu erkennen in welchen Offsets in (1) Regionen sind, die einen niedrigen beziehungsweise ein hohes Informationsgehalt besitzen.

2.2.2 Reverse Engineering

Reverse Engineering ist die Untersuchung von kompilierter Software und ein weiterer statischer Analyseansatz. Ziel ist dabei das Verständnis über die Abläufe, die in der Software passieren. Dafür muss der Quelltext analysiert werden. Damit das möglich ist, muss dieser zuerst aus dem Maschinencode erzeugt werden. Dieser Vorgang wird als *Dekompilieren* bezeichnet. Dekompilieren ist die Königsklasse beim Reverse Engineering und nur schwer zu realisieren, da der Maschinencode zuerst in eine Assemblersprache übersetzt werden muss und von der Assemblersprache noch zum Quellcode. Nun gibt es aber viele Interpretationsmöglichkeiten bei der Übersetzung von Assemblercode zum Quelltext. Wie der Quelltext zum Maschinencode übersetzt wird, hängt allein vom *Compiler* ab. Dieser Vorgang ist dabei zu vergleichen mit einem *Hashverfahren*. Bei Hashverfahren wird eine Eingabe durch eine Hashfunktion in eine eindeutige Ausgabe transformiert. Ziel ist dabei, dass die Ausgabe aber nicht wieder zur Eingabe zurück transformiert werden kann. Aus diesem Grund werden Analysen von Spezialisten in der Regel schon auf Assembler Ebene durchgeführt. Die Übersetzung von Maschinencode zu Assemblercode wird dabei als *Disassemblieren* bezeichnet [16]. Disassemblierer wie zum Beispiel *IDA* [20], führen zunächst analytische Schritte durch, bevor der Maschinencode in eine Assemblersprache übersetzt werden kann. Ein Assemblerbefehl setzt sich aus einer Operation und den zugehörigen Operanden zusammen. Die Operation wird auch *Mnemonic* genannt. Die Operation selbst wird durch einen *Opcode* in Maschinensprache kodiert [27].

IDA ist in der Lage eine Vielzahl von Informationen aus bekannten Dateiformaten zu erzeugen, um unter anderem die CPU Architektur auf der die Datei ausführbar ist herauszufinden, den Einstiegspunkt in das Programm sowie Regionen zu identifizieren, die Code enthalten. Darauf aufbauend werden dann Regionen, welche als Code identifiziert wurden, mit einem rekursiven Algorithmus von der Maschinensprache, Instruktion für Instruktion in die Assemblersprache übersetzt. Für diesen Vorgang ist die Kenntnis über die Architektur eine Voraussetzung, damit die Byte Sequenzen in die richtigen Opera-

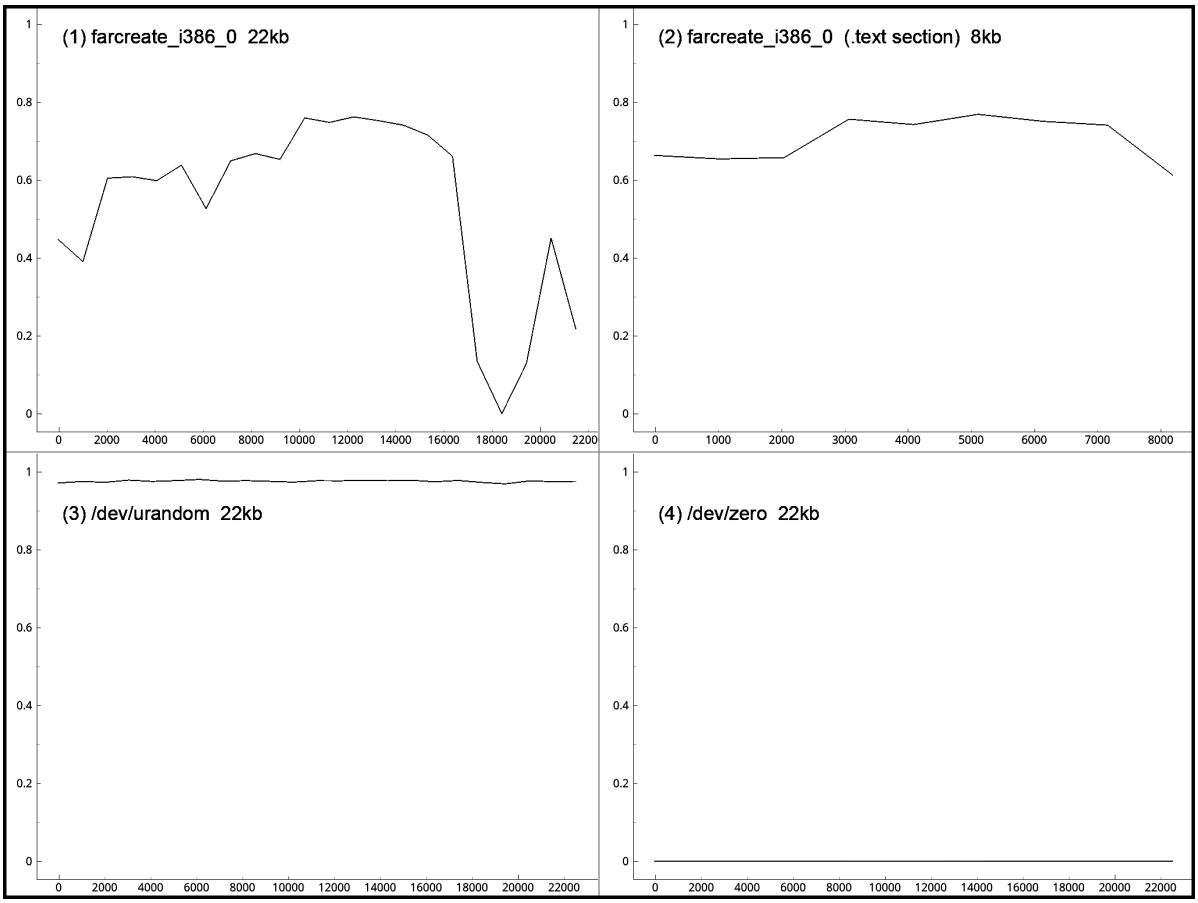

Abbildung 1: In dieser Abbildung soll mit Binwalk beispielhaft dargestellt werden wie man aus der Entropie, angewendet auf eine Datei, potenzielle Informationen über die Datei erhalten kann. So ist in (1) der Entropie-Graph eines ELF Binaries dargestellt und in (2) nur die Region die Code enthält. In (3) und (4) sollen Extrema dargestellt werden, welche verschlüsselten Signaturen (3) oder Inhaltslosen Bereichen wie '0xFF' Sequenzen (4) entsprechen könnten. Die x-Achse stellt den Offset dar, die y-Achse von 0-1 die totale Entropie.

tionen mit ihren Operanden übersetzt werden können [16].

Das Problem ist, dass Disassemblierer wie IDA bei unbekannten Dateitypen mit zusätzlichen Informationen vom Benutzer selbst ausgestattet werden müssen. Die wichtigste Information die IDA braucht und in der Regel nicht in der Lage wäre zu erkennen, ist die CPU Architektur, auf der die Datei ausführbar ist. Die CPU Architektur müsste also unabhängig von IDA ermittelt werden, was mit dieser Arbeit erreicht werden könnte. Andernfalls könnte IDA den Maschinencode auf Grundlage eines falschen Befehlssatzes übersetzen, was in falschen Resultaten enden würde. Ohne den Einstiegspunkt, wäre der Disassemblierer mit Angabe der richtigen CPU Architektur schon zumindest in der Lage die Datei *linear* zu disassemblieren, was nicht das optimale Ergebnis liefern könnte, aber man seinem Ziel doch signifikant näher kommen würde. Bei der linearen Disassemblierung wird der Maschinencode, Instruktion für Instruktion übersetzt, ohne auf Sprünge oder Verzweigungen einzugehen.

2.3 CPU Architekturen

Da für diese Arbeit das Verständnis von CPU Architekturen ausschlaggebend ist, werden in diesem Kapitel, Unterschiede und Gemeinsamkeiten der betrachteten CPU Architekturen, sowie Eigenschaften, die für die Architekturerkennung wichtig sind genauer betrachtet. Dabei wird nur auf die wichtigsten Punkte eingegangen, da die genauen Spezifikationen einer CPU Architektur zu große Mengen an Informationen bieten, die in dieser Arbeit nicht verarbeitet werden können. Ziel ist es eine Basis herauszuarbeiten, die zwecks Architekturerkennung, die Architekturen nach ihren Charakteristiken differenzieren kann. Zusätzlich wird erläutert, welchen Einfluss die Byte Reihenfolge auf die Erkennung von CPU Architekturen haben kann.

Der Prozessor ist die zentrale Recheneinheit in jedem Computer und für die korrekte Ausführung von Maschinenbefehlen verantwortlich. Maschinenbefehle sind Instruktionen, welche zu einem Befehlssatz einer Architektur gehören. Eine Instruktion besteht in der Regel aus einer Operation und einem bis mehreren Operanden und wird durch den Prozessor sequentiell abgearbeitet [27]. Zunächst werden aber die wesentlichen *Design Philosophien* moderner Prozessoren erläutert.

2.3.1 Design

Bei der Charakterisierung von Prozessoren gibt es zwei Design Philosophien, die mit den korrespondierenden Befehlssätzen in Verbindung gebracht werden können. Dabei handelt es sich konkret um *RISC* (Reduced Instruction Set Computing) und *CISC* (Complex Instruction Set Computing). Ziel bei der RISC Philosophie ist es die Laufzeit dadurch gering zu halten, dass Instruktionen im Befehlssatz schnell ausführbar sind und der Befehlssatz möglichst gering gehalten wird. Zusätzlich findet die Adressierung direkt auf Registern mit Lade und Speicher Befehlen statt. So ist zum Beispiel ARM ein bekannter Vertreter der RISC Philosophie [6]. Bei der CISC Philosophie kommen große

Befehlssätze mit 200-400 Instruktionen zum Einsatz, so ist das Ziel dabei mit möglichst wenigen Operationen, komplexe Instruktionen auszuführen. Im Gegensatz zu RISC kann bei CISC von Speicher zu Speicher adressiert werden [26].

Allerdings ist heute eine konkrete Klassifizierung nicht mehr möglich, da Hersteller versuchen das beste von beiden Philosophien zu implementieren [26].

2.3.2 x86 und IA-32/64

Der x86 Befehlssatz wurde 1978 mit dem 8086 Prozessor von Intel eingeführt und seitdem immer weiter entwickelt. Ziel war es die 8 Bit Architektur durch eine 16 Bit Architektur zu ersetzen. Nun hat sich über die Jahre der physische Adressraum von 16 bis zu 64 Bit erweitert. Die Architektur wird mit den IA-32/64 Prozessoren überwiegend von Intel vertrieben, allerdings verkaufen auch andere Hersteller wie AMD, Prozessoren mit dem x86 Befehlssatz. Neben dem Einsatz in Personal Computern für Heimanwender, wird der x86 Befehlssatz auch im Eingebetteten Bereich eingesetzt, wie zum Beispiel Intels Atom Prozessor [40].

Der x86 Befehlssatz hat eine variable Instruktionslänge. Das bedeutet, dass Instruktionen zwischen zwei und acht Byte lang werden können [24]. Variable Instruktionen können die Arbeit für Disassemblierer schwieriger machen, da zum Beispiel eine acht Byte lange Instruktion auch als zwei vier-Byte Instruktionen interpretiert werden könnte, wenn die Sequenz an der falschen Stelle disassembliert wird. Der folgende Assembler Ausschnitt aus [32] verdeutlicht das Problem genauer:

Listing 1: Fehldisassemblierung

```
1
2     Korrekte Interpretation
3
4     83 ff 15     cmp edi,0x15
5     74 0a        jz $+0x8
6     64 00        push 0
7
8     Ab dem falschen Byte disassembliert
9
10    ff 15 74 0a 64 00     call dword ptr[0x640a74]
```

Dadurch das der x86 Befehlssatz der CISC Philosphie folgt, umfasst der Befehlssatz sehr viele individuelle Operationen. So arbeitet der x86 Befehlssatz konkret mit Stack Operationen wie *pop* oder *push*, welche bei anderen Prozessoren anders realisiert werden. Mit Hilfe einer Validierungsmenge, welche mit dem Capstone Disassemblierer [31], disassembliert wurde, konnten bis zu 175 verschiedene Operationen gefunden werden. Das zeigt den enormen Umfang des x86 Befehlssatzes. Die Idee hinter der x86 Architektur ist, dass sich dieser Befehlssatz auch aus komplexen Operationen zusammensetzt und folglich Charakteristika der CISC Philosophie folgt. Die 64 bit Architektur ist zu 32 bit Prozessoren abwärtskompatibel und implementiert zusätzlich noch weitere Operationen.

Die Speicheraddressierung wird durch eine Speicherverwaltungseinheit, der *MMU* (Memory Management Unit) im Prozessor geregelt. Die Opcodes im x86 Befehlssatz können bis zu zwei Byte lang sein und werden je nach Operation an unterschiedlicher Stelle in der Instruktion kodiert [24].

2.3.3 MIPS

Die Mips Architektur wurde 1981 von John L. Hennessy entwickelt und wird heute von Imagination Technologies lizensiert. MIPS Prozessoren sind einfache, hoch skalierbare Prozessorarchitekturen, welche der RISC Philosophie folgen. MIPS Prozessoren gibt es mit 32 und 64 Bit Adressbüssen und einem vergleichsweise kleinen Befehlssatz [38]. Im Gegensatz zu x86 verfügt der MIPS Befehlssatz keine speziellen Operationen für Stack Operationen, sondern regelt Stack Operationen über die herkömmlichen Lade und Speicher Befehle in den Registern [22].

MIPS Prozessoren verfügen über feste Instruktionslängen von vier Byte. Sechs Byte in der Instruktion repräsentieren den Opcode. Der MIPS Befehlssatz verwendet *'General purpose'* Instruktionen. Dazu gehören unter anderem arithmetische Operationen, Lade und Speicher Befehle, logische Operationen, Bedingungen und Sprünge, so wie Shift und Rotationsoperationen [22].

2.3.4 ARM

Die ARM Architektur wurde 1983 entwickelt und vergibt ähnlich wie MIPS, Lizenzen an Chip Hersteller. Geschätzt sind heute bis zu 37 Milliarden Prozessoren von ARM im Umlauf [19]. Die Prozessoren werden für verschiedene Szenarien entwickelt, so können durch die Cortex Serie, Modelle erworben werden die einerseits niedrige Energiekosten bei niedrigerer Leistung verbrauchen, aber auch Prozessoren die speziell auf hohe Lasten im Computing Bereich ausgelegt sind [7]. Dadurch können ARM Prozessoren flexibel eingesetzt werden. Im Jahre 2013 wurde mit der Revision Acht, die ARM Architektur auf 64 Bit erweitert [35].

Der Befehlssatz von ARM kann zwischen einem allgemeingültigen und dem *THUMB* Befehlssatz unterschieden werden. Der allgemeingültige Befehlssatz kann bei arithmetischen Operationen bis zu drei Register adressieren und hat in der Regel Vier Byte große Instruktionen. Dieser Befehlssatz findet vor allem bei älteren Portierungen von Firmware oder Betriebssystemen Verwendung. Der THUMB Befehlssatz dagegen, besteht aus 16 Bit breiten Befehlen und soll anhand dieser Größe den Speicherbedarf verringern, erhöht aber dafür zu Kosten die Programmdichte. Programme für ARM ab Revision Vier, vermischen den THUMB und den älteren ARM Befehlssatz, was als *Thumb Interworking* bezeichnet wird und zu variablen Instruktionslängen führen kann. Das bedeutet, dass die Instruktionen zwischen Zwei und Vier Byte lang sind. Der Opcode von ARM Prozessoren ist an unterschiedlichen Stellen kodiert, so kann er am Anfang einer Instruktion stehen aber auch am Ende. Die Länge kann sich dabei unterscheiden. Generell folgt die ARM

Architektur der RISC Philosophie, weshalb der Befehlssatz auch überwiegend 'General Purpose' Operationen implementiert [35] [6].

2.3.5 PowerPC

Die PowerPC Architektur ist eine im Jahr 1991 in den USA entwickelte CPU Architektur und findet sowohl in Workstations, wie auch in Eingebetteten Systemen Verwendung. Prominente Vertreter der PowerPC Architektur sind unter anderem die aktuellen Spielekonsolen GameCube, Wii, Wii U von Nintendo, wie auch die Xbox 360 von Microsoft oder die Playstation 3 von Sony. Die PowerPC Architektur unterstützt neben 32 Bit mittlerweile auch eine erweiterte 64 Bit Architektur [39].

PowerPC unterstützt feste Instruktionslängen von Vier Byte und verzichtet wie MIPS und ARM auf spezielle Stack Operationen. Diese werden durch Lade und Speicher Befehle in die jeweiligen zuständigen Register geregelt. PowerPC folgt wie ARM und MIPS der RISC Philosophie und implementiert folglich eine Auswahl an 'General purpose' Operationen [33]. Eine Beobachtung mit IDA [20] ergab, dass die Opcodes nicht eindeutig an einer festen Stelle in der Instruktion kodiert sind und auch für namensgleiche Operationen unterschiedlich kodiert sind.

2.3.6 Byte Reihenfolge

Zuletzt soll noch auf die Byte Reihenfolge eingegangen werden, da diese auch einen signifikanten Einfluss auf die verbauten Prozessorarchitekturen hat. Die Byte Reihenfolge gibt an, in welcher Reihenfolge die Byte Werte in einer Datei gespeichert und vom Prozessor ausgewertet werden. Grundsätzlich wird zwischen Little und Big Endian unterschieden. So speichert Little Endian die Bytes nach dem niederwertigsten Byte zuerst. Big Endian speichert die Bytes nach dem höchstwertigen Byte zuerst [4]. In Listing 2 ist angegeben wie die Integer Zahl **10012016** in hexadezimal **00 98 C5 70** in Little Endian, sowie in Big Endian gespeichert wird. Für bestimmte Prozessoren spielt die Byte Reihenfolge keine große Rolle mehr, da sie in der Lage sind zwischen Little oder Big Endian umzuschalten oder nur auf einem von beidem Programme ausführen können. So sind ausgewählte Modelle (Beispiel: *Cortex-A8*) von ARM über ein Status Bit in der Lage, Daten sowohl als Big und Little Endian zu interpretieren, Instruktionen aber immer als Little Endian [23]. Einige Modelle der PowerPC Architektur können dagegen ganz zwischen Little Endian und Big Endian umschalten [21].

Bei der statischen und dynamischen Analyse von Dateien spielt die Byte Reihenfolge dagegen eine wichtigere Rolle. Disassemblierer wie Capstone brauchen die Angabe der Byte Reihenfolge als Parameter, anderseits wird der Maschinencode fehlerhaft oder gar nicht disassembliert

Listing 2: Little und Big Endian

```
1
2     10012016 in hexadezimal: 00 98 C5 70
3
4     Little Endian 70 C5 98 00
5
6     Big Endian 00 98 C5 70
```

3 Verwandte Arbeiten

In diesem Kapitel werden zunächst verwandte Arbeiten vorgestellt, die die Erkennung von CPU Architekturen in beliebigen Dateien thematisieren und aus den Erkenntnissen abgeleitet, welche Werkzeuge für das entstandene Verfahren eine Grundlage bieten können.

3.1 Architekturerkennung mit Metadaten

Binärdateien können in Abhängigkeit von Informationen in der Datei, unterschiedlich zuverlässig auf ihre Architektur klassifiziert werden. *Firmadyne* [18] ist ein Tool, welches zur dynamischen Analyse von Linux Firmware entwickelt wurde und implementiert unter anderem auch ein statisches Modul zur Architekturerkennung. Dabei bezieht sich Firmadyne auf das Linux Kommando *file*, welches anhand von Meta Informationen im ELF Dateiheader einer beliebigen ELF Datei, die Ziel Architektur abliest. Die Informationen über die Architektur sind dabei, bei allen ELF Dateien in einem festen Offset kodiert [12].

Dieses Verfahren funktioniert mit absoluter Sicherheit, vorausgesetzt die Quelldatei liegt im ELF Format vor. ELF ist aber nicht das einzige Dateiformat, welches die Quellarchitektur im Dateiheader kodiert hat. Es gibt noch eine größere Auswahl an Formaten, wie zum Beispiel PE die diese oder ähnliche Informationen im Dateiheader kodiert [29]. Wie bereits in Kapitel 2 erläutert wurde, kann Firmware aber auch ganz unterschiedliche, unter Umständen auch kaum dokumentierte Dateiformate, beziehungsweise eigene Dateiformate implementieren. Das würde diesen Ansatz an seine Grenzen führen. Da dieser Ansatz wegfällt, ist die einzige Möglichkeit alle Formate abzudecken ein statisches Analyseverfahren, welches die Informationen zum Beispiel anhand struktureller Charakteristika auf Maschinencode Ebene zurückführt.

Eine Firmware Datei setzt sich abstrakt aus einem Datenanteil, dem Programm zur Ausführung von Instruktionen, sowie Meta Informationen zusammen. Daten können sich aber in jedem Firmware Image unterscheiden, dagegen wären vorkommende Operationen im Programm ein struktureller Aspekt, der bei allen Firmware Dateien einer Architektur gleich sein dürfte.

3.2 Architekturerkennung durch Maschinencode

Binwalk [14] ist ein Werkzeug, welches primär zur Firmware Analyse eingesetzt wird und verfügt über eine hohe Funktionalität und Erweiterbarkeit. Dabei implementiert Binwalk auch zwei Verfahren zur Architekturerkennung.

Das erste Verfahren sucht nach Signaturen, die den Opcodes einer Architektur entsprechen und listet diese in der Kommandozeile auf. Das Problem bei diesem Ansatz ist, dass auch die Opcodes anderer Architekturen aufgelistet werden können, zudem überlässt das Verfahren die Entscheidung über die Architektur dem Benutzer selbst.

Das zweite Verfahren verwendet den Capstone Disassemblierer [31]. Binwalk beruft sich bei seinen Verfahren also auf die Instruktionen beziehungsweise Operationen im Programm der Dateien.

In einer Projektarbeit entstand im Laufe des Wintersemesters 15/16 das Verfahren *O.b.d.a* (Opcode based detection of CPU architectures) [2], welches bei Eingabe einer beliebigen Datei, zwischen den Architekturen *x86, x86-64, MIPS und ARM* klassifizieren kann. Grundlage bietet ein statistisches Verfahren, bei dem Opcode Tabellen erstellt werden, die für jede Architektur die Häufigkeit der Operationen beinhaltet. Die Häufigkeiten wurden dabei statistisch mithilfe einer Datenmenge von ELF Dateien erstellt. Aus den ELF Dateien wurden die Code Sektionen *(.text section)* extrahiert und anschließend mit dem Capstone Disassemblierer disassembliert. Aus der Assembler Ausgabe wurden anschließend die Operationen gezählt und statistisch zusammengefasst. Für die Klassifizierung wird die zu untersuchende Datei anschließend für alle untersuchten Architekturen komplett disassembliert und die darin erzeugten Häufigkeiten von Operationen anschließend mit den Opcode Tabellen verglichen. Das Verfahren konnte bei der Klassifizierung von unterschiedlichen Befehlssätzen gute Ergebnisse erzielen, außer bei erweiterten Befehlssätzen wie x86 auf x86-64. Die Ergebnisse sind ein weiteres Argument für die Analyse von Operationen im Programm für eine Architekturerkennung. Tabelle 1 [3] zeigt beispielhaft, wie eine Opcode Tabelle für MIPS aussieht. Dabei repräsentiert die erste Spalte die Operationen und jede weitere das Vorkommen in der untersuchten Datei.

3.3 Hybride Verfahren

Ein weiteres für die Architekturerkennung interessantes Verfahren, ist der *Codescanner* [32]. Dieser wird primär für die Erkennung von x86 Code in beliebigen Dateien eingesetzt. Dabei können die verwendeten Ansätze theoretisch auch für die Architekturerkennung angewendet werden. Das ganze Verfahren lässt sich in 3 Schritte eingliedern. Im ersten Schritt wird eine Datenfilterung durchgeführt. Das Ziel dabei ist, redundante Daten und Zeichenketten in der Datei zu erkennen und zu markieren, anschließend alle Regionen die nicht markiert wurden, als potenziellen Code zu klassifizieren. Dabei wird die Shannon Entropie angewendet, da Code und Kryptographische Signaturen, in der Regel eine hohe Entropie aufweisen, wogegen Zeichenketten und Wiederholungen zum

Tabelle 1: Opcode Tabelle für MIPS

Opcodes	1	2	3	...	13	14	15	Average
Total Opcodes	5016	9836	19276	...	59584	7224	7020	551616
lw	0.31081	0.25315	0.24248	...	0.22269	0.29748	0.36325	0.30287
addiu	0.14912	0.19917	0.13727	...	0.15553	0.15185	0.1104	0.15026
move	0.14254	0.10492	0.10116	...	0.09628	0.13787	0.19872	0.13706
sw	0.10008	0.11194	0.10319	...	0.08987	0.09496	0.09345	0.09164
jalr	0.0616	0.06944	0.03554	...	0.05366	0.08306	0.12222	0.0732
lui	0.02273	0.06893	0.04586	...	0.0607	0.04914	0.05655	0.03673
beqz	0.03768	0.0246	0.03403	...	0.02643	0.02907	0.01168	0.02922
bnez	0.0303	0.02125	0.02936	...	0.02405	0.01813	0.00413	0.01924
j	0.0301	0.02938	0.03393	...	0.00057	0.02893	0.00712	0.01644
jr	0.01475	0.0119	0.01095	...	0.01041	0.0108	0.0114	0.01297
addu	0.00399	0.00397	0.03683	...	0.01114	0.00748	0.00114	0.01283
b	0.00478	0.0001	0.00228	...	0.03165	0.00014	0.00014	0.01231
beq	0.02253	0.00813	0.01318	...	0.00864	0.01398	0.00157	0.01127
lbu	0.01196	0.01057	0.01261	...	0.00616	0.0054	0.00014	0.01051
bne	0.01176	0.01169	0.01043	...	0.01165	0.00817	0.00328	0.00904
nop	0	0.00031	0.0001	...	0.10191	0.00014	0	0.00719
sll	0.00179	0.00366	0.01691	...	0.00596	0.00554	0.00028	0.00678
andi	0.00857	0.01118	0.01105	...	0.00648	0.00664	0	0.00657

Beispiel eine niedrige Entropie aufweisen oder nur einen Byte Bereich von [0-127] für die ASCII Kodierung abdecken.

Der zweite Schritt umfasst die konkrete Erkennung von Code. Dabei wird ein heuristisches Verfahren angewendet, dass als Grundlage die Idee, dass ein lauffähiges Programm immer bestimmte Operationen in unterschiedlichen Häufigkeiten benutzen muss, umsetzt. So verwenden lauffähige Programme zum Beispiel eine hohe Anzahl von Speicher oder Lade Operationen, gleichzeitig kann dafür die Häufigkeit von Stack Befehlen geringer ausfallen oder umgekehrt. Wichtige Opcodes werden also den *semantischen* Gruppen Lade/Speicher Befehle, Stack Operationen, Bedingungen, Funktionsaufrufe zugeordnet. Anschließend werden die Häufigkeiten der Opcodes, die den semantischen Gruppen angehören in Eingabeblöcken gemessen und so anhand des Vorkommens bestimmt, ob der Block eindeutig als Code oder als potenzieller Code einzuordnen ist. Im dritten und letzten Schritt sollen die potenziellen Code Blöcke auf x86 spezifische Muster untersucht werden. Als Beispiel kann in dem potenziellen Code ein typischer *Stack Setup* untersucht werden, was den Block dann noch zusätzlich sicher als Code klassifizieren würde. Mit diesem Verfahren konnten in der Evaluation 97 Prozent der Regionen richtig erkannt werden, was auf eine hohe Zuverlässigkeit des Verfahrens hinweist.

Ein weiteres Verfahren zur Architekturerkennung wurde von John Clemens untersucht, dabei verwendet er mehrere Verfahren mit maschinellem Lernen [11]. Die Lernverfah-

ren werden dabei analog zum Codescanner auf Byte Histogramme angewendet, welche durch eine Datenmenge erzeugt wurden. Mit diesem Verfahren konnte auch eine hohe Genauigkeit bei der Erkennung von CPU Architekturen erzielt werden.

Zusammenfassend zeigen die Ergebnisse, dass es möglich ist bei der Betrachtung von Opcodes beziehungsweise Byte Mustern, Regionen von Code zu identifizieren beziehungsweise man aufgrund der Verteilung von bestimmten Byte Mustern, Maschinencode auf eine Architektur zurückführen kann.

4 Konzept

In diesem Kapitel soll das Konzept des Verfahrens in seinen Bestandteilen erläutert werden. Die Idee besteht darin, CPU Architekturen auf ermittelte Code Regionen in der Datei zurückzuführen. Das Verfahren besteht dabei aus zwei Kernfunktionen. Die erste Kernfunktion strebt das Eliminieren von Daten beziehungsweise Bereiche in der Datei an, die nicht Code enthalten. Die zweite Kernfunktion versucht auf Grundlage von architekturspezifischen Code in Byte Blöcken die Datei auf eine Architektur zurückzuführen. Als Eingabe wird dabei der im ersten Schritt extrahierte Rest gegeben. Abbildung 2 zeigt schematisch den Ablauf des Verfahrens.

4.1 Erkennung von Daten

Bei der Datenfilterung besteht die Aufgabe darin, die eingelesenen Blöcke zwischen potenziellem Code und Daten zu unterscheiden. Als Daten wird in diesem Kontext alles bezeichnet, was nicht Code entspricht. Unabhängig von der Architektur sind Daten in erster Linie zum Beispiel:

- Zeichenketten
- Regionen mit einer niedrigen Entropie zum Beispiel Wiederholungen
- Kryptographische Signaturen

Code und kryptographische Signaturen charakterisieren sich durch einen hohen Informationsgehalt. So wird analog zum Codescanner auf diese Bereiche die Shannon Entropie angewendet um Datenregionen zu eliminieren, die eine niedrige Entropie aufweisen. Zeichenketten können eine höhere Entropie aufweisen, aber aufgrund der ASCII Reichweite von [0-127] (dezimal), reicht es aus Datenblöcke die grundsätzlich diese Byte Reichweite abdecken, als Zeichenkette zu klassifizieren und folglich zu eliminieren.

Damit ein Block als potenzieller Code klassifiziert werden kann, muss die Entropie einen bestimmten Schwellwert erreichen. Dieser Schwellwert muss vorerst ermittelt werden. Dazu wird eine Datenmenge von allen abgedeckten Architekturen erstellt, die größtenteils aus reinem Code bestehen. Dazu wird jede eingelesene Datei vorher modifiziert und die Code Sektion extrahiert. Anschließend liest das Programm die Datenmenge ein und aus den erzeugten Entropie Ergebnissen wird zuletzt ein Mittelwert gebildet. Dieser Mittelwert stellt den Schwellwert dar, der mindestens zu erreichen ist, damit ein Datenblock als potenzieller Code Block klassifiziert werden kann.

4.2 Erkennung von Code

Nachdem die Filterung der Daten abgeschlossen ist, besteht der nächste Schritt darin, potenziellen Code im extrahierten Rest zu identifizieren. Dazu werden Byte Blöcke variabler Größe eingelesen und darauf eine Opcode Häufigkeitsanalyse durchgeführt. Ähnlich

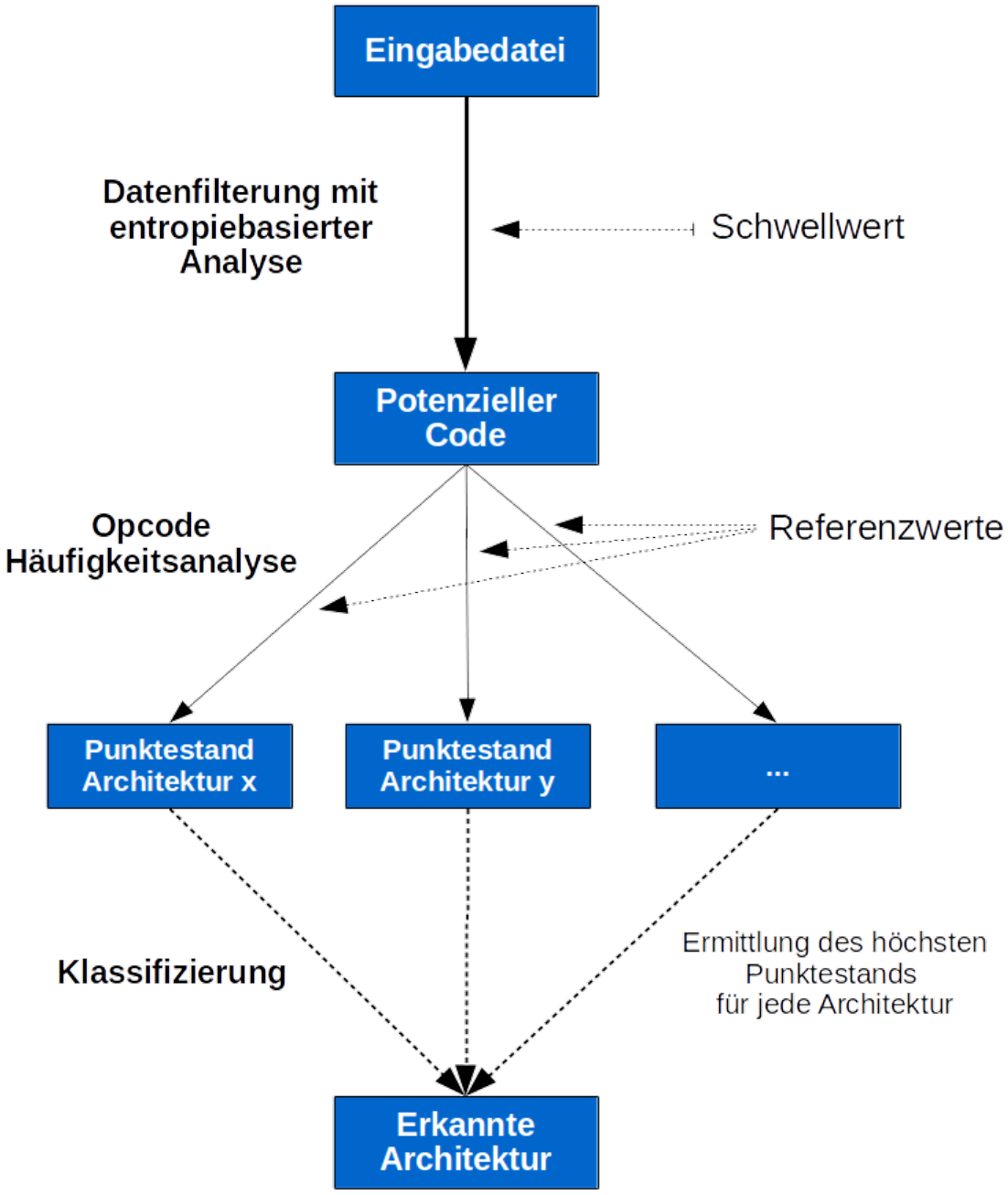

Abbildung 2: Die Abbildung stellt das Konzept zur Architekturerkennung schematisch dar. Es wird schrittweise dargestellt, wie eine Eingabedatei die einzelnen Schritte durchläuft bis es im Anschluss anhand des Punktestands einer Architektur zugeordnet wird.

zum Codescanner [32] besteht die Idee darin, ausführbare Regionen zu identifizieren, indem nach Operationen gesucht wird, welche für ein korrektes semantisches Ausführungsverhalten nicht fehlen dürfen. Diese Operationen lassen sich dabei in 4 Kategorien bei 2 Gruppen einteilen.

- Lade/Speicher Operationen
- Stack Operationen

sowie:

- Bedingungsgebundene Operationen
- Funktionsaufrufe

Die Gruppeneinteilung stellt dabei die Ausführungstypen in ein kausales Verhältnis. So können Operationen zu beiden Kategorien einer Gruppe gehören, allerdings müssen diese Operationen in einem Gleichgewicht stehen. So kann ein Programm zum Beispiel sehr wenige Lade oder Speicheroperationen ausführen, allerdings wäre für ein korrektes Ausführungsverhalten zu erwarten, dass zum Ausgleich mehr Stack Operationen zum Einsatz kommen müssen [32] . Ähnlich verhält es sich mit Bedingungsgebundenen Operationen, wie Sprüngen und Verzweigungen. So kann ein Programm komplett auf Funktionsaufrufe verzichten, wenn dafür der Bedarf an bestimmten Funktionalitäten mit Verzweigungen ausgeglichen wird. Auf Grundlage dieser *Heuristik*, werden die Opcodes der jeweiligen Operationen in dem untersuchten Byte Block gezählt und anschließend ausgewertet, welche Architektur für den analysierten Byte Block in Frage kommt. Damit abgeschätzt werden kann wie das Verhältnis der Ausführungsgruppen bei jeder Architektur zu einander ist, wird eine Datenmenge von Code eingelesen und dort ähnlich zur entropiebasierten Analyse mehrere Schwellwerte erzeugt, welche im weiteren Verlauf der Arbeit als *Referenzwerte* bezeichnet werden. Dabei wird jede Architektur mit einem Punktestand initialisiert. Sollte der untersuchte Byte Block für eine Architektur positiv ausfallen, so wird jener Punktestand einer Architektur inkrementiert. Zuletzt wird anhand des Punktestands entschieden, welche Architektur für die Datei am ehesten in Frage kommen würde.

Listing 3: Ablauf der Architekturerkennung

```
1   Ablauf der Architekturerkennung
2
3   Read File();
4
5   begin for
6       for Block in File
7       do: detect_data()
8   end for
9
10  begin for
11      for Block in Rest
12      do: detect_code()
13      do: classify_block()
14  end for
15
16  classify_file()
```

5 Umsetzung

In diesem Kapitel soll detailliert die Umsetzung des untersuchten Verfahrens dokumentiert werden. Dabei wird systematisch auf die Entwicklung, sowie die Herausforderungen eingegangen, die bei der Implementierung des Verfahrens entstanden sind. Die Idee zur Umsetzung und die eingesetzten Algorithmen werden der Lesbarkeit halber in Pseudocode formuliert.

Das Verfahren umfasst eine Sammlung von Skripten und Tools, die in *Python 3* sowie mittels *Shell Programmierung* umgesetzt wurden. Das Verfahren, dass konkret zur Architekturerkennung eingesetzt wird, wird im folgenden als *ARCH* (Architecture Recognition by CPU Characteristic Heuristics) bezeichnet.

Das ganze Verfahren lässt sich wie in Kapitel 4 erläutert, in die Filterung von Daten, sowie die Erkennung von Code trennen. Die Erkennung von Code bildet dabei mit der Klassifizierung einen fließenden Übergang. Diese beiden Kernelemente werden im weiteren Verlauf unabhängig voneinander betrachtet und erläutert. Das Programm, dass die Architekturen anschließend klassifiziert, verwendet Referenzwerte, die vorerst durch andere Programme evaluiert werden müssen. Dabei handelt es sich um optimierte Werte, die für die Datenerkennung und die Codeerkennung zwingend notwendig sind. Dieser Thematik ist aber ein spezieller Abschnitt in der Evaluation gewidmet. Listing 3 beschreibt abstrakt die Schritte die das Verfahren durchläuft.

Dabei wird eine Datei eingelesen und für eine feste Blockgröße durch die Datei zwei mal iteriert. In der ersten Iteration wird nach Daten gesucht, erkannte Datenblöcke ausgeschlossen und der Rest zusammengefasst. In der zweiten Iteration wird im Rest nach Code Blöcken gesucht um diese im Anschluss einer Architektur zuzuordnen. Der

letzte Schritt umfasst die Erkennung der Architektur auf Grundlage der erkannten Code Blöcke. In den folgenden Abschnitten werden die Funktionen im Detail betrachtet.

5.1 Datenerkennung

In diesem Abschnitt soll dokumentiert werden, wie das in Listing 3 beschriebene Modul *detect_data()* funktioniert. Bei der Erkennung von Daten, besteht das Ziel darin Blöcke zu identifizieren, die möglichst keinen Code enthalten. Wie in Kapitel 2.2.1 bereits erläutert wurde, kann diesem Problem entgegen gekommen werden indem man die Shannon Entropie auf Byte Blöcke fester Größe anwendet und anhand eines Schwellwerts entscheidet, ob der Block als potenzieller Datenanteil klassifiziert werden kann oder nicht. Dazu wurde ein Modul entwickelt, welches iterativ Byte Blöcke einliest und eine modifizierte Version der Shannon Entropie durchführt. Bei dieser Version wird nicht die ganze Reichweite eines Byte im Intervall von [0-256] berücksichtigt. Stattdessen werden Bytes mit einer Reichweite von [0-127] gar nicht erst in die Wahrscheinlichkeiten aufgenommen, aus denen sich der eigentliche Entropiewert zusammensetzt. So können neben Byte Wiederholungen auch effektiv Zeichenketten gefiltert werden, ohne dass andere Bereiche davon maßgeblich beeinträchtigt werden. Anhand eines Werts, welcher an das Programm als Parameter übergeben wird, wird mittels Fallunterscheidung entschieden ob der Block als Code ausgeschlossen werden kann oder nicht.

In einer Datei können auch Bereiche vorkommen die weder die ASCII Reichweite abdecken, sowie gleiche Byte Sequenzen aufweisen, die aber gleichzeitig nicht Code charakterisieren. Damit ein optimaler Wert gefunden werden kann, der alle Charakteristika von Code repräsentiert, wird ein Schwellwert für die Entropie angestrebt, der unterhalb der Entropie von Code liegt. Das Finden dieses Werts wird in Kapitel 6.2.1 thematisiert.

5.2 Codeerkennung und Klassifizierung

In diesem Abschnitt soll auf die Funktionen *detect_code()*, *classify_block()*, sowie *classify_file()* in Listing 3 eingegangen werden. Bei der Erkennung von Code, beschrieben durch *detect_code()* und *classify_block()*, besteht die Aufgabe darin aus den gefilterten Blöcken, potenzielle Code Blöcke zu finden und diese einer CPU Architektur zuzuordnen.

5.2.1 classify_file()

Zur Klassifizierung, beschrieben durch *classify_file()* wurde ein **Punktesystem** implementiert, welches anhand von richtig klassifizierten Blöcken über die CPU Architektur der Datei entscheidet. Genau bedeutet dass, dass jeder eingelesene Block konkurrierend nur einer Architektur zugeordnet werden kann. Listing 4 zeigt den Verlauf der Klassifizierung ohne auf die Kernkomponente, die zum Erkennen von Code verwendet wird genauer einzugehen.

Allerdings hängt die Aussagekraft der Klassifizierung von der Komponente ab, die zur Erkennung von Code, also *detect_code()* verwendet wird. Im folgenden wird dass Verfah-

Listing 4: classify_file()

```
Klassifizierung von Architekturen

    begin for
    for all architectures
        do: architecture_score = initialize_score()
    end for

    begin for
    for all block in filtered_data
        do: detect_code()
        do: estimate_arch_for_code()
        do: update_score(arch_score)
    end for

    architecture = max(architecture_score)
```

ren genauer beschrieben und Umstände erläutert, die die korrespondierende Heuristik beeinflusst.

5.2.2 detect_code()

Das Modul zur Erkennung von Code umfasst wie in Kapitel 4.2 erläutert, einen heuristischen Ansatz mit der Idee, funktionaler Code müsse semantisch bestimmte Kriterien erfüllen um lauffähig zu sein. Diese Idee dürfte für ziemlich jede Architektur gelten, da jeder Computer gleiche Bausteine zur Berechnung von Befehlen verwendet [27].

Zu diesen Bausteinen gehören grob das Laden und Speichern von Werten in Register oder aus dem Speicher. Arithmetische Operationen auf den Werten, Bedingungen überprüfen um Schleifen zu realisieren oder der Aufruf von Funktionen. Der Code wird nun heuristisch in vier semantische Gruppen aufgeteilt, die alle oben genannten Kriterien erfüllen. Diesen semantischen Gruppen werden Opcodes zugeordnet, die als umkodierte Dezimalzahlen die Operationen repräsentieren. Das Programm liest eine *Konfigurationsdatei* ein, die alle Opcodes in den semantischen Gruppen enthält. Dabei wird jeder Opcode als Dezimalzahl dargestellt und verarbeitet. Mit dieser Konfigurationsdatei kann im folgenden, die Häufigkeit bestimmter Operationen in den verbleibenden Blöcken festgestellt werden. Dabei werden die Häufigkeiten in den Blöcken gezählt und mit einem Referenzwert abgeglichen. Der Referenzwert stellt die semantischen Gruppen in ein Verhältnis und muss vorerst ermittelt werden, was in Kapitel 6.2 genauer erläutert wird. In Listing 5 wird skizziert, wie im eingelesenen Byte Block, Code erkannt und einer Architektur zugeordnet wird.

Dabei bezeichnet *average_values* die Werte der durchschnittlichen Häufigkeiten, die vorher evaluiert wurden. *group_with_opcode_values* beschreibt die Werte der Konfigurationsdatei, die für jede Architektur die Opcodes in den semantischen Gruppen beinhaltet.

Listing 5: detect_code()

```
1   Erkennung von architekturspezifischem Code
2
3   read (group_with_opcode_values)
4   read(average_values)
5   read (block)
6
7   begin for
8   for all architectures
9       do: percental_arch_value =
            calculate_percental_opcode_distribution_of_block(
            group_with_opcode_values)
10
11      begin for
12      for all value in average_values
13          do: architecture_deviation = | value - percental_arch_value |
14      end for
15
16  end for
17
18  arch = min(architecture_deviation)
```

architecture_deviation stellt die Abweichung dar, die aus der Berechnung entsteht, die man aus den ermittelten durchschnittlichen Werten im Block, so wie den *average_values* herauskriegt. Dieses Verfahren bietet also keine Option eine Architektur als unsicher zu klassifizieren, sondern ermittelt die CPU Architektur per konkurrierenden Punktestand. Da die Robustheit des Verfahrens neben den ermittelten Mittelwerten zusätzlich von den Opcodes abhängt, die in den semantischen Gruppen zugeordnet werden, wird im folgenden Abschnitt auf diese Problematik genauer eingegangen.

5.3 Auswahl der Opcodes

Die korrekte Auswahl der Opcodes kann für die abgedeckten Architekturen individuell ausfallen. Wie in Abschnitt 2.3 erläutert wurde, können Architekturen wie x86 zum Beispiel, komplexe Befehlssätze implementieren und dementsprechend sehr viele Operationen für äquivalente Aufgaben benutzen. Architekturen wie MIPS dagegen, haben zum Beispiel nur eine kleine Anzahl an Operationen, weshalb dementsprechend die Anzahl an Opcodes in den semantischen Gruppen geringer ausfallen würde. Zusätzlich besitzen MIPS, ARM und PowerPC keine konkreten Stack Befehle, wie der x86 Befehlssatz, der *pop* und *push* Operationen direkt ausführen kann. Stack Befehle werden bei diesen Architekturen über die herkömmlichen Lade und Speicher Befehle realisiert. Aufgrund der hohen Anzahl an Operationen in den Befehlssätzen, ist es wichtig herauszufinden, welche Opcodes überhaupt in Frage kommen, da eine zu große Auswahl an Opcodes zu äquivalenten Ergebnissen führen könnte, wenn mit steigender Anzahl an Architekturen, die Gefahr bestünde die Byte Reichweite [0-256] für alle Gruppen abzudecken. Optimal

Listing 6: Ausschnitt einer analysierten Instruktion mithilfe des Opcode Carver

```
1
2       $ less beq
3
4   beq 16 67 0 71
5   beq 18 114 255 244
6   beq 16 67 255 245
7   beq 17 7 0 40
8   beq 17 2 255 195
9   beq 19 213 0 191
10  beq 16 67 0 104
11  beq 16 67 0 126
12  beq 16 94 0 106
13  beq 18 176 0 84
14  beq 16 67 0 33
15  beq 16 67 255 228
16  beq 16 67 0 57
17  ...
```

wäre also eine Konfiguration von Opcodes, die in den Gruppen sowie Gruppen übergreifend geschnitten eine leere Menge ergibt.

Dieses Problem gehörte zu der größten Herausforderung bei der Implementierung des Verfahrens. Es wurde deshalb der *Opcode Carver* entwickelt, der die Generierung und Analyse von Opcodes bei Eingabe von Maschinencode übernimmt. Dabei implementiert der *Opcode Carver* unter anderem den Capstone Disassemblierer [31], um Maschinencode zu disassemblieren. Desweiteren wurden elftools [17] verwendet um Meta Informationen aus den ELF Dateien zu generieren. Da die Opcodes aus dem Maschinencode gewonnen werden müssen, wurde eine Menge von ELF Dateien eingelesen und der enthaltene Maschinencode extrahiert.

Angefangen bei x86 hat man die Möglichkeit mit dem Capstone Disassemblierer die Opcodes direkt auszulesen, was durch den Opcode Carver problemlos erledigt werden kann, vorausgesetzt man ist in Kenntnis über die verfügbaren Operationen im x86 Befehlssatz. Leider ist der Capstone Disassemblierer, dabei nur auf x86 beschränkt und verfügt nicht über die Möglichkeit, auch die Opcodes der anderen Architekturen auszulesen.

Bei MIPS, ARM und PowerPC müssen die Opcodes anders ermittelt werden. Selbst mit Hilfe von IDA [20] gab es keine Möglichkeit die tatsächlichen Opcodes aus dem Maschinencode dieser Architekturen auszulesen. Dokumentationen waren diesbezüglich nicht hilfreich und konnten höchstens nur einen Überblick über verfügbare Operationen verschaffen. Aus dieser Tatsache heraus fiel als letzter Ausweg die Entscheidung auf einen statistischen Ansatz. Der Kern bei diesem Ansatz besteht darin, für jede relevante Operation in den Befehlssätzen herauszufinden, mit welchen Byte Kodierungen diese assoziiert sind. Dazu müssen die Byte Sequenzen der Instruktionen, die die Operationen repräsentieren untersucht werden.

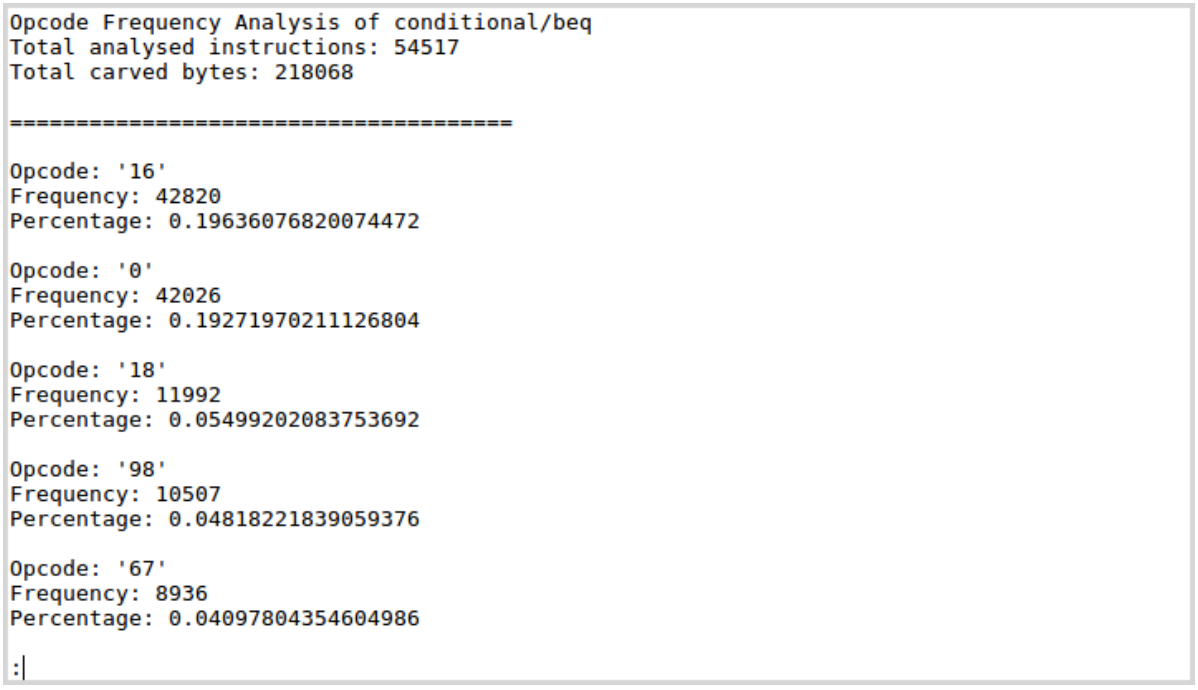

Abbildung 3: Diese Abbildung zeigt die Ausgabe des Inhalts aus Listing 6 nach Anwendung der Frequenzanalyse. Dabei ist zu sehen, dass die Byte Kodierung in Dezimalzahl '16', 19% Prozent aller Bytes die mit dieser Operation in Verbindung stehen repräsentiert und deshalb für diese Operation am ehesten als Opcode Kandidat in Frage kommen würde.

Als nächstes wurde für jede erzeugte Ausgabedatei ermittelt, wie häufig die verschiedenen Byte Kodierungen in den jeweiligen Sequenzen vorkommen. Dazu muss für jeden Eintrag in der Liste iteriert werden und darin die Häufigkeit jeder Byte Kodierung gezählt werden. Anschließend wird in der Ausgabe das durchschnittliche Vorkommen dargestellt. In Abbildung 3 ist dargestellt, welche Frequenzen aus den Byte Sequenzen in dem Beispiel aus Listing 6 entstehen.

Mit diesem Ansatz ist es möglich, Byte Werte in die Konfiguration aufzunehmen, die eigentlich keine Opcodes sind, allerdings können so auch Werte ermittelt werden, die mit der Instruktion die den Mnemomic repräsentiert definitiv in Verbindung stehen müssen, was zusätzlich eine Charakterisierung der Architektur mit sich bringt.

Als letztes soll erläutert werden, wie die Auswahl von geeigneten Opcode Kandidaten für die Konfiguration erfolgt ist. Wie in Kapitel 2.3 gezeigt wurde, können die Befehlssätze unterschiedlich komplex sein und unter Umständen könnte so die Byte Reichweite komplett abgedeckt werden, bei Berücksichtigung aller Operationen. Es ist auch möglich, dass unterschiedliche Operationen äquivalente Aufgaben erfüllen, sofern unterschiedlich konfigurierte Compiler zum Einsatz kommen. Deshalb muss darauf geachtet werden nur Opcode Kandidaten in die Konfiguration aufzunehmen, die tatsächlich im Code auch vorkommen. Dazu wurden zur Hilfe die Opcode Tabellen mit einbezogen, die sich in

```
[x86]
moving = [139, 161, 137, 199, 163, 185, 187, 186, 190, 184, 198, 178, 136, 189, 191, 162, 180, 138, 176, 142, 179, 177,
 160, 15]
calls = [232, 255, 141, 195, 194, 201]
stack_access = [87, 86, 83, 80, 84, 82, 104, 81, 85, 106, 91, 94, 95, 93, 88, 89, 90]
jumping = [131, 128, 129, 126, 57, 60, 61, 59, 56, 58, 15, 116, 255, 235, 233, 119, 118, 120, 117]

[mips]
moving = [143, 153, 188, 140, 16, 142, 175, 174, 172, 16, 162, 176, 160, 163, 2, 144, 146, 147, 148, 66]
calls = [12, 16, 21, 10, 248, 31, 9, 3]
stack_access = []
jumping = [16, 255, 254, 252, 64, 128, 18, 20, 4, 17]

[powerpc]
moving = [128, 129, 131, 8, 12, 144, 147, 145, 120, 124, 127, 16, 60, 61, 148, 33, 3, 166]
calls = [72, 1, 75, 128, 78, 255]
stack_access = []
jumping = [255, 254, 75, 47, 131, 137, 64, 190, 128]

[arm]
moving = [229, 48, 159, 157, 16, 225, 232, 3, 141, 48, 4, 225, 160, 227]
calls = [235, 255, 254, 253]
stack_access = []
jumping = [26, 10, 234, 227, 225, 83, 80]
(END)
```

Abbildung 4: Diese Abbildung zeigt die resultierende Konfigurationsdatei, die als Eingabe zur Klassifizierung von CPU Architekturen benötigt wird.

[2] in diesem Kontext gut dazu nutzen lassen, um mögliche Kandidaten herauszufinden. Allerdings konnte aufgrund der zu großen Differenzen in den Befehlssätzen der Architekturen, keine gleichmäßige Verteilung der Opcodes in den Konfigurationen ermöglicht werden. Es wurde nämlich stets darauf geachtet, alle semantischen Gruppen abzudecken. Abbildung 4 zeigt die resultierende Konfigurationsdatei, die nach Auswahl der Opcodes entstanden ist.

Tabelle 2: Gesamter Datenpool für die Evaluation

Architektur	Dateien	Größe in GB
AMD64	5121	2.1
Armel	4778	1.3
Armhf	4793	1.5
MIPS	4906	2
i386	5039	1.9
PowerPC	5083	1.7
Insgesamt	29720	10.5

Tabelle 3: Trainingsmenge und Evaluationsmenge

Arch.	Anzahl Daten insg.	Trainingsdaten	Evaluationsdaten	Größe insg.
x86	5039	1680	3359	1,9 GB
MIPS	4861	1625	3236	2,0 GB
ARM	4778	1592	3186	1,3 GB
PowerPC	5083	1695	3388	1,7 GB
Insgesamt	19761	6592	13169	6,9 GB

6 Evaluation

In diesem Kapitel soll der entstandene Ansatz mit existierenden Verfahren unter anderem in ihrer Erkennungsrate, so wie den ökonomischen Eigenschaften verglichen werden. Zusätzlich wird auf die Evaluation des Schwellwerts eingegangen, welcher durch die Shannon Entropie festlegt, ob ein Byte Block Daten enthält oder nicht. Desweiteren wird auf die Erzeugung der Referenzwerte eingegangen, die bei der Codeerkennung von großer Bedeutung sind. Zu den Verfahren die im Anschluss miteinander verglichen werden gehört das untersuchte Verfahren, Binwalk [14] mit dem *-Y* und dem *-A* Parameter. Zunächst wird erläutert, welche Daten für die Evaluation verwendet wurden und wie diese für die Evaluationsdurchläufe angepasst wurden.

6.1 Vorbereitung der Evaluationsdaten

6.2 Evaluation der Referenzwerte

In diesem Abschnitt soll auf die Erzeugung der Schwellwerte eingegangen werden, die zur Klassifizierung benötigt werden. Dabei wird in den einzelnen Abschnitten jeweils zuerst die Anpassung der Evaluationsdaten erläutert und darauffolgend die Ergebnisse die dabei entstanden sind.

Tabelle 4: Optimierter Schwellwert

Architektur	Dateien	Blöcke	Summe Entropie	Durchschn. Entropie	Größe in MB
AMD64	1707	1293851	2941324	2.2733	316
Armel	1592	893611	2317435	2.5933	219
Armhf	1597	777430	1967639	2.5309	190
Mips	1635	1092324	2292876	2.099	267
i386	1678	1048629	2681420	2.557	257
powerpc	1694	1163874	2624931	2.255	285
Insgesamt	9903	6269719	14825625	2.364624	1534

6.2.1 Datenerkennung

Die Datenerkennung braucht wie in Abschnitt 5.1 erläutert, einen Schwellwert um zu entscheiden ob der eingelesene Block weiter untersucht werden soll oder verworfen werden kann. Für die Erzeugung dieses Schwellwerts muss der Maschinencode an sich betrachtet werden. Dieser Prozess kann architekturunabhängig durchgeführt werden, weshalb auch weitere Architekturen miteinbezogen wurden. Die korrespondierenden Evaluationsdaten umfassen die Daten in Tabelle 4. Dabei wurden ein Drittel aller Dateien aus Tabelle 2 randomisiert für alle Architekturen mit dem Linux Kommando *ls |sort -R |tail -<EinDrittel>* kopiert. Anschließend wurden alle Dateien modifiziert, indem jeweils nur die Code Sektionen extrahiert wurden, da der Schwellwert möglichst an den Maschinencode angepasst werden soll. Zum Anpassen wurde wieder der Opcode Carver verwendet, wie in Abschnitt 5.3 bereits erläutert wurde.

Für die Durchführung der Evaluation wurde auch die modifizierte Shannon Entropie verwendet wie in Abschnitt 5.1 erläutert wurde. Es wurden nur 256 Byte große Blöcke untersucht, da die Shannon Entropie mindestens die doppelte Größe der Eingabe benötigt, wie das Eingabealphabet an möglichen Zeichen hat, also abgezogen von der ASCII Reichweite [127-256] [32]. Für jeden Block wurde die Entropie berechnet und summiert, im Anschluss wurde der Durchschnitt berechnet. Die Ergebnisse sind in Tabelle 4 dargestellt. Dabei liegt der durchschnittliche Entropie Wert bei **2.36**. Jeder Block mit einem Wert unterhalb dieses Werts wird als Datenanteil klassifiziert und folglich verworfen.

6.2.2 Codeerkennung

Die Codeerkennung benötigt neben der Opcodes, die explizit ausgewählt werden müssen noch Referenzwerte als Eingabe, die die semantischen Gruppen in ein Verhältnis setzt. Damit die Referenzwerte erzeugt werden können, wurden randomisiert aus dem Datenpool ein Drittel aller x86, MIPS, ARM und PowerPC Daten gefiltert. Die anderen zwei Drittel werden wie unten erläutert, für die Evaluation aller Verfahren verwendet um mit der Gesamtheit aller Daten eine leere Menge zu bilden und nicht Evaluationsdaten zu verwenden, die schon zum Erzeugen der Referenzwerte verwendet wurden. Tabelle 3 zeigt die Aufteilung. Das Drittel, dass zum Erzeugen der Referenzwerte verwendet wird,

Tabelle 5: Referenzwerte für die semantischen Gruppen

Arch./ Sem. Gruppe	Lade/Speicher	Stack	Bedingung	Unterprog. Aufruf
ARM	31.00%	0.00%	6.00%	1.00%
MIPS	23.00%	0.00%	10.00%	11.00%
PowerPC	24.00%	0.00%	12.00%	12.00%
x86	13.00%	4.00%	12.00%	17.00%

Tabelle 6: Evaluation ARCH

Architektur	Anzahl Dateien	Richtig	Falsch	True positive
ARM	3186	2796	390	87,7%
MIPS	3236	2854	382	88,1%
PowerPC	3388	2688	700	79,3%
x86	3359	2992	367	89,1%

wurde zusätzlich noch auf gleiche Weise, wie bei der Datenerkennung modifiziert, indem die reinen Code Regionen extrahiert wurden.

Anschließend wird ein Programm ausgeführt, dass die Konfiguration der ausgewählten Opcodes als Eingabe entgegen nimmt und auf allen Dateien eine Frequenzanalyse für die angegebenen Opcodes durchführt. Dabei wird nach den Mustern, die in der Konfiguration angegeben sind im Programm gesucht und im Anschluss das durchschnittliche Vorkommen berechnet. Tabelle 5 zeigt die Ergebnisse der Häufigkeiten, die dabei entstanden sind.

6.3 Vergleich der Verfahren

6.3.1 ARCH

In diesem Abschnitt soll die Durchführung der Evaluation für das entstandene Verfahren erläutert werden. Dazu wurde mit einem Skript, das entstandene Tool *ARCH* (siehe Kapitel 5) , welches das Verfahren implementiert über die Evaluationsdaten aus Tabelle 3, für jede Architektur iteriert. Als Eingabe wurde die Opcode Konfiguration aus Abbildung 4 verwendet. Als Referenzwerte wurden die aus Tabelle 5 evaluierten Ergebnisse verwendet.

In Tabelle 7, 8, 9 und 10 sind die Erkennungsraten für alle Evaluationsdaten aufgelistet. Dabei repräsentiert jede Tabelle die Evaluationsdaten für eine Architektur. Tabelle 6 repräsentiert das gesamte Ergebnis für die Evaluation von ARCH. Aus den Tabellen kann man entnehmen, dass die Evaluationsdaten für ARM, MIPS und x86 mit einer ähnlichen True positive Rate die Architekturen zwischen 88-89 % richtig klassifizieren konnten. Lediglich PowerPC Dateien konnten nur zu 79 % richtig klassifiziert werden. Insgesamt ergibt das eine True positive Rate von ca. 86 % für das gesamte Verfahren.

Tabelle 7: ARCH Ergebnis für ARM Dateien

Architektur	**Erkannte Architekturen**	**Anteil ∼**
ARM	2796	87,7 %
MIPS	167	5,2 %
PowerPC	8	0,2 %
x86	215	6,7 %

Tabelle 8: ARCH Ergebnis für MIPS Dateien

Architektur	**Erkannte Architekturen**	**Anteil ∼**
ARM	91	2,8 %
MIPS	2854	88,1 %
PowerPC	132	4,0 %
x86	159	4,9 %

Tabelle 9: ARCH Ergebnis für PowerPC Dateien

Architektur	**Erkannte Architekturen**	**Anteil ∼**
ARM	111	3,2 %
MIPS	186	5,4 %
PowerPC	2688	79,3 %
x86	403	11,8 %

Tabelle 10: ARCH Ergebnis für x86 Dateien

Architektur	**Erkannte Architekturen**	**Anteil ∼**
ARM	6	0,1 %
MIPS	10	0,2 %
PowerPC	351	10,4 %
x86	2992	89,09 %

6.3.2 Binwalk

Als nächstes sollen die Erkennungsraten für das Firmware Analyse Tool Binwalk [14] untersucht werden. Dabei wurden zwei Durchläufe durchgeführt, da Binwalk zwei Methoden besitzt um die Architektur einer Datei zu klassifizieren. Die erste Methode kann mit dem Kommando *'binwalk -A Zieldatei'* aufgerufen werden, welche keine konkrete Klassifizierung durchführt und stattdessen nur erkannte Opcode Signaturen auflistet. Abbildung 5 zeigt ein Beispiel, wie die Ausgabe einer Binwalk -A Ausführung aussieht. Die zweite Methode kann mit *'binwalk -Y Zieldatei'* aufgerufen werden und benutzt den Capstone Disassemblierer um die Datei auf die Architektur zurückzuführen. Bei der Ausführung mit dem '-Y' Parameter wird die erkannte Architektur angezeigt.

Angefangen mit dem Ansatz, der mit dem '-A' Parameter aufgerufen wird ist eine Klassifizierung nicht direkt möglich. Allerdings kann dadurch, dass Zeilen aufgelistet werden, in denen unter anderem der Offset, sowie die Architektur, die dem gefundenem Opcode entspricht, abgeschätzt werden wie viele architekturspezifische Opcodes im Verhältnis zu denen anderer Architekturen gefunden werden können. Dazu reicht es in jeder Zeile nach der Zeichenkette zu suchen, die der Architektur entspricht. Das Programm, dass Binwalk evaluiert nimmt als Eingabe die Quelldatei, sowie den Namen der Architektur auf die die Datei überprüft werden soll. Optional kann aber auch der '-Y' Parameter übergeben werden um Binwalk's '-Y' Kommando auszuführen, was bei dem zweiten Evaluationsdurchlauf zum Einsatz kommt. Anschließend iteriert das Programm über alle relevanten Zeilen in der Ausgabe und untersucht die Zeichenketten nach dem Namen der Architektur, der als Parameter übergeben wurde. Sollten mehr als 85 % der Zeilen mit der Architektur übereinstimmen, so wird die Architektur als richtig klassifiziert, andernfalls als *unbestimmt*. Das Ergebnis der Evaluation ist aus Tabelle 11 zu entnehmen. Die Ergebnisse weisen bei einer durchschnittlichen Erkennungsrate von 97,625 % eine höhere True positive Rate als ARCH auf.

Als nächstes soll Binwalk's '-Y' Parameter untersucht werden. Diese Methode nimmt die selben Evaluationsdaten, wie der erste Durchlauf entgegen. Bei der Klassifizierung wird nur eine Zeile ausgegeben, die die erkannte Architektur enthält. Das gleiche Programm liest nun jede Datei ein und entscheidet anhand der relevanten Zeile, welche Architektur klassifiziert wurde. Das Ergebnis aus diesem Durchlauf ist aus Tabelle 12 zu entnehmen. Insgesamt ergibt sich eine True positive Rate von 28,66 % somit schneidet dieses Verfahren schlechter ab, als die beiden anderen Verfahren. Tabelle 13 zeigt die konkrete Verteilung der Klassifizierung an. In Abbildung 6 soll zuletzt noch ein Überblick über die evaluierten Verfahren gegeben werden.

6.3.3 Weitere Verfahren

Auf das Verfahren von John Clemens wird hier nicht ausführlich eingegangen, da die Implementierung in seiner Arbeit nicht genauer erläutert wird und eine Evaluation beziehungsweise ein ausführlicher Vergleich, deshalb nicht möglich ist. Allerdings verzeichnet

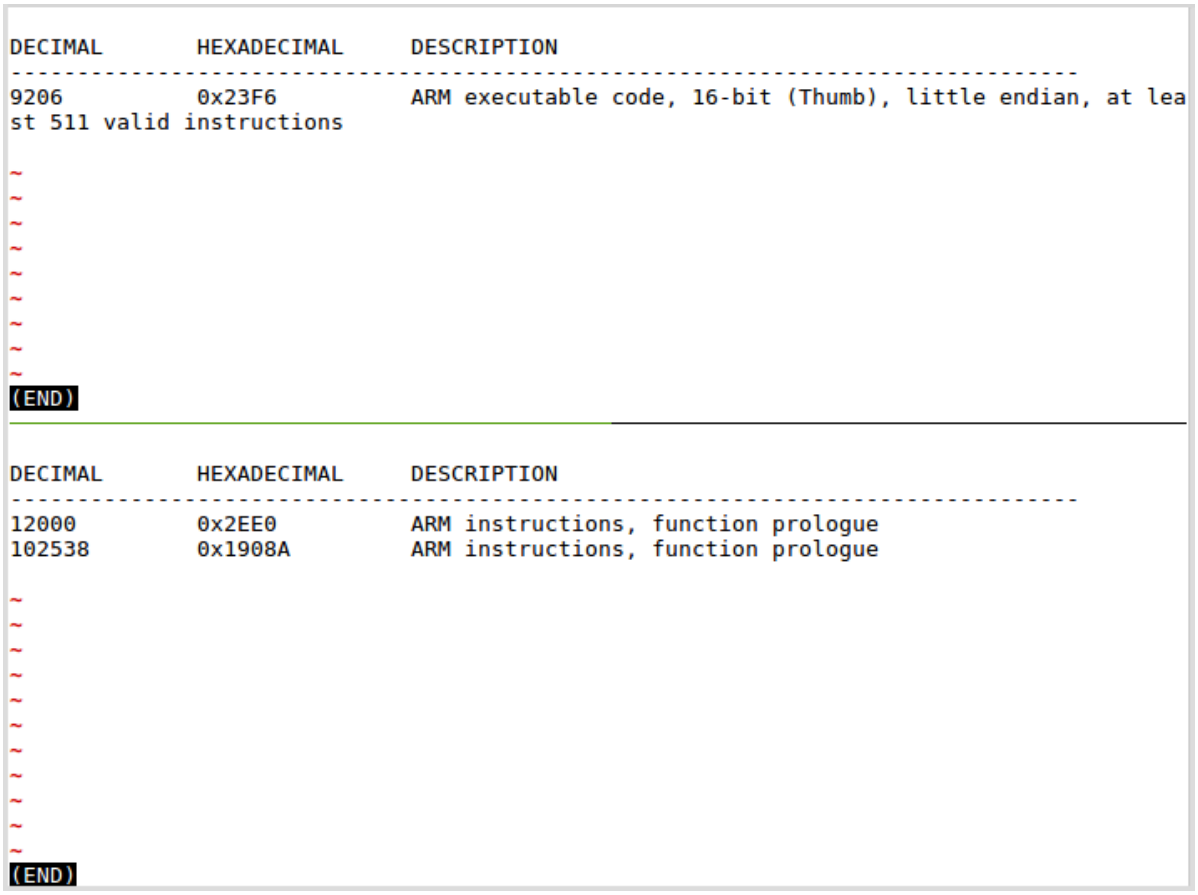

Abbildung 5: Diese Abbildung zeigt in der oberen Hälfte die Ausgabe eines binwalk -Y Kommandos, angewendet auf eine ARM Binärdatei. Die untere Hälfte zeigt die Ausgabe eines binwalk -A Kommandos, angewendet auf die gleiche Datei.

Tabelle 11: Evaluation binwalk -A Parameter (Opcodes auflisten)

Architektur	**Anzahl Dateien**	**Richtig**	**Falsch**	**True positive**
ARM	3186	3183	3	99,9%
MIPS	3236	3224	12	99,6%
PowerPC	3388	3384	4	99,9%
x86	3359	3061	298	91,1%

Tabelle 12: Evaluation binwalk -Y Parameter (Capstone Disassemblierer)

Architektur	**Anzahl Dateien**	**Richtig**	**Falsch**	**True positive**
ARM	3186	3164	22	99,3%
MIPS	3236	611	2625	18,8%
PowerPC	3388	0	3388	0,0%
x86	3359	0	3359	0,0 %

Tabelle 13: binwalk -Y Parameter (Konkrete Verteilung)

Architekturen	**ARM**	**MIPS**	**PowerPC**	**x86**	**Unbestimmt**
ARM Dateien	3164	3	0	0	19
MIPS Dateien	2605	611	0	0	20
PowerPC Dateien	3332	7	0	0	49
x86 Dateien	3298	10	0	0	51

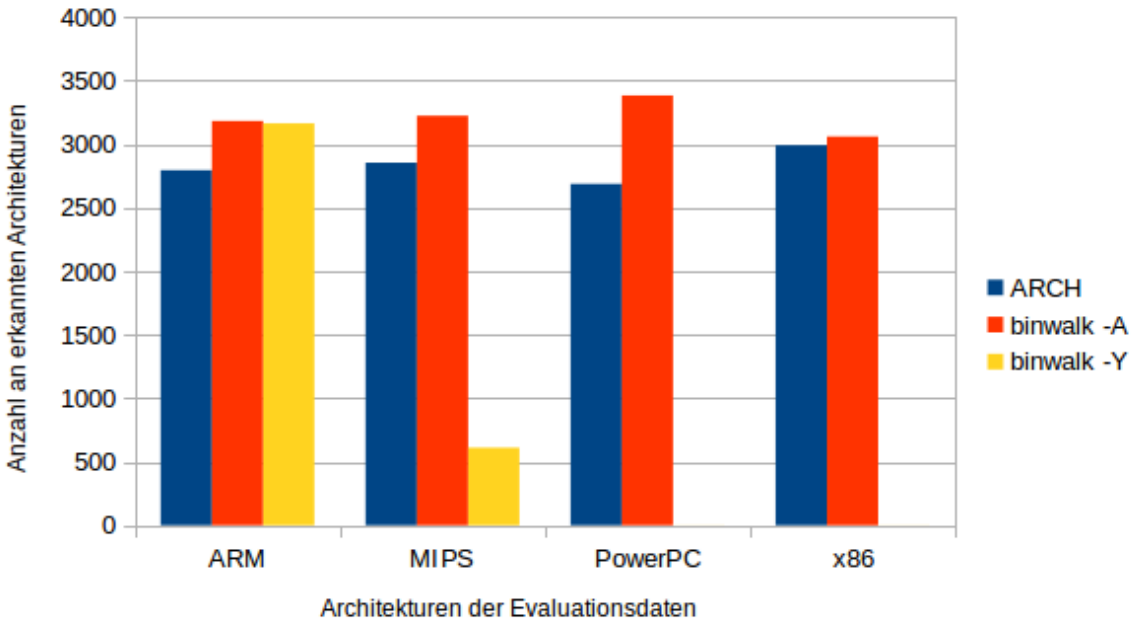

Abbildung 6: In dieser Abbildung sind alle Verfahren noch einmal graphisch dargestellt. Die x-Achse steht für die Architektur der Evaluationsdaten, die y-Achse steht für die Anzahl an richtig erkannten Architekturen in den Evaluationsdaten.

er in seiner Arbeit eine Erkennungsrate von über 90 % bei 20 verschiedenen Architekturen und 16000 Code Stichproben [11].

6.4 Ökonomische Aspekte

In diesem Kapitel soll zuletzt noch auf die ökonomischen Hintergründe eingegangen werden, die bei der Anwendung sowie Vorbereitung der betrachteten Verfahren zu Kosten kommen.

Angefangen bei dem Verfahren, welches im Rahmen dieser Arbeit angefertigt wurde, müssen vorerst einige Schritte durchgeführt werden, bevor Architekturen erkannt werden können. Zunächst muss ein Schwellwert für die Entropie gefunden werden. Dieser müsste einmalig allgemeingültig für alle Architekturen erzeugt werden und könnte dann für jedes Einsatz Szenario eingesetzt werden. Die Auswahl der Opcodes müsste auch durchgeführt werden, was aber den Vorteil mit sich bringen könnte das Verfahren auf unterschiedliche Compiler anzupassen. Zuletzt müssen noch die Referenzwerte für die semantischen Opcode Gruppen und der korrespondierenden Heuristik erzeugt werden. Diese muss man auch einmalig erzeugen, allerdings würden diese Werte bei einer anderen Opcode Konfiguration ihre Gültigkeit verlieren. Die Laufzeit der Erkennung verläuft linear mit steigender Größe des Eingabevektors und der Anzahl an Architekturen, da nur Häufigkeiten gezählt und verglichen werden und über jede Architektur iteriert wird. So dauerte die Evaluation für eine Datenmenge einer Architektur (ca. 1.7 Gigabyte Evaluationsdaten) bei einem Computer mit 8 Gigabyte RAM Arbeitsspeicher und einem 4 Kern Prozessor mit 1,6 GHZ, bis zu 2 Stunden. Die Evaluationsdaten als Eingabeparameter umfassen nur zwei Text Dateien, sowie eine Zahl, die den Schwellwert für die Shannon Entropie repräsentiert. Das Verfahren kann also für eine große Anzahl von CPU Architekturen mit nur wenigen Ressourcen auskommen. Die Vorbereitung kann zeitlichen Aufwand in Anspruch nehmen um die Schwellwerte zu generieren, allerdings kann das Verfahren schnell durchgeführt werden und zusätzlich um beliebige Architekturen erweitert und optimiert werden.

Binwalk sucht mit dem -A Parameter nach architekturspezifischen Opcode Signaturen. Diese müssen vorher erzeugt werden und gegebenenfalls in einer Datenbank abgespeichert werden, da mit steigender Anzahl an Architekturen auch die Anzahl an Signaturen wächst. Die Evaluation mit den selben Voraussetzungen wie in diesem Kapitel bereits erläutert, dauerte mit dem -A Option im Vergleich bis zu 6 Stunden. Die -Y Option dauerte bei der Evaluation im Vergleich bis zu 24 Stunden, dass entspricht das vierfache der -A Option. In dem Paper von John Clemens wird über die Laufzeit seines Verfahrens nicht viel ausgesagt, allerdings benötigt das Verfahren aufgrund des maschinellen Lernens der Code Stichproben auch Vorbereitungsaufwand, dagegen sollte eine Klassifizierung aufgrund der erzeugten Klassifizierer vergleichsweise schnell verlaufen.

7 Auswertung

Aus den Evaluationsergebnissen in Kapitel 6.3, soll in diesem Kapitel über die Güte der Verfahren entschieden werden und was aus den Ergebnissen abgeleitet werden kann.

Angefangen mit dem Verfahren, welches im Rahmen dieser Arbeit entstanden ist, konnten durchschnittlich bis zu 86 % aller Daten richtig klassifiziert werden. Am häufigsten konnten x86 Binärdateien richtig klassifiziert werden und am wenigsten PowerPC Dateien. Betrachtet man Tabelle 10 und 9 so sieht man, dass ARM und MIPS vergleichsweise selten klassifiziert wurden und Fehlklassifizierungen überwiegend zwischen x86 und PowerPC stattfanden. Daraus folgt, dass der x86 Befehlssatz viele Opcodes implementieren muss, die ähnlich zu denen von PowerPC sein müssen. MIPS konnte vergleichsweise gleichmäßig fehlklassifiziert werden. Betrachtet man die ausgewählten Opcodes in Abbildung 4, so sieht man, dass tatsächlich viele einzigartige Opcodes in der MIPS Konfiguration vorkommen, was Ursache für das Ergebnis sein könnte. ARM dagegen konnte kaum Fehlklassifizierungen mit PowerPC verzeichnen. Betrachtet man dabei die Opcodes von ARM und PowerPC, so erkennt man sehr wenige übereinstimmende Opcodes. Die Erkennungsrate ist also maßgeblich von der Auswahl der Opcodes abhängig. Eine bessere Konfiguration könnte vermutlich zu einem noch besseren Ergebnis führen. Die hohe Erkennungsrate bei x86 konnte bei der aktuellen experimentellen Konfiguration vermutlich resultieren, da mehr Opcodes in den Gruppen vorkammen als bei denen anderer semantischen Gruppen.

Die Ergebnisse für Binwalk zeigen, dass das Verfahren, welches mit der -A Option ausgeführt wird, sehr gute Erkennungsraten bei einer 95 % True positive erzielt. Das deutet auf eine sehr gute Implementierung von Opcode Signaturen hin. Das Verfahren, welches mit der -Y Option ausgeführt wird, konnte dagegen nur 28,66 % der Daten richtig klassifizieren. Dabei wurden die Daten überwiegend als ARM klassifiziert, unabhängig davon, ob die Datei tatsächlich den ARM Befehlssatz implementiert oder nicht. Die schlechte Erkennung, sogar bei x86 Dateien könnte auf einen Fehler im Programm hindeuten. Dieses Verfahren stellt also keineswegs eine Alternative zur Architekturerkennung dar. Überraschenderweise bezeichnet der Hersteller die -Y Option als die robustere Variante bei der Architekturerkennung, was spätestens mit dieser Arbeit widerlegt werden kann.

Allgemein lässt sich aus den Ergebnissen der verglichenen Ansätze folgern, dass vor allem der Einsatz von Opcode Analysen eine wichtige Rolle bei der Klassifizierung von CPU Architekturen spielt. Dabei erzielt der signaturbasierte Ansatz von Binwalk bessere Eregbnisse als der heuristische Ansatz von *ARCH*. Die zusätzliche entropiebasierte Analyse kann einen Vorteil bei der Identifizierung von Code Abschnitten verschaffen. Allerdings erweist sich der heuristische Ansatz als flexibler und ressourcenschonender, da er auf verschiedene Compiler angepasst werden kann und die Laufzeit mit der Größe der Datei und der Anzahl an Architekturen steigt. Zusätzlich lässt sich das Verfahren noch weiter optimieren, sofern eine gute Konfiguration für Opcodes gefunden werden kann.

Dagegen braucht der signaturbasierte Ansatz, Zugriff auf die Signaturen um korrekt zu funktionieren und kann auch Signaturen von unterschiedlichen Architekturen erkennen. Hinzu kommt das Binwalk die Entscheidung dem Benutzer selbst überlässt. Dieses Problem lässt sich aber provisorisch lösen, wie in Kapitel 6.3.2 exemplarisch gezeigt wurde.

8 Zusammenfassung und Fazit

Im Laufe dieser Arbeit wurde ein Verfahren entwickelt, welches einer beliebigen Binärdatei eindeutig eine CPU Architektur zuordnet. Dabei wird auf die Verwendung von Metadaten verzichtet, um den Einsatz vor allem im Bereich der Firmware Analyse zu ermöglichen. Dazu wurden verschiedene Punkte untersucht, die eine Klassifizierung ermöglichen sollen. Im Ansatz soll in der Datei nach architekturspezifischen Code gesucht werden um auf dieser Grundlage die Architektur zu erkennen. Dazu wurde die Shannon Entropie aus der Informationstheorie angewendet, um mögliche Regionen in der Datei zu identifizieren, die keinen Code enthalten. Aus dem Rest sollen dann feste Blockgrößen auf architekturspezifischen Code untersucht werden. Dazu wurde die Frequenz von ausgewählten Operationen jeder Architektur analysiert, um aus dessen Häufigkeiten eine Architektur zu erkennen. Das Verfahren liest als Parameter einen Schwellwert für die Shannon Entropie, die Opcodes auf denen die Datei untersucht werden soll, sowie einen einmalig erzeugten Mittelwert für jede Architektur ein, die zur heuristischen Bestimmung der Architektur eingesetzt wird. Aus der Arbeit geht hervor, dass bestimmende Charakteristika von Architekturen vor allem Operationen sind, die sich sowohl in ihrer Instruktionslänge, wie auch Ausführungssemantik unterscheiden können. Allerdings konnte in der Evaluation gezeigt werden, dass durch die Analyse von Opcode Häufigkeiten (ARCH) und Opcode Signaturen (Binwalk) eine Klassifizierung mit bis zu 99 % Erkennugsrate (Binwalk) möglich ist. Binwalk ist bis jetzt nur auf eine bestimmte Anzahl an Architekturen beschränkt, ARCH lässt sich auf beliebig viele Prozessorarchitekturen erweitern.

9 Ausblick

In dieser Arbeit wurde gezeigt, dass es möglich ist die Architektur einer Datei, alleine auf Grundlage des ausführbaren Codes mit einer 86% Erkennusrate ohne Metadaten zu erkennen. Dafür ist wichtig, möglichst viele Regionen in der Datei ausfindig zu machen, die keinen Code enthalten. Dazu könnte man die Datei zusätzlich auf *Signaruren* untersuchen, die im Datenteil von Firmware Dateien vorkommen. Desweiteren wäre die nächste Herausforderung, die Auswahl der Opcodes zu optimieren. Zusätzlich könnte man noch architekturspezifische Signaturen erzeugen und die Datei darauf untersuchen und das in die zusätzliche Bewertung des Punktestands mit einbeziehen. Konkret könnte man nach bestimmten Instruktionen suchen die wie bei bestimmten Architekturen immer wieder vorkommen und dazu mehr als nur die reinen Opcodes in die Konfiguration mit einbeziehen. Ein gutes Verfahren zur Ermittlung solcher Instruktionen stellt bereits

der in Kapitel 5.3 vorgestellte *Opcode Carver* dar.

Literaturverzeichnis

[1] IEEE 1275-1994 - IEEE Standard for Boot (Initialization Configuration). Firmware: Core Requirements and Practices, 1994.

[2] Marcel Aniol. O.b.d.a, Opcode based detection of CPU architectures. *Projektgruppe IT-Sicherheit*, 2015.

[3] Marcel Aniol. Opcode Tabelle für MIPS. *Projektgruppe IT-Sicherheit*, 2015.

[4] Prof. Dr. Joachim K. Anlauf. Technische Informatik WS 12/13, 2. Vorlesung . Vorlesung, 2012.

[5] Arduino. Arduino. `https://www.arduino.cc/`.

[6] ARM. ARM Processor architecture. `http://www.arm.com/products/processors/instruction-set-architectures/index.php`.

[7] ARM. ARM Processors. `http://www.arm.com/products/processors`.

[8] Bitkom. `https://www.bitkom.org/`.

[9] Bitkom. Eingebettete Systeme – Anwendungsbeispiele, Zahlen und Trends, 2010.

[10] Ian Black. NSA spying scandal: what we have learned. `https://www.theguardian.com/world/2013/jun/10/nsa-spying-scandal-what-we-have-learned`, 2013.

[11] John Clemens. Automatic classification of object code using machine learning. *DFRWS 2015 USA*, 2015.

[12] TIS Committee. *Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification Version 1.2*, 1995.

[13] Debian. Debian Repository main pool. `http://ftp.halifax.rwth-aachen.de/debian/pool/main/`.

[14] devttys0. Binwalk Firmware Analysis Tool. `http://binwalk.org/`.

[15] devttys0. What the Ridiculous Fuck, D-Link?! `http://www.devttys0.com/2015/04/what-the-ridiculous-fuck-d-link/`, 2015.

[16] Chris Eagle. *The IDA Pro Book*. no starch press, 2008.

[17] eliben. pyelftools. `https://github.com/eliben/pyelftools`.

[18] Firmadyne. Towards Automated Dynamic Analysis for Linux-based Embedded Firmware. https://github.com/firmadyne.

[19] Dan Grabham. From a small Acorn to 37 billion chips: ARM's ascent to tech superpower. http://www.techradar.com/news/computing/from-a-small-acorn-to-37-billion-chips-arm-s-ascent-to-tech-superpower-1167034.

[20] Hex-Rays. IDA: About. https://www.hex-rays.com/products/ida/index.shtml.

[21] IBM. PowerPC User Instruction Set Architecture. http://math-atlas.sourceforge.net/devel/assembly/ppc_isa.pdf, 2003.

[22] Imagination. *MIPS® Architecture For Programmers Volume I-A: Introduction to the MIPS32® Architecture*, 2014.

[23] ARM Infocenter. Cortex-A8 Technical Reference Manual. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0344k/Cdffdceb.html.

[24] Intel. *Intel® 64 and IA-32 Architectures Software Developer's Manual*, sep 2016.

[25] itwissen.info. Embedded System. http://itwissen.info/definition/lexikon/Embedded-System-ES-embedded-system.html.

[26] Prof. Dr. Peter Martini. Systemnahe Informatik SS 2013, Kapitel 1 Teil 1. Vorlesung, 2013.

[27] Prof. Dr. Peter Martini. Systemnahe Informatik SS 2013, Kapitel 1 Teil 2. Vorlesung, 2013.

[28] Joseph Menn. Russian researchers expose breakthrough U.S. spying program. http://www.reuters.com/article/us-usa-cyberspying-idUSKBN0LK1QV20150217, 2015.

[29] Microsoft. Was ist Windows PE? https://technet.microsoft.com/dede/library/cc766093(v=ws.10).aspx, 1995.

[30] Raspberry Pi. Teach, Learn, and Make with Raspberry Pi. https://www.raspberrypi.org/.

[31] Nguyen Anh Quynh. Capstone The Ultimate Disassembler. http://www.capstone-engine.org/.

[32] Michael Meier Viviane Zwanger. Codescanner: Detecting (Hidden) x86/x64 Code in Arbitrary Files. *Malcon 2014*, 2014.

[33] wiibrew.org. Assembler Tutorial. http://wiibrew.org/wiki/Assembler_Tutorial.

[34] Wikipedia. Binary Large Object — Wikipedia, Die freie Enzyklopädie. https://de.wikipedia.org/w/index.php?title=Binary_Large_Object&oldid=146999404, 2015.

[35] Wikipedia. ARM-Architektur — Wikipedia, Die freie Enzyklopädie. https://de.wikipedia.org/w/index.php?title=ARM-Architektur&oldid=157803847, 2016. [Online; Stand 29. Oktober 2016].

[36] Wikipedia. Eingebettetes System — Wikipedia, Die freie Enzyklopädie. https://de.wikipedia.org/w/index.php?title=Eingebettetes_System&oldid=154974430, 2016.

[37] Wikipedia. Firmware — Wikipedia, Die freie Enzyklopädie. https://de.wikipedia.org/w/index.php?title=Firmware&oldid=159351088, 2016. [Online; Stand 8. November 2016].

[38] Wikipedia. MIPS-Architektur — Wikipedia, Die freie Enzyklopädie. https://de.wikipedia.org/w/index.php?title=MIPS-Architektur&oldid=158933810, 2016. [Online; Stand 29. Oktober 2016].

[39] Wikipedia. PowerPC — Wikipedia, Die freie Enzyklopädie. https://de.wikipedia.org/w/index.php?title=PowerPC&oldid=157905656, 2016.

[40] Wikipedia. X86-Prozessor — Wikipedia, Die freie Enzyklopädie. https://de.wikipedia.org/w/index.php?title=X86-Prozessor&oldid=154362949, 2016. [Online; Stand 29. Oktober 2016].

[41] Jonas Zaddach and Andrei Costin. Embedded devices security and firmware reverse engineering. *Black-Hat USA*, 2013.