Arbeitsgruppe Programmiersprachen und Übersetzerkonstruktion
Institut für Informatik
Christian-Albrechts-Universität zu Kiel

Master-Projekt

Kianxali:
Kieler Analyzer for Executables and Libraries

vorgelegt von

Folke Will

Betreut von:	Prof. Dr. Michael Hanus
Version:	11. Mai 2014

Inhaltsverzeichnis

1 Einleitung

Kianxali (Kieler Analyzer for Executables and Libraries) ist ein interaktiver Disassembler zur Analyse von Maschinencode. Interaktiv bedeutet dabei, dass der Benutzer Einfluss auf die Analyse nehmen kann – so beispielsweise gibt es die Möglichkeit, disassemblierte Funktionen umzubenennen, um sie an anderer Stelle beim Aufruf am Namen zu erkennen. Zusätzlich gibt es eine Schnittstelle zur Skriptsprache *Ruby*, um Interaktionen mit dem Maschinencode zu automatisieren.

1.1 Motivation

Ein Disassembler dient der Analyse von Maschinencode und wird deshalb hauptsächlich zur Analyse von Programmen eingesetzt, deren Quellcode nicht verfügbar ist. Dies ist insbesondere bei Schadsoftware wie beispielsweise Viren, Botnetzen, Rootkits und trojanischen Pferden der Fall. Ein Disassembler kann hier dabei helfen, die Funktion des Programms aufzudecken.

Es existieren bereits frei verfügbare Disassembler – `objdump` ist als Bestandteil der GNU Binutils[1] weit verbreitet. Allerdings benutzt `objdump` den Ansatz des *linear sweep*, der gut zur Analyse einzelner Funktionen eines Maschinenprogramms geeignet ist, bei der Analyse eines ganzen Programms allerdings häufig fehlschlägt. Ein besser geeigneter Ansatz ist der des *recursive traversal*, der in freien Disassemblern üblicherweise nicht implementiert ist. Die Algorithmen werden in Abschnitt 3.2 erläutert.

Ein weiteres Problem bei der Analyse eines Maschinenprogramms ist der Umgang mit der großen Menge an Assembler-Instruktionen, die im Programm enthalten sind. Ohne die Möglichkeit, einzelne Bereiche im disassemblierten Code zu kommentieren, zusammenhänge Instruktionen als Funktion zu erkennen, diese benennen zu können und referenzierte Zeichenketten im Code zu finden, ist die Analyse sehr aufwändig.

In Kianxali soll daher der Ansatz des *recursive traversal* umgesetzt werden, um eine große und korrekte Abdeckung des Maschinencodes zu erhalten. Weiterhin soll der Benutzer die Möglichkeit haben, mit dem disassemblierten Code auf die oben beschriebene Weise zu interagieren. Als Alleinstellungsmerkmal gegenüber freien Disassemblern soll es außerdem möglich sein, mithilfe der Skriptsprache Ruby automatisierte Operationen auf dem disassemblierten Code durchzuführen. Dadurch wird es möglich, verschiedene Ansätze zur Obfuscation auf automatisierte Umkehrung untersuchen zu können.

1.2 Zielsetzung

Es ergeben sich daher folgende Anforderungen an die jeweiligen Komponenten des zu entwickelnden Disassemblers:

Decodierer

- Abbildung aller Instruktionen im x86-Basis-Befehlssatz
- Kenntnis der Semantik von wichtigen Befehlen, um den Kontrollflussgraph des Programms zu ermitteln und zu verstehen, an welchen Stellen Daten wie Zeichenketten aus dem Datensegment der ausführbarten Datei benutzt werden

[1] http://www.gnu.org/software/binutils/

Disassembler

- Umsetzung des Recursive-Traversal-Ansatzes
- Erkennen von nicht direkt angesprungenen Funktionen mittels einer Heuristik
- Verständnis des Microsoft Windows PE-Dateiformats (EXE-Dateien), um den Maschinencode innerhalb der ausführbaren Datei lokalisieren zu können

Schnittstelle

- Schnittstelle zur Skriptsprache Ruby, um Analysen zu automatisieren
- Grafische Oberfläche, um durch den Maschinencode zu navigieren
- Annotieren des Maschinencodes durch den Benutzer (Benennung von Funktionen, Einfügen von Kommentaren)

2 Grundlagen

Da Schadcode hauptsächlich Windows-Systeme betrifft und diese x86-kompatible Prozessoren verwenden, werden das Instruktionsformat und der Aufbau von ausführbaren Windows-Dateien kurz beschrieben.

2.1 Der x86-Befehlssatz

Zum Verständnis dieses Abschnitts sind die Intel-Handbücher [Int14a], [Int14b] und [Int14c] zum x86-Prozessor erforderlich, da dort die Tabellen zur Codierung der Befehle beschrieben sind. Ein x86-kompatibler Prozessor besitzt 8 Basisregister (vgl. [Int14a, Kap. 3-10]), die in den ersten Generationen des Intel 8086-Prozessors für fest definierte Zwecke bestimmt waren:

- AX: Akkumulator. Wurde als Operandenregister für arithmetische Operationen benutzt.
- CX: Counter. Wird bei Schleifeninstruktionen als Zählregister verwendet.
- DX: Data. Wurde als Operandenregister für arithmetische Operationen benutzt.
- BX: Base. Diente als Zeigerregister für indirekte Speicherzugriffe.
- SP: Stack-Pointer. Zeigt auf die aktuelle Adresse des nächsten freien Stack-Elements.
- BP: Base-Pointer. Enthält die Adresse des Prozedurrahmens der aktuellen Funktion.
- SI: Source-Index. Enthält bei String- und Speicher-Operationen die Quelladresse.
- DI: Destination-Index. Enthält bei String- und Speicher-Operationen die Zieladresse.

Diese Register haben eine Breite von 16 Bit. Bei den ersten 4 Registern lassen sich das höherwertige Byte (High-Byte) und niederwertige Byte (Low-Byte) einzeln ansprechen, dazu wird das Suffix L beziehungsweise H statt X verwendet. AX besteht also aus den Teilen AH und AL.

Bei der Einführung neuerer Generationen fielen viele Beschränkungen der ersten 4 Register weg, sodass nun alle Register als Quelle oder Ziel arithmetischer Operationen dienen können. Die historischen Namen wurden allerdings aus Kompatibilitätsgründen beibehalten. Mit dem

R	64-Bit	Unterste 32b	Unterste 16b	Davon obere 8b	Unterste 8b
0	RAX	EAX	AX	AH	AL
1	RCX	ECX	CX	CH	CL
2	RDX	EDX	DX	DH	DL
3	RBX	EBX	BX	BH	BL
4	RSP	ESP	SP	-	SPL (nur x64)
5	RBP	EBP	BP	-	BPL (nur x64)
6	RSI	ESI	SI	-	SIL (nur x64)
7	RSI	EDI	DI	-	DIL (nur x64)
8	R8	R8D	R8W	-	R8B
9	R9	R9D	R9W	-	R9B
10	R10	R10D	R10W	-	R10B
11	R11	R11D	R11W	-	R11B
12	R12	R12D	R12W	-	R12B
13	R13	R13D	R13W	-	R13B
14	R14	R14D	R14W	-	R14B
15	R15	R15D	R15W	-	R15B

Tabelle 1: Die Tabelle zeigt die verfügbaren Register der x86-Familie. Die unteren 8 Register sind dabei ausschließlich im 64-Bit-Modus verfügbar. Ebenfalls sind die mit x64 markierten 8-Bit-Register nur im 64-Bit-Modus verfügbar. Quelle: nach [Int14a, Kap. 3-12].

Intel 80386-Prozessor wurde die Breite der 8 Register auf 32 Bit erhöht. Aus Gründen der Kompatibilität haben die Register unter den obigen Bezeichnungen allerdings weiterhin nur 16 beziehungsweise 8 Bit; zur Verwendung der vollen 32 Bit wird dem Registernamen ein E vorangestellt, beispielsweise EAX.

Seit den späteren Versionen des Intel Pentium IV-Prozessors unterstützen die Intel-Prozessoren die von AMD entwickelte Erweiterung x86-64, die eine abwärtskompatible 64-Bit-Unterstützung bietet. Dazu wurde die Breite der obigen 8 Register auf 64 Bit erhöht, die durch die Voranstellung eines R an den Registernamen gekennzeichnet sind, beispielsweise RAX. Die 8 neuen Register tragen die Namen R8 bis R15. Bei diesen tragen die 32-Bit-Varianten die Bezeichner R8D bis R15D, die 16-Bit-Varianten R8W bis R15W und die 8-Bit-Varianten R8B bis R16B. Dabei stehen W für Wort (Wortbreite 8086: 16 Bit), D für Doubleword (32 Bit) und B für Byte (8 Bit). Auf den höherwertigen Teil der 16-Bit-Variante kann bei den neuen 8-Bit-Registern nicht einzeln zugegriffen werden. Dafür wurden die 4 Register SP, BP, SI und DI um einen Zugriff auf das niederwertige Byte erweitert, die tragen die Bezeichner SPL, BPL, SIL und DIL. Tabelle 1 zeigt alle frei verwendbaren Register der x86-Familie.

Befehle werden durch sogenannte *Opcodes* codiert. Dabei handelt es sich um ein oder mehrere Bytes, die einen Befehl darstellen. Je nach Alter des Befehls gibt es verschiedene Varianten, die Operanden eines Befehls zu codieren:

Bestandteil des Opcodes: Insbesondere bei den alten, bereits in der 8086-Architektur verfügbaren Befehlen sind die Operanden direkter Bestandteil des Opcodes. Der Opcode `04` ist beispielsweise eine Variante der `add`-Instruktion, bei der als Ziel das Register AL vorgesehen

ist. Soll ein anderes Zielregister verwendet werden, so muss ein anderer Opcode für die gleiche Instruktion verwendet werden (vgl. dazu [Int14b, Kap. 3-27]).

Codierung im Opcode: Einige Instruktionen codieren das verwendete Register in den untersten 3 Bit eines Opcodes. Ein Beispiel dafür ist Opcode `50`: die unteren 3 Bits codieren das Register, sodass `53` der Instruktion `push ebx` entspricht (vgl. auch Tabelle 1). Diese Art der Codierung ist im Intel-Handbuch als `+r` angegeben, da der Register-Index sozusagen auf den Opcode addiert wird (vgl. [Int14c, Kap. 3-267]).

Codierung durch ModR/M-Byte: Die am häufigsten verwendete Codierung ist das sogenannte ModR/M-Byte. Dieses Byte wird nach den Opcodes codiert, die ein ModR/M-Byte erwarten, und enthält eine Angabe über eine indirekte Speicheradressierung sowie das Zielregister der Instruktion. Es hat folgenden Aufbau, der in [Int14b, Kap. 2-3] genauer beschrieben ist:

- mod: die obersten 2 Bit bestimmen den Modus des Speicherzugriffs. Modus 0 greift auf eine Adresse im Speicher zu, die durch das Register im R-Teil des Bytes angegeben sind. Modus 1 addiert zusätzlich eine 8-Bit-Konstante auf den Inhalt des Registers, die auf das ModR/M-Byte folgen muss. Modus 3 funktioniert analog mit einer 32-Bit-Konstante. Modus 3 wird schließlich verwendet, wenn die Quelle auch ein Register und keine Speicheradresse ist.
- R: die folgenden 3 Bit bestimmen das Zielregister. Bei Instruktionen, die nur einen Operanden benötigen, werden diese Bits verwendet, um den Opcode zu erweitern und dadurch weitere Instruktionen zu unterscheiden.
- M: die untersten 3 Bit bestimmen das Register, das die Speicheradresse enthält (bei Modus 3 wird der Inhalt direkt verwendet, ohne ihn als Speicheradresse zu interpretieren).

Bis auf 2 Ausnahmen entspricht die Codierung von M und R genau Tabelle 1. Wenn als Wert für M eine 4 codiert und der Modus ungleich 3 ist, folgt auf das ModR/M-Byte noch ein SIB-Byte, das im folgenden Absatz beschrieben wird. Wenn als Wert für M im Modus 0 eine 5 codiert ist, wird eine 32-Bit-Konstante nach dem Mod-R/M-Byte als Speicheradresse interpretiert, die nicht auf ein Register addiert, sondern auf die direkt zugegriffen wird.

Das SIB-Byte erlaubt für M = 4 ein Zugriff auf den Speicher der Art Basisregister + Skalierung · Indexregister + Offset. Der Aufbau ist wie folgt:

- Skalierung: die obersten 2 Bit bestimmen den Skalierungsfaktor für das Indexregister. Der Wert wird als Exponent zur Basis 2 interpretiert, sodass die möglichen Skalierungen 1, 2, 4 und 8 sind.
- Indexregister: die folgenden 3 Bit bestimmen das Indexregister. Der Wert 4 bedeutet dabei, dass kein Indexregister verwendet wird, der Aufbau ist dann also Basisregister + Offset, die Skalierung wird ignoriert.
- Basisregister: die untersten 3 Bit bestimmen das Basisregister. Der Wert 5 trägt je nach Modus des ModR/M-Bytes eine andere Bedeutung: Im Modus 0 wird die auf das SIB-Byte folgende 32-Bit-Konstante als Basisadresse interpretiert, statt sie aus einem

Register zu laden. Im Modus 1 folgt auf das SIB-Byte eine 8-Bit-Adresse, die auf das Register EBP addiert wird, wobei das Ergebnis als Basisadresse interpretiert und nicht in EBP gespeichert wird. Im Modus 3 gilt das gleiche mit einer 32-Bit-Konstante.

Es folgen einige Beispiele zum ModR/M-Byte:

- `23 37`: Opcode `23` steht für eine `and`-Instruktion zwischen Register und Speicher. In Binärdarstellung entspricht das Mod-R/M-Byte (37) dem Wert 00.110.111. Der Modus ist also 0, R = 6 und M = 7. Die Und-Verknüpfung findet also zwischen dem Register ESI und dem Speicherinhalt an der durch EDI gegebenen Adresse statt, die Decodierung der Instruktion lautet also `and ESI, [EDI]`.
- `23 51 12`: Der Opcode entspricht wieder der `and`-Instruktion, das ModR/M-Byte ist 01.010.001, also Modus 1 mit R = 2 und M = 1. Im Modus 1 wird nach dem ModR/M-Byte eine 8-Bit-Konstante erwartet, die auf das Register M addiert wird, in diesem Fall 12. Die decodierte Instruktion lautet also `and EDX, [ECX + 12]`
- `23 E2`: Der Opcode entspricht wieder der `and`-Instruktion, das ModR/M-Byte ist 11.100.010, also Modus 3 mit R = 4 und M = 2. Im Modus 4 wird der Inhalt von Register M nicht als Speicheradresse interpretiert, sodass die Decodierung `and ESP, EDX` lautet.
- `23 04 8A`: Der Opcode entspricht wieder der `and`-Instruktion, der Modus ist 0 und es gelten R = 0 und M = 4. Das Zielregister ist also `EAX`, die Quelle wird über ein SIB-Byte gegeben, das auf das ModR/M-Byte folgt und hier den Wert 8A hat, was in binärer Darstellung 10.001.010 entspricht. Die Skalierung ist also $2^2 = 4$, das Indexregister 1 und das Basisregister 2. Insgesamt lautet der Quelloperand also `[EDX + 4 * ECX]`, wodurch die Instruktion insgesamt als `and EAX, [EDX + 4 * ECX]` decodiert wird.
- `80 E8 05`: Der Opcode `80` gehört zu 8 verschiedenen Instruktionen, die einen Speicherinhalt als 8-Bit-Wert und eine 8-Bit-Konstante verknüpfen. Der erste Operand wird via ModR/M-Byte angegeben, der zweite direkt als Konstante codiert. Da der zweite Operand somit nicht Teil des ModR/M-Bytes ist, wird dessen R-Wert als Opcode-Erweiterung genutzt und bestimmt, welche der 8 möglichen Instruktionen codiert wird. In diesem Fall hat das ModR/M-Byte Modus 3, R = 5 und M = 0. Da der Modus 3 ist, wird Register AL mit der Konstante 5 verknüpft. R = 5 steht in diesem Fall für die `sub`-Instruktion, sodass die Decodierung `sub AL, 5` lautet.

Weitere Details sind im Intel-Handbuch [Int14b, Kap. 2-4] beschrieben.

2.2 Binärformate für Maschinencode

Ein ausführbares Programm benutzt üblicherweise Funktionen des Betriebssystems oder aus anderen Bibliotheken, um mit seiner Umgebung zu interagieren. Beim Starten eines Programms muss das Betriebssystem diese Bibliotheken laden, damit das Programm sie benutzen kann. Dabei muss das Problem gelöst werden, dass der interne Aufbau der verwendeten Bibliotheken über verschiedene Versionen nicht konstant ist – insbesondere ist also nicht bekannt, an welcher Adresse eine benötigte Funktion sich befinden wird. Das gleiche gilt für die Funktionen, die das Betriebssystem selbst bereitstellt. Ein anderes Problem ist, dass ein Programm seine eigene Adresse im physikalischen Speicher bei der Ausführung nicht kennt –

es sind also bereits Aufrufe von Funktionen innerhalb des Programms problematisch. Durch die Verbreitung von Multi-Tasking ist es nicht möglich, einem Programm die gewünschte Adresse im Hauptspeicher selbst bestimmen zu lassen, denn dann könnten zwei Programme von überlappenden Adressbereichen nicht gleichzeitig gestartet werden. Unter MS-DOS war dies noch der Fall, dort konnten COM-Dateien Maschinencode enthalten, der direkt ausgeführt wurde – in diesen Dateien befand sich ausschließlich Maschinencode und die zugehörigen Daten.

Moderne Betriebssysteme lösen Probleme dieser Art durch die Verwendung virtuellen Speichers: Jedes Programm wird in einen eigenen, isolierten Adressbereich geladen. Dies ist auf x86-Prozessoren seit dem 80286-Prozessor möglich – durch die Einführung der *Memory Management Unit* (MMU) kann das Betriebssystem für jeden Prozess getrennt konfigurieren, wie virtuelle Adressen auf den realen Hauptspeicher abgebildet werden sollen. Dadurch ist es möglich, dass Programme selbst angeben können, an welche virtuelle Adresse sie geladen werden sollen. Weiterhin erlaubt die MMU, die Zugriffsrechte auf die Speicherbereiche zu ändern. Virtuelle Speicherbereiche können als lesbar, schreibbar und ausführbar gekennzeichnet sein. Aus diesem Grund haben ausführbare Programme heutzutage ein bestimmtes Binärformat, das folgenden Informationen beinhaltet:

- Speicher-Layout: da die MMU unterschiedliche Zugriffsrechte auf Speicherbereiche erlaubt, werden Programme vom Linker in verschiedene Bereiche aufgeteilt, die innerhalb der Binärformate *Sektionen* genannt werden. Eine Sektion hat eine Adresse innerhalb der Datei, die oft als *offset* bezeichnet wird, da die Werte sich relativ auf den Beginn der Datei beziehen. Zusätzlich hat eine Sektion eine virtuelle Adresse, an die das Betriebssystem den Inhalt der Sektion laden wird. Auch die Größe einer Sektion wird im Binärformat spezifiziert. Tabelle 2 zeigt häufig verwendete Sektionen eines Programms.
- Einstiegspunkt: die virtuelle Adresse, bei der die Ausführung des Programms beginnen soll. Dies ist in der Regel nicht die Adresse von `main`, da vorher ein Stack eingerichtet werden muss, die Parameter für `main` vom Betriebssystem erfragt werden müssen und einige andere Abläufe erledigt werden müssen. Dieser Code wird als *startup code* bezeichnet und ruft am Ende `main` auf.
- Bibliotheken: wenn Funktionen des Betriebssystems oder auf anderen Bibliotheken aufgerufen werden sollen, entsteht das oben beschriebene Problem, dass deren Adressen dem Linker nicht bekannt sind. Hier hilft auch der virtuelle Speicher nicht, da das Programm nicht verlangen kann, dass bestimmte Funktionen immer an derselben Adresse geladen werden sollen. Ansonsten könnte das Betriebssystem Bibliotheken, die von mehreren Programmen gleichzeitig genutzt werden, nicht nur einmalig im Speicher halten, sondern müsste sie für jedes Programm nach dessen Wünschen an definierte Adressen laden. Stattdessen werden die Funktionen über die *Namen* von Bibliothek und Funktion importiert. Beim Laden füllt das Betriebssystem eine Tabelle, die die Adressen aller benötigten Funktionen enthält. Diese wird je nach System als *stub table*, *import address table* oder *symbol table* bezeichnet. Wird beispielsweise die Funktion `printf` verwendet, so wird dazu folgende Instruktion benutzt: `call [printf_stub]`. Dabei ist `printf_stub` die Adresse innerhalb der Tabelle, an der das Betriebssystem beim Laden die Adresse der tatsächlichen Funktion hinterlegen

Name	Verwendung	Rechte
.text	Programmcode	ausführbar, lesbar
.data	Initialisierte Daten: Zeichenketten, Konstanten usw.	lesbar, schreibbar
.bss	Uninitialisierte Daten: wird beim Laden mit 0 initalisiert und belegt keinen Platz im Programm	lesbar, schreibbar

Tabelle 2: Die Tabelle zeigt übliche Sektionen in Binärformaten für Maschinencode. Sektionen werden verwendet, um unterschiedliche Rechte für den virtuellen Speicher zu verwenden, damit der Programmcode sich beispielsweise nicht zur Laufzeit ändern kann. Ein weiterer Vorteil ist, dass eine Sektion angelegt werden kann, die in der eigentlichen Datei gar nicht vorhanden ist und vom Betriebssystem mit Nullen initialisiert wird: Auf diese Weise bleiben die Dateien kleiner, da alle nicht-initialisierten globalen oder statische Variablen in dieser Sektion untergebracht werden können

wird. Eine andere Variante ist die Verwendung einer sogenannten Trampolin-Funktion: Hier wird eine normale `call`-Instruktion ohne Adress-Derefenzierung verwendet, also beispisweise `call printf_trampoline`. Diese Trampolin-Funktion enthält nur eine einzelne Sprunganweisung, deren Adresse das Betriebssystem beim Laden einfügen wird, sodass an die korrekte Funktion gesprungen wird.

3 Planung

Zur Umsetzung des Programms wird die Aufgabe in folgende Komponenten zerteilt:

- Lader: liest eine Datei mit Maschinencode ein und stellt Methoden bereit, um zwischen virtuellen Speicheradressen und Adressen innerhalb der Datei konvertieren zu können.
- Decodierer: decodiert einzelne Opcodes. Als Eingabe dient eine Bytesequenz, die als Opcode interpretiert wird. Die Ausgabe ist ein Objekt, das die Instruktion darstellt und Informationen über die Anweisung und ihre Operanden enthält.
- Disassembler: nutzt den Decodierer, um Sequenzen von Opcodes zu disassemblieren. Dabei werden höhere Strukturen wie Funktionen erkannt und der Zusammenhang zwischen Instruktionen und Datenstrukturen wie Zeichenketten wird hergestellt.
- GUI: nutzt den Disassembler, um den disassemblierten Code einer ausführbaren Datei darzustellen und den Benutzer damit interagieren zu lassen.
- Ruby-Anbindung: stellt eine API für den Benutzer bereit, um mit dem disassemblierten Code mittels Ruby-Skripts zu interagieren.

3.1 Decodier-Strategie

Der Decoder übersetzt eine Bytesequenz in eine Assembler-Instruktion und stellt Informationen über die Instruktion sowie deren Operanden bereit. Da der x86-Befehlssatz durch viele Erweiterungen mehr als 700 Instruktionen besitzt, von denen viele durch unterschiedliche Operandentypen mehrere Opcodes haben, wäre die vollständige Entwicklung eines eigenen

Decoders sehr aufwändig und fehleranfällig. Deshalb werden bestehende Projekte für die Umsetzung des Decoders evaluiert:

- udis86[2]: C-Bibliothek, die eine Bytesequenz in Assembler-Instruktionen decodiert. Die Instruktionen werden dabei in eine Datenstruktur bereitgestellt, die Informationen über die Instruktionen und ihre Operanden enthält. Die Informationen sind für dieses Projekt allerdings nicht ausreichend – es fehlt bei Operanden beispielsweise die Angabe, ob sie Quell- oder Zieloperand ist. Dadurch wäre es nicht möglich, automatisiert festzustellen, ob ein Speicherzugriff auf den Code des Programms lesend oder schreibend ist. Diese Information wäre allerdings notwendig, um selbstmodifizierenden Code im Programm zu finden.
- x86ref[3]: XML-Dokument, das alle Informationen über Instruktionen und Operanden bereitstellt, die in der offiziellen Intel-Dokumentation angegeben sind. Dazu gehört auch eine kurze Beschreibung der Instruktion, die beispielsweise für eine Hilfe-Funktion bei unbekannten Instruktionen benutzt werden kann.
- BeaEngine[4]: C-Bibliothek analog zu udis86, die mehr Informationen über die decodierten Instruktionen bereitstellt. Es werden allerdings keine Beschreibungen der Instruktionen für eine Hilfe-Funktion bereitgestellt.

Die Verwendung des XML-Dokuments erscheint am flexibelsten, da alle relevanten Informationen über alle Instruktionen enthalten sind und erweiterte Befehlssätze durch Austauschen der XML-Datei bereitgestellt werden können. Ein weiterer Vorteil ist, dass das Dokument auch die Entwicklung eines Assemblers ermöglicht, während die obigen Bibliotheken nur disassemblieren können. Die Möglichkeit der Assemblierung wäre sinnvoll, um dem Benutzer den Austausch einer Instruktion durch eine andere zu ermöglichen. Daher wird der Decodierer auf Basis des XML-Dokuments entwickelt.

3.2 Disassembler-Strategie

Die Disassembler-Komponente benutzt den Decoder, um die Instruktionen in Funktionen zu gruppieren, Datenstrukturen zu erkennen, Zeichenketten zu decodieren und aus diesen Informationen eine Datenstruktur zu erstellen, die für automatisierte (Ruby-Skripte) und menschliche (Ansicht in der GUI) Verarbeitung genutzt werden kann. Für das Vorgehen eines Disassemblers gibt es verschiedene Verfahren [SDA02], von denen zwei beschrieben werden:

- linear sweep: alle Sektionen, die innerhalb des Programms als ausführbar gekennzeichnet sind, werden als Instruktionssequenzen aufgefasst und disassembliert. Dieser Ansatz ist sehr einfach, da die Datei linear verarbeitet wird und der Disassembler keinerlei Informationen über die Semantik der Befehle kennen muss. Ein Nachteil ist allerdings, dass das Vorhandensein von Daten-Bytes innerhalb der Code-Sektionen dafür sorgt, dass die Analyse ab diesen Bytes vollständig fehlschlagen kann. Der Grund ist, dass x86-Opcodes eine unterschiedliche Länge haben; der Disassembler muss also genau dem Fluss der Instruktionen folgen, um die Grenzen zwischen zwei Instruktionen zu erkennen. Befindet sich nun ein Datenbyte zwischen den Instruktionen, so wird ein

[2] http://udis86.sourceforge.net/
[3] http://ref.x86asm.net/
[4] http://www.beaengine.org/

Pseudocode 1 recursive traversal

```
toVisit ← {programEntryPoint}
while toVisit ≠ ∅ do
    address ← takeFromSet(toVisit)
    instruction ← decodeAt(address)
    if instruction has branch addresses then
        toVisit ← toVisit ∪ branch addresses
    end if
    if execution flow continues after instruction then
        toVisit ← toVisit ∪{address + size(instruction)}
    end if
end while
```

Disassembler sie bei diesem Ansatz fälschlicherweise als Instruktionen decodieren und dadurch die Grenzen zwischen allen folgenden Instruktionen nicht mehr erkennen, sodass die Analyse ab dem Datenbyte falsch ist. Das Einfügen von Datenbytes zwischen Instruktionen ist aber ein normales Vorgehen von Compilern: So werden beispielsweise Sprungtabellen für Switch-Case-Anweisungen einer Hochsprache innerhalb der Funktion abgelegt, damit diese möglichst im gleichen Block des Caches landen und somit schnell erreichbar sind.

- recursive traversal: Bei diesem Ansatz folgt der Disassembler dem Programmfluss des analysierten Programms. Zunächst werden die Anweisung sequentiell analysiert. Sobald aber eine Instruktion decodiert wurde, die den Programmfluss verzweigen lässt, werden die Zieladressen gespeichert. Wird das Ende eines Pfades im Programmfluss erreicht, beispielsweise durch eine Instruktion zum Verlassen einer Prozedur, so stoppt die Analyse an dieser Instruktion und setzt an einer der zuvor gespeicherten Adressen fort. Auf diese Weise werden nur Instruktionen disassembliert, die tatsächlich im Programmfluss erreichbar sind. Der Nachteil dieses Ansatzes ist, dass Instruktionen nicht erreicht werden, wenn sie vom Programm nicht direkt durch Verzweigungen erreichbar sind. Dies ist beispielsweise bei Switch-Case-Anweisungen der Fall, da hier die Zieladresse eines Sprungs nicht direkt in der Instruktion, sondern indirekt in einer Tabelle codiert ist. Ein weiteres Beispiel sind Tabellen mit Adressen von überschriebenen Methoden bei objektorientierter Programmierung: Hier besitzt jede Instanz einer Klasse eine Tabelle, die Adressen der Methoden der zugehörigen Klasse enthält. Zur Umsetzung dieses Ansatzes muss der Disassembler erkennen können, ob es sich bei einer Instruktion um eine Verzweigung handelt und an welche Zieladressen verzweigt werden kann. Der Algorithmus ist in Pseudocode 1 beschrieben.

Der Nachteil des rekursiven Ansatzes kann durch die Verwendung von Heuristiken gemildert werden. Es ist beispielsweise möglich, erkannte Instruktionen zu markieren und nach der Analyse alle nicht besuchten Bereiche innerhalb der Code-Segmente mittels des linearen Ansatzes zu untersuchen. Diese Technik ist als *hybrider Ansatz* bekannt. Da der rekursive Ansatz robuster ist, wird er für den Disassembler gewählt.

3.3 Wahl der Programmiersprache

Da das Projekt über eine Ruby-Schnittstelle verfügen soll, wird die Wahl der Programmiersprache auf C, C++ und Java eingeschränkt, denn hierfür gibt es einfache Möglichkeiten der Einbettung seitens Ruby. Die Anbindung an C ist allerdings nicht sehr intuitiv, da Ruby eine stark objektorientierte Sprache ist, während C dieses Konzept nicht kennt. Zur Umsetzung einer API müsste also viel Aufwand betrieben werden, um die Ruby-Klassen mit C-Funktionen zu verbinden. In C++ ist dieses Vorgehen intuitiver, da die Ruby-Klassen direkt mit C++-Klassen verbunden werden können. In Java ist dieser Aufwand minimal, da dank *Reflection* alle Informationen über Klassen, Objekte und Methoden zur Laufzeit zur Verfügung stehen. Die Bibliothek JRuby[5] nutzt dieses Vorgehen und erlaubt dadurch eine Interaktion zwischen Ruby-Code und Java-Code, ohne dass der Programmierer Schnittstellen definieren muss. Da Java auch die Umsetzung einer GUI und das Einlesen des XML-Dokuments für den Decoder mit der Java-Standardbibliothek erlaubt, wird das Projekt mit Java umgesetzt.

4 Implementierung

Die Umsetzung des Projekts erfolgte gemäß der oben beschriebenen Planung. Die Paketstruktur des Quellcodes orientiert sich daher an den in Abschnitt 3 geplanten Komponenten:

- kianxali.loader: Enthält Klassen zum Laden von Maschinencode-Binärformaten wie PE (Windows), ELF (Unix, Linux), Mach-O (OS X). Für jedes Format existiert eine statische Methode zum Prüfen, ob eine Datei zum jeweiligen Formaten gehört. Dies wird anhand der ersten Bytes erkannt, da jedes Format dort eine eigene Signatur besitzt. Die einzelnen Lader erlauben das Navigieren innerhalb einer Maschinencode-Datei durch die Angabe virtueller Speicheradressen. Auch Adressen von Betriebssystem-Prozeduren werden erkannt und gespeichert, sodass die Assembler-Listings später damit annotiert werden können.
- kianxali.decoder: Decodiert in einem Datenstrom aus Opcodes die erste Instruktion und ihre Operanden. Die Instruktion wird dabei mit allen Informationen des XML-Datei annotiert.
- kianxali.disassembler: Nutzt den Decoder und den Lader, um Basisblöcke und Funktionen innerhalb der Maschinencode-Datei zu finden und in einer Datenstruktur zu speichern. Dazu werden auch Heuristiken verwendet, um spezielle Konstrukte wie Sprungtabellen, Zeichenketten und über indirekte Sprünge aufgerufene Funktionen zu erkennen.
- kianxali.gui: Stellt eine grafische Benutzeroberfläche für den Benutzer zur Verfügung, mit der Maschinencode-Dateien geladen und analysiert werden können.
- kianxali.scripting: Stellt mithilfe von JRuby eine Ruby-Schnittstelle zur Verfügung, die dem Benutzer eine automatisierte Interaktion mit den Datenstrukturen ermöglicht.

[5] http://jruby.org/

Typ	Breite	Vorzeichen	Java-Datentyp
UBYTE	8	Nein	short
SBYTE	8	Ja	byte
UWORD	16	Nein	int
SWORD	16	Ja	short
UDWORD	32	Nein	long
SDWORD	32	Ja	int
UQWORD	64	Nein	-
SQWORD	64	Ja	long

Tabelle 3: Die Tabelle zeigt die Datentypen der x86-Architektur. Da der 8086-Prozessor eine Wortbreite von 16 Bit besaß, gilt weiterhin die Konvention, dass der Word-Typ eine Breite von 16 Bit besitzt (vgl. [Int14a, Kap. 3-4]). DouleWord und QuadWord orientieren sich entsprechend daran. In Java 7 existieren keine primitiven Datentypen ohne Vorzeichen, weshalb bei der Verwendung von vorzeichenlosen x86-Typen der nächstgrößere Datentyp verwendet werden muss.

4.1 Lader

Die Lader-Klassen leiten von der abstrakten Basisklasse `ImageFile` ab und implementieren diese. Die Instanzen dieser Klasse dienen als zentrale Datenstruktur beim Decodieren des Maschinencodes. Sie kapseln den Zugriff auf die Datei, indem diese als navigierbarer Datenstrom dargestellt wird. Dieser Datenstrom ist in der Klasse `ByteSequence` implementiert und erlaubt das Lesen der typischen x86-Datentypen, die in Tabelle 3 beschrieben sind.

`ImageFile` kann eine `ByteSequence` auf eine virtuelle Speicheradresse positionieren, sodass Daten ab dieser Adresse gelesen werden können. Dazu wird die Methode `getByteSequence` verwendet. Die Lader-Implementierungen müssen dazu Methoden implementieren, die eine Umrechnung von virtuellen Speicheradressen in Adressen innerhalb der Datei ermöglichen. Folgende Lader-Klassen wurden implementiert:

- `PEFile`: Erlaubt das Laden und navigieren innerhalb von Dateien des Windows-Betriebssystems. Unterstützt werden dabei EXE-Dateien (ausführbare Dateien) und DLL-Dateien (dynamische Bibliotheken). Beim Laden wird auch das Format PE+ unterstützt, das die 64-Bit-Erweiterung des PE-Formats darstellt. Die Implementierung unterstützt das Auflösen der *Imports*, die Einbindungen anderer DLL-Dateien beschreiben. Zusätzlich wird die *Import Address Table* interpretiert, die Funktionen aus DLL-Dateien in Sprungtabellen des Codes abbildet. Dadurch sind Aufrufe aus Laufzeit-Bibliotheken mit DLL-Name und Funktionsname sichtbar, was die Analyse erleichtert. Da auch die Windows-API von Anwendungen in Form von DLL-Dateien eingebunden wird, sind alle Betriebssystem-Aufrufe im Code sichtbar. Zur Implementierung wurde die Spezifikation des PE-Formats von Microsoft [Mic13] verwendet.
- `ELFFile`: Dient zum Laden von UNIX- und Linux-Objektdateien. Unterstützt werden die 32- und die 64-Bit-Variante des Formats. Als CPU-Architekturen werden nur x86 und x86-64 unterstützt. Es können auch Programmbibliotheken (.so-Dateien) geladen werden. Das Laden von externen Symbolen wurde implementiert, sodass auch in diesem Format die Aufrufe von Funktionen aus anderen Bibliotheken sichtbar ist. Zur

Implementierung wurde die Beschreibung aus [Mau03] genutzt, die online verfügbar[6] ist. Da sich in der Beschreibung Fehler befinden, wurde zur Überprüfung und Korrektur zusätzlich die C-Header-Datei `elf.h` des Linux-Kernels genutzt.

- `MachOFile`: Stellt einen Lader für das neue Maschinencode-Format von Apple OS X bereit, das seit OS X Version 10.6 verwendet wird. Es werden die 32-Bit- und die 64-Bit-Variante des Formats unterstützt, wobei die CPU-Architektur auch hier auf x86 und x86-64 beschränkt ist. Für dieses Format wurde ebenfalls eine Erkennung von Aufrufen aus anderen Bibliotheken implementiert. Zur Implementierung wurde die Spezifikation von Apple [App09] verwendet.
- `FatFile`: OS X erlaubt die Unterbringung von Maschinencode für mehrere Architekturen innerhalb einer einzelnen Datei. In der Praxis wird dieses Format verwendet, um eine einzelne Datei für die Verwendung auf 32- und 64-Bit-Systemen bereitstellen zu können. Innerhalb eines solchen *Fat File* können mehrere Mach-O-Dateien gespeichert sein. Beim Laden einer solchen Datei kann der Benutzer auswählen, welche enthaltene Architektur analysiert werden soll. Die Beschreibung des Formats befindet sich ebenfalls in [App09].

4.2 Maschinencode-Decodierer

Zur Umsetzung des Opcode-Decodierers für die x86-Architektur wurde die in Abschnitt 3.1 erwähnte XML-Datei verwendet. Mit dieser kann eines Instanz von `X86Decoder` erzeugt werden. Die Datei wird mittels eines SAX-Parsers einmalig ausgewertet und in eine Liste aller Instruktionen überführt. Zu jeder Instruktion werden die nötigen Operanden und alle weiteren Informationen der XML-Datei gespeichert. Der Parser für diese Aufgabe ist in der Klasse `XMLParserX86` realisiert.

Die Liste wird in einen Präfixbaum überführt, der in der Klasse `X86Decoder` erzeugt wird. Um zukünftig auch andere Architekturen als x86 unterstützen zu können, ist die Implementierung des Präfixbaums nicht an eine Architektur gebunden. Von der Wurzel des Baums führt jedes gelesene Byte des Opcode-Stroms zu einem Knoten, der eine Liste aller Opcodes enthält, die die gelesenen Bytes als Präfix enthalten. Da verschiedene Opcodes identische Präfixe besitzen können und die Entscheidung dann erst beim Decodieren der Operanden geschehen kann, befinden sich die möglichen Opcodes nicht nur in den Blättern des Baums. Zur Entscheidung wird das *Principle of Longest Match* verwendet, d.h. der Opcode mit dem längsten passenden Präfix wird zuerst untersucht. Falls die Operanden nicht zu diesem Opcode passen, werden die Opcodes mit kürzeren Präfixen untersucht. Diese Datenstruktur ist in der Klasse `DecodeTree` umgesetzt. Da einige Opcodes bei der Einführung späterer Prozessormodelle obsolet wurden und durch andere ersetzt wurden, muss der Decoder wissen, welche Variante der x86-Architektur verwendet wird. Dies wird in der Klasse `X86Context` realisiert. Der Kontext wird von der Lader-Klasse bereitgestellt, da im Maschinencode-Format spezifiziert ist, für welches Modell der Code vorgesehen ist.

Nachdem feststeht, zu welchem Opcode die gelesenen Bytes gehören, werden in der Klasse `X86Instruction` die Operanden gelesen. Die Klasse leitet von `Instruction` ab, damit später auch andere Architekturen unterstützt werden können. Als Schnittstelle dient die oben beschriebene `ByteSequence`, die vom Lader bereitgestellt wird. Die Methode `decodeNext`

[6] http://www.linux-kernel.de/appendix/ap05.pdf

des Decoders erhält als Parameter eine `ByteSequence` und einen `DecodeTree` und liest die nächste `Instruction` aus dem Strom, die schließlich zurückgegeben wird.

4.3 Disassembler

Der eigentliche Disassembler ist vollständig unabhängig von der Architektur, er arbeitet ausschließlich mit Schnittstellen und abstrakten Klassen. Die Klasse `Disassembler` wird mit einer Instanz von `ImageFile` erzeugt und füllt eine Datenstruktur vom Typ `DisassemblyData`. Der Ablauf bei der Initialisierung ist wie folgt:

1. Erzeugung einer `ImageFile`-Instanz mittels einer Lader-Klasse
2. Übergabe der `ImageFile`-Instanz an `Disassembler`
3. Disassembler-Instanz besorgt eine `Context`-Instanz durch das ImageFile
4. Context-Instanz stellt eine `Decoder`-Instanz für den Disassembler bereit

Dadurch ist der Disassembler in der Lage, einzelne Instruktionen zu decodieren. Im `ImageFile` ist die Startadresse des Maschinencodes bekannt, der sogenannte *entry point*. An diesem wird die Analyse des Codes gestartet. Es werden nun so lange Instruktionen decodiert, bis eine Instruktion auftritt, die den Programmfluss auf nicht-lineare Weise verzweigt – beispielsweise ein unbedingter Sprung oder eine Instruktion zum Verlassen einer Hochsprachenprozedur. Diese Eigenschaft wird geprüft, indem die Methode `stopsTrace` der decodierten `Instruction` aufgerufen wird. Bei allen Instruktionen wird in der Methode `examineInstruction` geprüft, ob die Instruktion den Programmfluss verzweigen kann und ob die Operanden möglicherweise auf Daten im Datensegment der Datei zugreifen. Die Adressen der potentiellen Daten und der Verzweigungen werden in einer Prioritätsliste gespeichert, die aufsteigender nach Adressen sortiert ist. Wenn des Disassembler nun auf eine Instruktion stößt, die den Programmfluss beendet, wird die Analyse am nächsten Eintrag der Liste fortgesetzt. Dies entspricht genau dem Ansatz des *recursive traversal*. Bei Instruktionen, die eine Hochsprachenprozedur aufrufen, wird die Zieladresse als Startadresse einer Prozedur markiert, sodass bei der späteren Analyse dieser Adresse auch das Ende bestimmt werden kann und somit eine Funktion im Code erkannt wird.

Dieser Ansatz würde allerdings nur das Auffinden von Code erlauben, der direkt durch Verzweigungen oder Prozeduraufrufe erreicht werden kann. Es gibt allerdings zwei häufige Fälle, bei denen Code nicht auf diese Weise von anderen Instruktionen erreicht wird, aber über Heuristiken entdeckt werden kann:

- Funktionszeiger: wird in Hochsprachen ein Funktionszeiger in einer Variablen gespeichert und diese später zum Aufrufen dereferenziert, so wird üblicherweise Maschinencode der Form `call <register>` erzeugt, wobei die Adresse der Funktion dann im Register gespeichert ist und somit bei der Analyse nicht bekannt ist. Um diese Funktionen zu finden, werden *nach* der Analyse, d.h. sobald die Prioritätsliste leer ist, alle nicht abgedeckten Bereiche in den Codesegmenten der Datei nach Signaturen durchsucht, die sich üblicherweise zum Einrichten eines Prozedurrahmens am Anfang einer Funktion befinden.

- Sprungtabellen: bei der Verwendung von switch-case-Anweisungen erzeugen Compiler üblicherweise Tabellen zur Verzweigung, sodass in konstanter Zeit die korrekte Sprungmarke erreicht werden kann. Eine Anweisung wie `case 5: p;` könnte beispielsweise dem 6. Eintrag einer solchen Tabelle entsprechen. In der Tabelle ist dann die Adresse von p eingetragen. Solche Tabellen werden auf Assembler-Ebene folgendermaßen umgesetzt: `jmp [tabAdr + 4 * register]`. Dabei ist `tabAdr` die konstante Adresse der Sprungtabelle, die üblicherweise am Ende der Funktion codiert wird. Da die Einträge 32-Bit-Zieladressen enthalten, wird der Index mit 4 multipliziert. Bei 64-Bit-Code würde hier entsprechend eine 8 verwendet. Der Disassembler erkennt dieses Muster. Problematisch ist allerdings, dass die Größe der Tabelle nicht bekannt ist. Der Disassembler liest daher so lange Einträge an `tabAdr`, bis eine Adresse gefunden wird, die außerhalb des virtuellen Speichers liegt (in diesem Fall ist die Tabelle zu Ende und die folgenden Instruktionen wurden als Adresse interpretiert) oder eine Adresse gefunden wird, deren Ziel bereits decodiert ist und die dabei nicht exakt den Anfang der Instruktion trifft. In diesem Fall ergaben die Instruktionen nach der Sprungtabelle zufällig eine gültige Adresse, aber diese zeigte nicht auf den Beginn des Opcodes, sondern auf die folgenden Bytes. Der Fall, dass dabei ein gültiger Start eines Opcodes gefunden wird, ist möglich, aber unwahrscheinlich. Die Einträge der Sprungtabelle werden der Prioritätsliste hinzugefügt und die Analyse fortgesetzt.

Alle gefunden Instruktionen und Daten werden in der Datenstruktur `DisassemblyData` gespeichert. Diese stellt über einen Suchbaum alle Adressen der Programms dar und erlaubt das Annotieren mit Instruktionen, Daten, Kommentaren usw. Die Knoten haben den Typ `DataEntry`. Der Disassembler setzt das *Observer Pattern* passiv um – andere Klassen können sich also als Observer registrieren und werden über alle Änderungen an der Datenstruktur informiert.

4.4 Ruby-Schnittstelle

Die Anbindung an die Skriptsprache Ruby wurde via JRuby durchgeführt. Dadurch ist eine beidseitige Nutzung von Objekten möglich: sobald der Ruby-Code eine Instanz eines Java-Objekts erhält, kann auf alle Methoden, Instanzvariablen usw. zugegriffen werden. Ruby-Objekte können ebenfalls an Java-Methoden übergeben werden und im Java-Code wie native Objekte behandelt werden. Die Umsetzung der Schnittstelle war deshalb sehr einfach – es mussten lediglich einige Methoden als Einstiegspunkte bereitgestellt werden, damit der Benutzer im Skript an Instanzen der internen Datenstrukturen gelangt. Diese Methoden sind in `ScriptAPI` spezifiziert und in `ScriptManager` implementiert. Im Verzeichnis `scripts/` befinden sich beispielhafte Ruby-Skripte, von denen zwei in Anhang A erläutert werden.

5 Diskussion

5.1 Reflexion

Das Projekt konnte erfolgreich umgesetzt werden – es wurden alle Pflichtanforderungen sowie alle optionalen Anforderungen des Lastenhefts erfüllt. Als weiteres Kriterium wurde auf eine große Flexibilität des Codes geachtet, sodass neue Maschinencode-Formate, Prozessor-Architekturen oder Disassembler-Strategien einfach hinzugefügt werden können.

Die Wahl der Programmiersprache Java wurde in Abschnitt 3.3 mit der leichten Anbindung an die Skriptsprache Ruby begründet – dieser Teil des Projekts war dadurch auch tatsächlich am einfachsten. Ein störender Nachteil von Java war, dass für primitive Datentypen keine Alias-Namen angelegt werden können, wie es beispielsweise mit `typedef` in C möglich wäre. Dadurch mussten die Informationen über die Datentypen aus Tabelle 3 auf Seite 13 ständig bedacht werden, es konnte keine Abstrahierung davon stattfinden.

Ein weiterer Nachteil wurde im Vorfeld gar nicht bedacht und trat erst bei der Umsetzung der grafischen Benutzeroberfläche auf: Für Java existiert de facto nur die Swing-Bibliothek zur Umsetzung von grafischen Oberflächen, wenn auf native Bibliotheken verzichtet werden soll. Die Swing-Komponenten zur Darstellung von Text (`JTextPane` bzw. `JEditorPane`) basieren alle darauf, dass der anzuzeigende Text in Objekten vom Typ `Document` enthalten ist. Die vorhandenen Implementierungen dieser abstrakten Klasse erwarten alle, dass der gesamte Text in Form von Strings vorliegt – problematisch daran ist, dass beim Erzeugen des Disassembler-Listings eine sehr große Menge an Text entsteht, die Einfüge-Operationen sehr langsam machen, wenn diese nicht am Ende des Texts geschehen. Da die Analyse aber keinen linearen Lauf durch die Adressen des virtuellen Speichers darstellt, sind die Einfüge-Operationen auf den Text quasi zufällig. Dadurch sinkt die Effizienz sehr stark. Die Umsetzung war ursprünglich so geplant, dass das Listing gar nicht als Ganzes vorliegt, sondern nur die Teile als Text vorliegen, die gerade für den Benutzer sichtbar sind. Dies ist bei der Verwendung der verfügbaren `Document`-Implementierungen aber nicht möglich. Eine eigene Implementierung konnte das Problem ebenfalls nicht lösen: Jede `Document`-Implementierung muss Swing die Möglichkeit bieten, beliebige Stellen im Text zu markieren und später nachzuschlagen, wo sich diese nach Einfüge- oder Lösch-Operationen befinden. Eine Datenstruktur, die dies ermöglichen würde, ist ohne das Vorhalten des gesamten Listings nicht effizient möglich. Ein weiteres Effizienzproblem entsteht dadurch, dass jede Änderung am Text ein Neuzeichnen der Komponente auslöst, selbst wenn nicht-sichtbare Bereiche betroffen sind. Der Grund ist, dass die Größe des Rollbalkens sich ändern muss, wenn Text eingefügt wird. Dieses Problem wird umgangen, indem das Dokument erst angezeigt wird, nachdem die Analyse abgeschlossen ist. Zum Vergleich: Wenn jede analysierte Zeile auf der Konsole ausgegeben wird, dauert die Analyse einer großen Datei weniger als 5 Sekunden. Wird der Text stattdessen in eine `Document`-Instanz geschrieben, steigt die Zeit auf 50 Sekunden. Wenn der Text bereits während der Analyse angezeigt und aktualisiert werden soll, dauert der gesamte Vorgang 300 Sekunden, was außerhalb der zumutbaren Zeit liegt.

5.2 Ausblick

Bei der Implementierung des Projekts wurde an allen Stellen auf Erweiterbarkeit hinsichtlich anderer Architekturen geachtet. Sobald ein Decoder für eine andere Architektur bereitsteht, der aus einem Datenstrom ein einzelnes Opcode in eine Instruktion übersetzt, kann der Disassembler ohne weitere Änderungen am Code darauf aufbauen und die neue Architektur vollständig unterstützen. Hier wäre die Unterstützung von ARM-Architekturen interessant, da viele eingebettete Systeme wie beispielsweise DSL-Router oder Smartphones auf dieser Architektur basieren und es mittlerweile Schadsoftware für die ARM-Architektur gibt (vgl. dazu [Eik14]).

Um die Benutzbarkeit und Erweiterbarkeit zu verbessern, sind noch folgende Erweiterungen denkbar:

- bei der Erkennung von Daten wie Zeichenketten wird bereits angezeigt, an welchen Stellen diese verwendet werden. Es wäre nützlich, diese Adressen für ein Datum auf Wunsch in einem eigenen Fenster anzuzeigen, das die Instruktionen zeigt, die auf die Daten zugreifen. So könnte der Benutzer direkt sehen, welche der Instruktionen das Datum nur ausliest und welche es modifizieren – an dieser Stelle könnte die manuelle Analyse des Codes dann gezielt fortgesetzt werden, statt alle Adressen manuell zu prüfen.
- Java 8 wird mit JavaFX eine Alternative zu Swing bieten – möglicherweise kann das Darstellungsproblem des Assembler-Listings damit gelöst werden. Weiterhin wird es die primitiven Datentypen dann auch in einer Variante ohne Vorzeichen geben, sodass die Verwendung der unterschiedlichen x86-Typen vereinfacht werden könnte.

A Beispiel-Skripte

A.1 Selbstmodifizierender Code

Das folgende Ruby-Skript kann in den Disassembler geladen werden, um in einem Maschinenprogramm nach Stellen zu suchen, an denen der Code sich selbst modifizieren. Diese Technik wird oft zur Verschleierung von Code verwendet.

```
1  # Iterates all instructions and displays write-access to code,
2  # i.e. finds self-modifying code
3
4  $api.traverseCode do |inst|
5    for op in inst.getDestOperands
6      # Check if operand has code address as destination
7      destAddr = op.asNumber
8      if $api.isCodeAddress(destAddr)
9        puts "Instruction at #{inst.getMemAddress.to_s(16)} modifies code at
             #{destAddr.to_s(16)}"
10     end
11   end
12 end
```

Mittels der Anweisung in Zeile 4 werden alle Instruktionen des Programms dem folgenden Ruby-Block übergeben. In diesem Block werden alle Ziel-Operanden darauf überprüft, ob sie als Zahl interpretierbar sind – wenn ein Ziel-Operand eine Zahl enthält, ist dies immer eine Speicher-Adresse. In Zeile 8 wird schließlich geprüft, ob die Ziel-Adresse in einem ausführbaren Bereich des Programms liegt. Wenn dies der Fall ist, wird eine Information darüber ausgegeben. Eine erweiterte Version dieses Skripts befindet sich in der Datei `scripts/selfmod.rb`. Dort wird versucht, die Modifikation durch Auswertung der Anweisung auszuführen, um die Analyse schließlich dort fortzusetzen. Auf diese Weise können Verschleierungen durch selbstmodifizierenden Code analysiert werden.

A.2 Histogramm

Das folgenden Skript erzeugt ein Histogramm der verwendeten Instruktionen eines Programms und gibt dies textuell aus.

```
1  # Calculate a mnemonic histogram
2
3  histogram = Hash.new(0)
4  total = 0
5
6  $api.traverseCode do |inst|
7       mnem = inst.getMnemonic.to_s
8       histogram[mnem] += 1
9       total += 1
10 end
11
12 histogram.sort_by  {|mnem, count| count}.reverse_each do |a|
13   puts "%10s: %5d (%.2f%%)" % [a[0], a[1], a[1] / total.to_f * 100]
14 end
```

In Zeile 3 und 4 werden das Histogramm sowie die Gesamtzahl der Anweisungen initialisiert. In den Zeilen 7 bis 9 werden die Instruktionen anhand der zugehörigen Mnemonics gruppiert gezählt. In den Zeilen 12 bis 14 wird das Histogramm schließlich nach Häufigkeit sortiert ausgegeben.

B Literatur

[App09] Apple. *OS X ABI Mach-O File Format Reference*. 2009.
URL: `https://developer.apple.com/library/mac/documentation/DeveloperTools/Conceptual/MachORuntime/Reference/reference.html`.

[Eik14] Ronald Eikenberg. *Hack gegen AVM-Router auch ohne Fernzugang*. 2014.
URL: `http://heise.de/-2115745`.

[Int14a] Intel. *Intel® 64 and IA-32 Architectures Software Developer's Manual: Basic Architecture*. Bd. 1. Intel, 2014.

[Int14b] Intel. *Intel® 64 and IA-32 Architectures Software Developer's Manual: Instruction Set Reference, A-M*. Bd. 2A. Intel, 2014.

[Int14c] Intel. *Intel® 64 and IA-32 Architectures Software Developer's Manual: Instruction Set Reference, N-Z*. Bd. 2B. Intel, 2014.

[Mau03] Wolfgang Mauerer.
Linux Kernelarchitektur: Konzepte, Strukturen und Algorithmen von Kernel 2.6.
Carl Hanser Verlag, 2003.

[Mic13] Microsoft.
Microsoft Portable Executable and Common Object File Format Specification. 8.3.
Microsoft, 2013.

[SDA02] Benjamin Schwarz, S. Debray und G. Andrews.
„Disassembly of executable code revisited".
In: *Ninth Working Conference on Reverse Engineering, 2002. Proceedings.* 2002, S. 45–54. DOI: `10.1109/WCRE.2002.1173063`.