Universität der Bundeswehr München
Fakultät für Elektrotechnik und Technische Informatik
Wissenschaftliche Einrichtung 6 - Informationstechnik
Professur für Betriebssysteme und Rechnerarchitekturen
Prof. Dr. Harald Görl

Studienarbeit

Das binSpector-Framework zur Visualisierung von aus Objektcode generiertem LLVM-IR-Code und Assembler-Code

Betreuer:	Prof. Dr. Harald Görl
Autor:	Lt Michael Riedel
E-Mail:	michael.riedel@unibw.de
Matrikelnummer:	1100582
abzugeben am:	30.09.2014

Inhaltsverzeichnis

1. Einleitung

Der Mensch ist fähig, komplexe Aufgaben selbstständig umzusetzen. Computer dagegen können dies nicht. Sie sind lediglich in der Lage, eine Abfolge von Befehlen auszuführen. Damit ein von Menschen geschriebener Quellcode auf einem Computer ausgeführt werden kann, muss dieser kompiliert werden. Beim Kompilieren müssen die komplexen Strukturen des Quellcodes analysiert und in einfache Mikrobefehle[1] übersetzt werden. Im Anschluss kann das Programm auf dem Prozessor, für den es kompiliert wurde, ausgeführt werden.

Bei der Entwicklung neuer Software-Programme unterscheidet man zwischen *open-source*- und *closed-source*-Entwicklung. Bei *open-source*-Programmen wird der Quellcode direkt an den Endbenutzer weitergegeben. Dieser kann dann das Programm entweder für seine Computer und Betriebssysteme kompilieren oder es weiterentwickeln/verändern. Bei *closed-source* erhält der Anwender nur kompilierte Binärdateien. Diese können nur auf den jeweiligen Prozessorarchitekturen verwendet werden, für die sie vom Hersteller kompiliert wurden. Der Anwender hat keine Möglichkeit, den Quellcode zu lesen oder zu verändern. Wenn der Anwender jedoch auf ein *closed-source*-Programm und gleichzeitig auf die Wahrung der Sicherheit seines Computers (Vermeidung von Datenlecks oder Schadcode) angewiesen ist, muss er die Binärdatei analysieren können.

Um Programme zu analysieren, deren Quellcode nicht vorliegt, können Disassembler[2] wie IDA Pro oder Hopper verwendet werden. Mit einem Disassembler kann der Programmfluss als Diagramm oder der Objektcode in der Assemblersprache dargestellt werden. Ebenfalls kann der vorliegende Objektcode in eine Pseudosprache übersetzt werden, die der Programmiersprache C ähnelt. Da für die Übersetzung von Objektcode in diese Pseudosprache fest programmierte Annahmen getroffen werden müssen und keine Informationen zur Benennung von Variablen vorliegen, ist das Resultat nur sehr mühsam zu verstehen. Eine Manipulation des Quellcodes oder der Vergleich von zwei Versionen einer Binärdatei[3] ist

[1] Mikrobefehle bezeichnen die kleinste Befehlsgröße, die Prozessoren abarbeiten können. Mikrobefehle unterteilen sich dabei grob in drei Gruppen: Sprungbefehle (bedingte und unbedingte Sprünge), mathematische Funktionen (`add`, `sub`, `mul` etc.) und Lade-/Speicherbefehle.

[2] Ein Disassembler übersetzt die Maschineninstruktionen des Objektcodes in eine Assemblersprache. Diese Übersetzung basiert auf simplen *look-up*-Tabellen der jeweiligen Maschinenarchitektur.

[3] Aktualisiert der Hersteller seine Software, so wird meist auch die Binärdatei mit dem Objektcode verändert. Um die Unterschiede zu erkennen, müssen beide Binärdateien miteinander verglichen werden. Dies ist zum Beispiel dann sinnvoll, wenn überprüft werden soll, ob ein Update eher Sicherheitslücken öffnet, anstatt sie zu schließen.

auf dieser Grundlage nur schwer möglich.

Zum aktuellen Zeitpunkt fehlt ein *open-source*-Disassembler, der dem Benutzer eine komfortablere Darstellung als Assembler oder Pseudocode bietet, um die vorliegenden Maschineninstruktionen zu verstehen. Des Weiteren soll es möglich sein, viele verschiedene Architekturen zu unterstützen und möglicherweise den Objektcode auf diese zu portieren und auf sicherheitsrelevante Eigenschaften zu untersuchen.

In der vorliegenden Studienarbeit bearbeite ich folgende Forschungsfrage: „Wie kann Objektcode in eine Zwischensprache wie Eigenname, ehemals: Low Level Virtual Machine (LLVM)-Intermediate Representation (IR) übersetzt und somit bequem visualisiert und analysiert werden?“ Zur Beantwortung dieser Fragestellung habe ich in Kapitel **??** das `binSpector`-Framework[4] entwickelt. Es bietet eine erweiterbare grafische Oberfläche unter der Verwendung von Qt5, um eine angenehme Analyse von Objektcode und die Verwendung auf unterschiedlichen Betriebssystemen zu ermöglichen und wird. Zur Dekompilierung von Objektcode zu LLVM-IR (eine „Zwischensprache“ zwischen Assembler und höheren Programmiersprachen, wie C) greife ich auf Projekte wie *Dagger*[5] und *Fracture*[6] zurück, die in Kapitel/ref{vom-objektcode-zur-llvm–ir} näher erläutert werden. Zunächst biete ich in Kapitel 2 einen Überblick über das LLVM-Projekt und zeige Möglichkeiten zur sicherheitstechnischen Analyse von Objektcode auf.

Für die Recherche und die Entwicklung des *binSpector*-Frameworks habe ich das Betriebssystem OSX verwendet. Alle in dieser Arbeit enthaltenen Erkenntnisse, Anweisungen und Code-Listings wurden ausschließlich auf einer Intel Core-i7 CPU und der Betriebssystemversion 10.9.5 getestet. Auf weitere benötigte Programme weise ich im Fließtext hin und erläutere notwendige Schritte für die Installation.

Für die Lektüre dieser Studienarbeit werden fortgeschrittene Kenntnisse im Bereich der Programmerzeugung, höheren Programmiersprachen wie C/C++, UNIX und Intel-Assembler vorausgesetzt. Zur Auffrischung werden die Werke von Aho et al. (2007), Bach (1986), Eagle (2008), Katz (2014), Kernighan et al. (1988), Lattner (2007) und Pawelczak (2013b) empfohlen.

[4] Der Name ist abgeleitet von *binary* und *inspector*.

[5] *Dagger* ermöglicht die Umwandlung von Objektcode in die Zwischensprache LLVM-IR, siehe `http://dagger.repzret.org`.

[6] Link zum Repository inklusive Beispielen zur Verwendung: `https://github.com/draperlaboratory/Fracture`.

2. Das Projekt LLVM

Bei LLVM handelt es sich gemäß Lattner (2007) um ein modulares Compiler-Projekt, das seit 2000 unter Chris Lattner und Vikram Adve an der Universität von Illinois entwickelt wird. Das LLVM-Projekt setzt sich aus unterschiedlichen Einzelprojekten zusammen. Die bekanntesten Vertreter[7] sind:

- LLVM Core (Bibliotheken für den Code-Generator und Optimierer)
- Clang (der native LLVM C/C++/Objective-C Compiler/Frontend)
- LLDB (Ein Debugger, der die Bibliotheken von LLVM und Clang verwendet)
- klee (Eine *symbolic virtual machine*, die versucht, alle dynamischen Pfade eines Programms zu analysieren und somit Bugs zu identifizieren)

2.1. Funktionsweise von LLVM - Unabhängigkeit von Quelle und Ziel

Ein großer Vorteil bei der Verwendung von LLVM ist der Aufbau. LLVM kann durch Frontends und Backends individuell erweitert werden. Somit können Entwickler ihre eigenen Sprachdefinitionen schreiben[8] und diese mit den Mitteln von LLVM gemäß Abbildung 2.1 (verschiedene Code- und Laufzeitanalysen sowie Optimierungen) für ihre eigenen oder die bereits vorhandenen Backends kompilieren. Um die Unabhängigkeit von Quelle und Ziel sicherzustellen, wird die LLVM-IR, eine „Zwischensprache“, verwendet.

Der folgende Abschnitt dient zur Veranschaulichung der Projektstruktur von LLVM. Er zeigt die Entwicklung eines einfachen C-Programms und dem Kompiliervorgangs für eine Intel x86-64 Prozessor-Architektur.

2.2. Entwicklung eines Beispiel-Programms und Kompilierung mit LLVM

Im folgenden Abschnitt werden zwei Möglichkeiten zur Kompilierung eines C-Programms mit den LLVM-Tools erläutert. Dieses Beispiel zeigt, wie die Komplexität der verschiedenen

[7] Für eine vollständige Liste aller LLVM-Teilprojekte siehe `http://llvm.org`.

[8] Die Entwicklung eines LLVM-Frontends kann anhand des Tutorials von Segal (2009) durchgeführt werden.

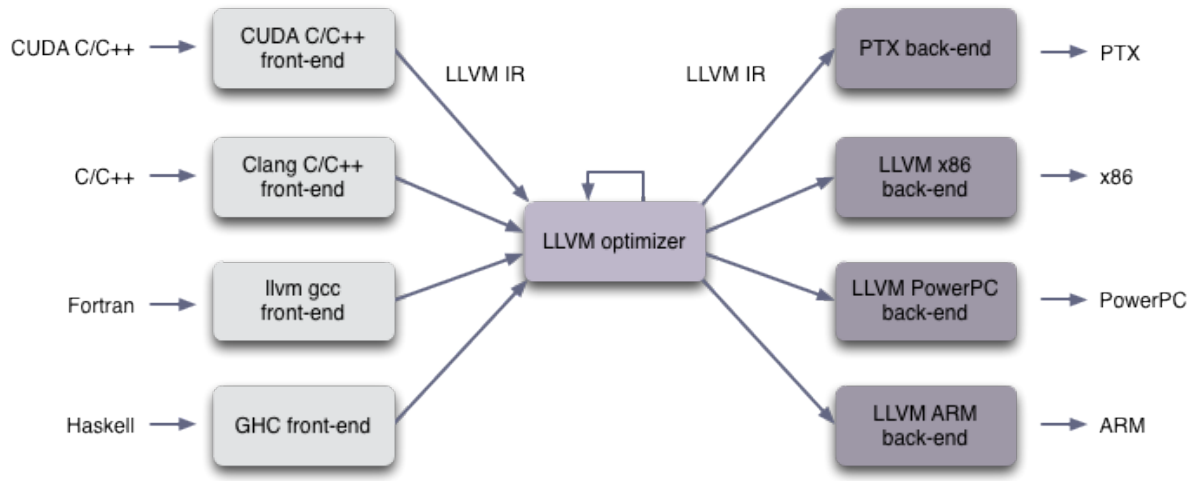


Abbildung 2.1: Übersicht der Funktionsweise von LLVM (QuantAlea GmbH 2014)

Zwischensprachen zunimmt, bis aus einem einfachen Programm aus einer Hochsprache die notwendigen Maschineninstruktionen (Objektcode) entstehen. Das Ziel des Experiments ist, zu zeigen, dass man der Zwischensprache LLVM-IR problemlos eine korrekte ausführbare Objektdatei kompilieren kann.

Aus Gründen der Komplexität wird das folgende Programm sehr einfach gehalten. Durch den Aufruf eines `printf(...)` oder `std::cout`-Befehls würde die kompilierte Objektdatei unnötig aufgebläht, da Bibliotheksfunktionen zusätzlich zum entwickelten Programm geladen, verwendet und kompiliert werden müssten. Als Überprüfung, ob das Programm erfolgreich ausgeführt wurde, wird das Programm nur einen Befehl enthalten, der an die ausführende `Shell` (Kommandozeile) die Zahl 141 zurückgibt.[9] Dieser Rückgabewert kann anschließend durch den Befehl `echo $?` ausgelesen werden.[10]

Zu Beginn wird eine neue C-Datei erstellt, die den zu testenden Programm-Code enthält.

```
1 echo "int main(int argc, char** argv){return 141;}" > main.c
```

[9] Bei der gewählten Zahl handelt es sich um eine zufällig gewählte Zahl. Ihr Wert hat keine Relevanz in Bezug auf die Ausführung des Programms und entspricht ungewollt dem Geburtstag und Geburtsmonat des Autors.

[10] Alle folgenden Codelistings sind zusammenhängend zu betrachten und werden jeweils vor dem Befehl erläutert. Bei den Listings handelt es sich um eingegebene Befehle und Ausgaben, die von aufgerufenen Programmen auf der Standard-Ausgabe (der Kommandozeile) zurückgegeben werden.

2.2.1. Der direkte Weg von C zum Objektcode

Im Anschluss wird die C-Datei mit dem C-Compiler in Objektcode kompiliert.

```
clang main.c -o main.out
```

Nachdem der Übersetzungsvorgang abgeschlossen ist, kann das Programm ausgeführt werden.

```
./main.out
```

Das Programm wird ohne eine Ausgabe auf das Terminal beendet, anschließend kann der Rückgabewert abgefragt werden.

```
echo $?
> 141
```

Das Programm wurde erfolgreich kompiliert, ausgeführt und lieferte das gewünschte Ergebnis. Nun wird das Programm über den Umweg der LLVM-IR kompiliert und überprüft, ob das Resultat das selbe ist.

2.2.2. Der Umweg von C über LLVM-IR und Assembler zu Objektcode

Zu Beginn wird die vorher erstellte C-Datei in LLVM-IR übersetzt.

```
clang -emit-llvm -S main.c -o main.ll
```

Anschließend kann die entstandene LLVM-Datei ausgelesen werden.[11]

```
file main.ll
> main.ll: ASCII text
tail -n+4 main.ll | head -n11
> ; Function Attrs: nounwind ssp uwtable
> define i32 @main(i32, i8**) #0 {
>   %3 = alloca i32, align 4
>   %4 = alloca i32, align 4
>   %5 = alloca i8**, align 8
>   store i32 0, i32* %3
>   store i32 %0, i32* %4, align 4
>   store i8** %1, i8*** %5, align 8
```

[11] Der LLVM-IR Code wurde aus Gründen der Übersichtlichkeit um einige Zeilen gekürzt. Es handelt sich dabei um Metadaten, die LLVM für interne Analysen und Optimierungen verwendet.

```
> ret i32 141
> }
```

Als nächstes wird der LLVM-IR-Code in Assembler kompiliert.

```
llvm-as main.ll -o main.as
file main.as
> main.as: LLVM bit-code object x86_64
```

Da es sich bei der ausgegebenen Datei um Bit-Code handelt, muss dieser noch in eine textuelle Version übersetzt werden.

```
llc main.as -o main.s
```

Die resultierende Datei enthält den Assemblercode für die Intel x86_64-Architektur in Textform und kann ausgelesen werden.[12] Als letzten Schritt muss die Assembler-Datei in eine Objektdatei kompiliert werden.

```
clang main.as -o main.llvm.out
```

Wurde die Kompilierung erfolgreich abgeschlossen, kann das Programm ausgeführt werden.

```
./main.llvm.out
```

Nachdem das Programm beendet wurde, kann der Rückgabewert abgefragt werden.

```
echo $?
> 141
```

Das Beispiel zeigt eine erfolgreiche Übersetzung aus der „Zwischensprache“ LLVM-IR in eine Objektdatei. Die Ausführung zeigt erwartungsgemäß das gleiche Ergebnis, wie die direkt kompilierte Datei. Die Tatsache, dass man aus einer LLVM-Datei stets eine Objektdatei kompilieren kann, ermöglicht die Verwendung von LLVM-IR als Analysegrundlage. In den folgenden Abschnitten werden einige weitere Vorteile von LLVM-IR für die Binärcodeanalyse vorgestellt.

2.3. Vorteile durch die Verwendung von LLVM

In den folgenden Abschnitten werden einige Vorteile erläutert, die durch die Verwendung von LLVM entstehen. Diese Vorteile bilden später die Grundlage für die Binärcodeanalyse mit `binSpector`.

[12] Der Inhalt der Assemblerdatei befindet sich im Anhang als Listing A.2.

2.3.1. Retargetability

Durch die *Retargetability* ist es gemäß Abbildung 2.1 möglich, Quellcode gleichzeitig mit einer konstanten Analyse- und Optimierungsumgebung für unterschiedliche Architekturen zu kompilieren. Ebenfalls können bereits vorhandene Projekte problemlos auf andere Architekturen portiert werden.

2.3.2. Detailliertere Diagnoseausgaben

Das Teilprojekt *Clang* verbessert im Unterschied zur Version 4.2 des GNU C-Compiler (GCC) die Analyse von Syntaxfehlern.[13] Speziell bei der Entwicklung von Projekten sind solche Fehler- und Warnmeldungen von großem Vorteil bei der Lösung von Problemen. Bei einer Umwandlung von einer Programmiersprache in eine andere können solche Ausgaben ebenso hilfreich sein.

2.3.3. Die lesbare Zwischensprache LLVM-IR

Das LLVM-Projekt verwendet für seine verschiedenen Codeanalysen und Optimierungsstrategien bei der Kompilierung von Quellcode die Zwischensprache LLVM-IR. Diese Zwischensprache ähnelt stark der C-Syntax.

```
1  int returnOne(void)
2  { return 1; }
3  int main(int argc, char** argv)
4  { return returnOne(); }
```

```
1  ; Function Attrs: nounwind ssp uwtable
2  define i32 @_Z9returnOnev() #0 {
3    ret i32 1
4  }
5  ; Function Attrs: nounwind ssp uwtable
6  define i32 @main(i32 %argc, i8** %argv) #0 {
7    %1 = alloca i32, align 4
8    %2 = alloca i32, align 4
9    %3 = alloca i8**, align 8
10   store i32 0, i32* %1
11   store i32 %argc, i32* %2, align 4
```

[13] Seit der Version 4.8 des GCC wurden die Diagnosefähigkeiten stark überarbeitet. Trieu (2013) bietet einen Vergleich der Diagnose-Ausgaben zwischen GCC-4.2, GCC-4.8 und Clang.

```
12    store i8** %argv, i8*** %3, align 8
13    %4 = call i32 @_Z9returnOnev()
14    ret i32 %4
15  }
```

```
1   a.out:
2   (__TEXT,__text) section
3   _main:
4   0000000100000f80        pushq       %rbp
5   0000000100000f81        movq        %rsp, %rbp
6   0000000100000f84        movl        $0x0, -0x4(%rbp)
7   0000000100000f8b        movl        %edi, -0x8(%rbp)
8   0000000100000f8e        movq        %rsi, -0x10(%rbp)
9   0000000100000f92        movl        -0x8(%rbp), %eax
10  0000000100000f95        popq        %rbp
11  0000000100000f96        ret
```

Listing 2.1: Vergleich zwischen C++, LLVM-IR und Assembly

Listing 2.1 zeigt anschaulich die Unterschiede zwischen einem sehr einfachen C-Programm (erster Codeabschnitt), der resultierenden und gekürzten LLVM-Syntax (mittlerer Codeabschnitt) und der disassemblierten Objektdatei (letzter Codeabschnitt).

Die LLVM-IR bietet sich sehr gut für die Binärcodeanalyse an, da die notwendigen Informationen wie Funktionsnamen, Rückgabe- und Übergabeparameter und Funktionsinhalte nachvollziehbar dargestellt werden. Der disassemblierte Objektcode bietet einen guten Überblick über die Maschineninstruktionen. Er erlaubt jedoch kaum Rückschlüsse auf den ursprünglichen Quellcode.

2.3.4. Eine Vielzahl an Tools zur Visualisierung

Mit der LLVM-Programmsuite werden verschiedene Tools zur Analyse von Code mitgeliefert. Die nachfolgenden Visualisierungen wurden nach Mikushin (2013) beispielhaft am Codebeispiel 2.1 durchgeführt. Zur Wahrung der Übersichtlichkeit des Dokuments befinden sich die Abbildungen im Anhang B.

Call-Graph

Ein Call-Graph zeigt die Funktionen und ihre Aufrufe untereinander. Dadurch kann der Ablauf eines Programms sehr gut verfolgt werden. Mit dem Befehl gemäß Listing 2.3 kann eine einfache `*.cpp` in die Zwischensprache LLVM-IR übersetzt, optimiert und anschließend

ein Call-Graph in eine `*.dot`-Datei exportiert werden. Diese Datei kann anschließend mit dem `dot`-Programm in ein Bild umgewandelt werden.

```
c++ -emit-llvm -S -c main.cpp -o -| opt -dot-callgraph -o /dev/null \
    && dot -Tpdf callgraph.dot -o callgraph.pdf
```

Listing 2.2: Befehl zur Erstellung eines Call-Graphen

Abbildung B.1 im Anhang zeigt den resultierenden Call-Graphen.

Controlflow-Graph (Basicblocks)

Ein Controlflow-Graph stellt die Anweisungen einer Funktion dar, die nacheinander durchlaufen werden. Dabei ist jeder Knoten über mindestens einen Pfad mit dem Einstiegsknoten verbunden. LLVM verwendet eine Static Single Assignment (SSA)-Darstellung von sogenannten *Basic Blocks*. Ein *Basic Block* ist eine finite Liste von Instruktionen, wobei die letzte stets die terminale Instruktion darstellt (siehe Moll (2011): S. 8). Durch diese Instruktion können verschiedene *Basic Blocks* miteinander verknüpft werden. Der Befehl zur Ausgabe einer Abbildung gemäß B.2 wird im Listing 2.3 aufgezeigt.

```
c++ -emit-llvm -S -c main.cpp -o -| opt -dot-fg -o /dev/null \
    && dot -Tpdf *.dot -o cfg.pdf
```

Listing 2.3: Befehl zur Erstellung eines Controlflow-Graphen

Control- und *Dataflow* Graph

Ein Control- and Dataflow Graph (CFG-DFG) kann dazu verwendet werden, um zu verstehen, welche Variablen auf welche Weise von welchen Funktionen verwendet werden. Dies kann helfen, unbekannte Variablen sinnvoll zu benennen.

```
c++ -emit-llvm -S -c main.cpp -o main.ll && ./graph-llvm-ir main.ll \
    && dot -Tpdf *.dot -o cfg-dfg.pdf
```

Listing 2.4: Befehl zur Erstellung der CFG-DFG

Abbildung B.3 im Anhang zeigt den zugehörigen CFG-DFG. Dieser Graph wurde mit dem Python Skript `graph-llvm-ir` erstellt.[14] Um das Skript verwenden zu können, muss `llvmpy` installiert werden. Dabei handelt es sich um einen LLVM-Wrapper für Python.[15] Listing 2.4 zeigt den Befehl zum Erstellen des Graphen.

[14] Das Skript kann unter `https://github.com/pfalcon/graph-llvm-ir` bezogen werden.
[15] Informationen zur Installation befinden sich unter `http://www.llvmpy.org`.

Zum aktuellen Zeitpunkt ist die letzte von `llvmpy` vollständig unterstützte Version LLVM-3.3. Dies kann in der zukünftigen weiteren Entwicklung von `binSpector` zu Problemen führen, da Programme wie *Fracture*, die für `binSpector` benötigt werden, höhere Versionen von LLVM verwenden. Es kann jedoch sein, dass *Fracture* auch LLVM-3.3 unterstützt. Diesbezüglich wurden noch keine Analysen durchgeführt. Alternativ ist denkbar, die Funktionalität des `graph-llvm-ir`-Skripts in C/C++ zu implementieren, sodass die Abhängigkeit von `llvmpy` gelöst wird.

Memory Dependence Analysen

Gemäß der Dokumentation der Klasse `llvm::MemoryDependenceAnalysis` aus der *Header*-Datei `llvm-3.5/include/llvm/Analysis/MemoryDependenceAnalysis.h` des LLVM-Projekts ist eine *Memory Dependence*-Analyse:

> „[...] an analysis that determines, for a given memory operation, what preceding memory operations it depends on.“

Diese Analyse kann zum Beispiel dazu verwendet werden, um die Funktionen zu identifizieren, die einen bestimmten Speicherbereich manipulieren. Bei dieser Analyse handelt es sich bisher um eine textuelle Auswertung. Es ist denkbar, alle Speicherbereiche, die verwendet werden, darzustellen und dabei die Bereiche, die am häufigsten verändert werden, farbig zu markieren. Dadurch kann ein Bild über relevante Speicherstellen gewonnen werden.

3. Vom Objektcode zur LLVM-IR

Wenn ein Programm nur in Form einer ausführbaren Datei ohne Quellcode vorliegt, kann es nur sehr schwer von Menschen analysiert werden. Analog zum Kompilieren muss das Programm von einer maschinennahen Repräsentation in eine vom Menschen lesbare Version dekompiliert werden. Dieser Prozess ist sehr schwierig, da die maschinennahen Instruktionen sequentiell sind; sie entsprechen einer Art Rezept, das vom Prozessor abgearbeitet wird.

Damit dieser Code in eine strukturierte Version übersetzt werden kann, sind genaue Kenntnisse über die Prozessorarchitektur, den verwendeten Compiler und die Bibliotheken erforderlich. Sind einige dieser Informationen fehlerhaft oder unbekannt, müssen Annahmen getroffen werden. Dadurch kann der übersetzte Code fehlerhaft sein. Ebenfalls wichtig ist, nach welchen Prinzipien die Dekompilierung durchgeführt wird. Einerseits kann der Objektcode von oben nach unten ge-*parsed* (gelesen) werden, andererseits kann der Objektcode ausgeführt und währenddessen analysiert werden.

Im Kapitel 2.3 wurden einige Vorteile erläutert, die die Verwendung von LLVM-IR als „Zwischensprache“ bietet. Im Folgenden werden einige aktuelle Projekte vorgestellt, die sich mit der Dekompilierung von Objektcode in LLVM-IR auseinandersetzen und verschiedene Lösungsansätze bieten.

3.1. Das Projekt *Dagger*

Dagger baut auf der LLVM-Projektstruktur auf. Der Ansatz von *Dagger* entspricht einem *recursive traversal disassembler*. Gemäß Bougacha et al. (2013) entspricht *Dagger* damit der Arbeitsweise des IDA Pro Disassemblers. Ein *recursive traversal disassembler* betrachtet den vorliegenden Binärcode nicht zeilenweise, er analysiert die Maschinenbefehle und führt dabei auftretende Sprungbefehle aus. Ein Sprung kann zum Beispiel an eine Adresse im Speicher verweisen, an der offensichtlich keine Funktion beginnt. Springt nun der Disassembler von *Dagger* an diese Adresse, wird der Binärcode ab dieser Adresse komplett neu analysiert. Ein herkömmlicher Disassembler wie `objdump` würde dabei nur auf einen für ihn fehlerhaften Code stoßen und die Disassemblierung abbrechen.

x86	Mir	IR
`sub ebx, ecx`	`... sub %td2, ...` `put EBX, %td2`	`... %ebx2 = sub i32 ...`

x86	Mir	IR

`add ebx, 12`	`get %td0, EBX`
	`mov %td1, 12`
	`add %td2, %td0, %td1`
	`put EBX, %td2`
		`%ebx3 = add i32 %ebx2, 12`

Tabelle 3.1: Generation von IR Code über die Zwischensprache Mir (Bougacha et al. 2013: S. 43)

Durch *Dagger* ist es möglich, nur die Maschinenanweisungen zu disassemblieren, die bei der Abarbeitung des Objektcodes ausgeführt werden und diese anschließend in LLVM-IR-Code zu dekompilieren. Um dies zu ermöglichen, verwendet *Dagger* die „Zwischensprache“ *Mir*, in der die gesammelten Informationen (Symboltabellen, Sprungtabellen, Registerbenennungen etc.) aufbereitet und verknüpft werden. Gemäß Tabelle 3.1 wird beispielhaft der LLVM-IR-Code aus 2 Assembler-Befehlen rekonstruiert. Dabei werden die Register möglichen Variablen zugeordnet, die durch den Kontrollfluss am wahrscheinlichsten sind.

Durch die Zuweisung von Registern zu Variablen wird der generierte LLVM-Code stark vergrößert. Listing C.4 zeigt das Resultat der Dekompilierung der `main.out`-Datei aus Abschnitt 2.2, das mit dem Programm `llvm-dec` aus dem Projekt *Dagger* generiert wurde. Es ist eindeutig der Mehranteil an Code im Verhältnis zum direkt-kompilierten LLVM-IR aus Listing A.1 zu erkennen.

Die Entwickler von *Dagger* organisieren das Projekt nicht als eigenständiges Projekt von LLVM, sondern integrieren es direkt in den Quellcode. Das hat einen entscheidenden Nachteil: solange *Dagger* nicht erfolgreich in LLVM übernommen wurde, muss der Anwender zwei getrennte Installationen des kompletten LLVM-Projekts ((1) die von *Dagger* unterstützte Version und (2) die aktuelle offizielle Version) bereithalten.

3.1.1. Die Installation von *Dagger*

Da es sich bei *Dagger* um eine Weiterentwicklung des LLVM-Quellcodes handelt, muss die komplette LLVM-Suite heruntergeladen und kompiliert werden. Listing 3.1 zeigt die dazu notwendigen Schritte.

```
1 # Setzen des Installationspfades
2 export DESTINATION=$HOME/Developer
3 # Herunterladen des Quellcodes
4 git clone http://repzret.org/git/dagger.git $DESTINATION/dagger
```

```
# Zum geklonten Verzeichnis wechseln
cd $DESTINATION/dagger
# Erstellen eines build-Verzeichnis
mkdir build && cd build
# Kompilieren von Dagger
cmake .. && make
```

Listing 3.1: Installation von *Dagger*

3.1.2. Kritik an der Verwendung von *Dagger*

Das Projekt *Dagger* hat einige Nachteile in seiner Umsetzung, die seine Verwendung verkomplizieren. Mit dem Installationspaket wird keine Dokumentation mitgeliefert, die erläutert, wie *Dagger* angewendet werden sollte. *Dagger* wurde direkt in die jeweiligen LLVM-Unterprogramme integriert, wodurch zunächst die jeweiligen Programme und die dafür benötigten Übergabeparameter in der Dokumentation der jeweiligen LLVM-Programme gefunden werden müssen. Dies ist ein sehr mühsamer Prozess, der viel Zeit in Anspruch nimmt. Ebenfalls fehlen wissenschaftliche Veröffentlichungen zur Erläuterung der Struktur von *Dagger*.

Bei der Verwendung von *Dagger* sind folgende Schwierigkeiten aufgetreten:

1. Die Dekompilierung schlägt teilweise fehl, weil noch nicht alle Instruktionen, die die x86-Architektur zur Verfügung stellt, in *Dagger* implementiert wurden;
2. Die dekompilierten LLVM-IR-Codes unterscheiden sich grundlegend vom ursprünglichen LLVM-IR-Code. Es war nicht möglich, festzustellen, ob es sich dabei um das gleiche Programm handelt. Die erneute Kompilierung des generierten LLVM-IR-Codes schlug aufgrund fehlerhafter Metainformationen fehl.

Ein weiteres Problem am Projekt *Dagger* ist eine unklare Veröffentlichungsstruktur. Ursprünglich war *Dagger* nicht als *open-source*-Projekt geplant. Auf Drängen der LLVM-Community wurde der Quellcode auf der Webseite des Projekts veröffentlicht. Es handelt sich dabei um ein privates git-Repository (siehe `http://dagger.repzret.org`), das nur stückweise aktualisiert wird. Der aktuelle Fortschritt ist nur schwer nachvollziehbar, sodass *Dagger* auf lange Sicht keine gute Basis für `binSpector` ist.

3.2. Das Projekt *Fracture*

Das Projekt *Fracture* ist ein Architektur-unabhängiger Decompiler von Objektcode zu LLVM-IR-Code. Das Projekt wird aktiv vom *Charles Stark Draper Laboratory*[16] in Cambridge (MA) entwickelt. Zum aktuellen Zeitpunkt können nur Binärdateien der ARM-Architektur von *Fracture* analysiert werden. Eine Unterstützung für die Architekturen x86, PowerPC und MIPS befindet sich momentan in der Entwicklung.

Das Projekt *Fracture* wird über ein `git`-Repository[17] von Draper Laboratories entwickelt und publiziert. Dadurch ist es möglich, an der Entwicklung mitzuwirken und Verbesserungen einzubringen. Der Hauptentwickler ist Richard Carback, der jegliche Anfragen per Mail oder *Github* binnen kürzester Zeit beantwortet und stets mit Rat und Tat zur Seite steht.

3.2.1. Die Installation von *Fracture*

Da es sich bei *Fracture* um ein Projekt handelt, das die Infrastruktur von LLVM verwendet und auf diverse interne Symbole angewiesen ist, muss LLVM mit `--enable-debug-symbols` kompiliert werden. Damit die *Fracture*-Bibliothek kompiliert werden kann, muss ebenfalls eine bestimmte Version von Clang verwendet werden, die von *Draper* gesondert zur Verfügung gestellt wird. Das Listing D.5 zeigt die notwendigen Schritte zur Installation von *Fracture*.

3.2.2. Die grundlegende Verwendung von *Fracture*

Nach der Installation befinden sich im Ordner `Debug+Asserts` von *Fracture* folgende Programme:

- `fracture-cl` (Eine Disassembler Shell)
- `fracture-tblgen` (Ein LLVM-Tabellengenerator, wird von `fracture-cl` verwendet)
- `mkAllInsts` (Erstellt einen Ordner mit einer Objektdatei je unterstützter Instruktion einer übergebenen Architektur)

Das Hauptprogramm ist `fracture-cl` und wird mit der Übergabe einer Objektdatei sowie der Prozessorarchitektur, für die die Objektdatei kompiliert wurde, gestartet. Anschließend kann der Anwender über die Kommandozeile die Binärdatei analysieren. Es ist unter anderem möglich, die verschiedenen Sektionen und Symbole zu extrahieren sowie die Binärdatei zu disassemblieren und in die LLVM-IR zu dekompilieren.

[16] Link zum Webauftritt der Draper-Laboratories: `http://www.draper.com`.
[17] Link zum Repository inklusive Beispielen zur Verwendung: `https://github.com/draperlaboratory/Fracture`.

3.2.3. Aktueller Stand des Projekts

Zum aktuellen Zeitpunkt steht in `fracture-cl` keine Funktion zum Speichern der Ausgabe zur Verfügung. Ebenfalls existiert ein Bug, der die Shell durch die Eingabe von CTRL+D abstürzen lässt. Dadurch ist es nicht möglich, mit mehreren *PIPE*'s die Ein- und Ausgabe durch ein anderes Programm zu steuern.

Trotz der Voraussetzung spezieller Versionen von LLVM und Clang sind die Entwickler stets bemüht, ihren Programmcode kompatibel zu den aktuellsten Entwicklerversionen von LLVM zu halten. Dadurch kann *Fracture* bereits mit der Version 3.5 von LLVM verwendet werden. Bei diesem Projekt mangelt es ebenfalls an einer vollständigen Dokumentation. Diese wird abschnittsweise freigegeben, sobald die jeweilige Fähigkeit vollständig zur Verfügung steht.

3.3. Das Projekt *McSema*

McSema ist der Spitzname für LLVM Machine Code Project (MC) Semantics und ist ein von Defense Advanced Research Projects Agency (DARPA) finanziertes Projekt. Die Entwicklung ist noch sehr jung und wurde auf der diesjährigen REcon[18] (Dinaburg/Ruef 2014) offiziell vorgestellt. Es hat genau wie *Dagger* das Ziel, x86-Binärdateien in LLVM-IR umzuwandeln. Die Entwickler wählen aber eine andere Herangehensweise. *McSema* führt eine statische Übersetzung von x86-Instruktionen in LLVM-IR-Code durch und versucht dabei die Nachteile von *Dagger* und *Fracture* zu vermeiden.

Zur Analyse einer Objektdatei, werden von *McSema* die Instruktionen und der Kontrollfluss unabhängig voneinander betrachtet. Diese Denkweise ist sinnvoll, da bereits bei der Gewinnung des Kontrollflusses viele unbekannte Faktoren eine schwierige Ausgangssituation darstellen. Im folgenden Abschnitt werden einige Probleme und ihre Lösungsansätze erläutert.

3.3.1. Erstellung eines Controlflow-Graphen

Das Erstellen eines Controlflow-Graphen ist je nach vorliegender Objektdatei schwierig. Je nachdem, ob der Entwickler aktiv *Code Obfuscation* („Verstümmelung“ von Code, der eine Dekompilierung erschwert) vorgenommen hat, kann es zu einigen Problemen bei einer reinen *Control Flow Recovery*-Strategie kommen:

- Indirekte Sprungbefehle durch im Programm berechnete Adressen (das Sprungziel

[18] REcon ist eine Computer-Sicherheits-Konferenz, die den Fokus auf *reverse engineering* und fortgeschrittene *Exploit*-Techniken legt. Sie findet jährlich in Montreal, Kanada statt.

ist bei der Dekompilierung unbekannt);
- Sprungtabellen mit zur Ausführung berechneten Offsets;
- Programmcode, der mit Daten vermischt ist;
- Die Bedeutung der vorliegenden Bytes ist unbekannt. Es könnte sich um Konstanten, Daten oder Code handeln.

Um einen Controlflow-Graphen zu extrahieren, verwendet *McSema* den IDA-Pro Disassembler. Für weitere Entwicklungen ist es angedacht, die Graphen durch symbolische Ausführungen zu erhalten; möglicherweise wird dafür das Projekt klee verwendet.

3.3.2. Generierung der LLVM-IR

Sobald der komplette Controlflow-Graph extrahiert wurde, beginnt die Übersetzung zu LLVM-IR. Dabei werden verschiedene Wissensquellen vereint:

- *Mapping* der Instruktionen auf den jeweils aktuellen Kontext des Prozessors;
- Betrachtung der Veränderung des Speichers;
- Übersetzung der einzelnen Funktionen in den aktuellen Kontext (Ersetzen von Registern durch mögliche Variablen);
- Optimierung der extrahierten Funktionen durch automatisierte Compiler;
- Verwendung von externem Wissen wie Windows DLL's zur Gewinnung von möglichen (Funktions-, Variablen-)Namen.

3.3.3. Aktueller Stand des Projekts

Bisher wurden laut Aussagen der Entwickler bereits folgende Fähigkeiten der x86-Architektur implementiert:

- einige Gleitpunkt-Register und -Instruktionen
- alle Integer-Instruktionen
- Unit-Tests
- alle Streaming Single Instruction, Multiple Data (SIMD) Extensions (SSE)-Register
- sehr wenige SSE-Instruktionen
- Callbacks
- einige externe Aufrufe
- Sprungtabellen
- Datenreferenzen

Die Entwicklung an *McSema* ist sehr aktiv, da es sich um ein *open-source*-Projekt auf *Github* handelt.[19] Zum aktuellen Zeitpunkt werden nur die Betriebssysteme Windows und Linux unterstützt. Ob eine Unterstützung für *Mach-O* (OSX) Binärdateien geplant ist, ist nicht bekannt. Auf Grund der noch fehlenden Unterstützung für Mac OSX kann *McSema* in dieser Studienarbeit nicht weiter betrachtet werden.

[19] Link zum Repository: `https://github.com/trailofbits/mcsema`. Link zum Webauftritt: `http://blog.trailofbits.com/?s=mcsema`.

4. Das binSpector-Framework

Das Framework `binSpector` soll eine einheitliche Oberfläche zur grafischen Analyse von Objektcode bieten. Da es sich bei `binSpector` um eine rein grafische Oberfläche handelt, wird in diesem Kapitel die Programmstruktur erläutert und die dafür verwendeten Projekte vorgestellt.

4.1. Die Build-Umgebung mit CMake

Das Build-System der LLVM-Compiler-Infrastruktur ist darauf ausgelegt, schnell und einfach die Entwicklung von Drittprojekten zu ermöglichen, die LLVM-Header oder Subprojekte verwenden. Für die Erstellung eines Projekts, das die LLVM-Compiler-Infrastruktur verwendet, soll gemäß den Codinganweisungen und Anleitungen der LLVM-Webseite[20] eine `CMake`-Build-Umgebung verwendet werden.

`CMake` ist eine *out-of-source*-Build-Umgebung[21], die erweiterbar und plattformunabhängig ist. Unter *out-of-source* versteht man eine Ordnerstruktur, bei der die entstehenden Objektdateien nicht mit den Quelldateien vermischt abgelegt werden.

4.1.1. Die Ordnerstruktur eines CMake-Projekts

Die Ordnerstruktur besteht gemäß Abbildung 4.1 aus den Ordnern `build`, `docs`, `include`, `lib` und `tools`. Die Datei `CMakeLists.txt` beinhaltet die Angaben, wie der vorliegende Quellcode mit `CMake` verarbeitet werden soll.

Der build-Ordner

Dieser Ordner beinhaltet nach der Abarbeitung mit `CMake` die entstandenen Binärdateien sowie eventuell erstellte Bibliotheken (zum Beispiel *shared libraries* oder *static libraries*). Wird der Quellcode eines Projekts verteilt verwaltet, zum Beispiel mit *git*, dann existiert

[20] Für weitere Informationen zur Erstellung eines LLVM-Projekts siehe: `http://llvm.org/docs/Projects.html`.

[21] Eine Build-Umgebung beschreibt die Struktur von Dateien und Ordnern sowie den Ablauf, wie die Quelldateien miteinander verknüpft werden, um zum Schluss ein ausführbares Programm zu ermöglichen. Wird der Build (das Kompilieren und Linken von Quellcode) durchgeführt, werden alle Zwischenprodukte wie temporäre Dateien und Nebenprodukte (zum Beispiel eine *static* oder *shared library*) entsprechend der Ordnerstruktur abgelegt.

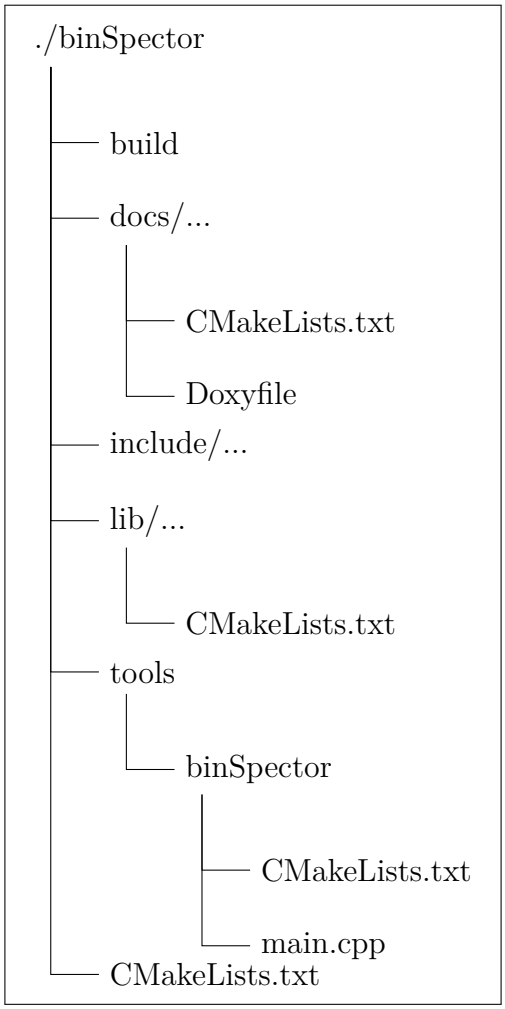


Abbildung 4.1: Ordnerstruktur der CMake-Build-Umgebung

dieser Ordner beim Klonen[22] noch nicht. Der Ordner muss vor dem Kompilieren vom Anwender erstellt werden. Dadurch wird eine unnötige Verwaltung von möglicherweise nicht auf dem Zielsystem laufender Binärdateien verhindert.

Der docs-Ordner

In diesem Ordner befindet sich eine `Doxyfile` und eine `CMakeLists.txt`. Die `CMakeLists.txt`-Datei enthält Anweisungen zur Erstellung der Dokumentation von `binSpector`. Zur Erstellung wird das Programm `doxygen` und die Datei `Doxyfile` verwendet. Letztere enthält Informationen über den Inhalt, das Aussehen der Dokumentation und die Struktur des vorliegenden Quellcodes.

[22] Klonen bezeichnet das Beziehen des Quellcodes aus einem Repository.

Der include-Ordner

Der `include`-Ordner enthält alle notwendigen *Header*-Dateien, die für ein Projekt angelegt wurden. Die *Header* können wiederum in weiteren Unterordnern organisiert werden.

Der lib-Ordner

Der `lib`-Ordner enthält alle notwendigen Quellcode-Dateien. Diese Dateien werden beim Kompilieren meist in Bibliotheken miteinander verknüpft und können dann bei der Erstellung der Binärdateien zum Linken verwendet werden. Die Quellcodedateien können ebenfalls in weiteren Unterordnern organisiert werden.

Der tools-Ordner

Dieser Ordner beinhaltet alle Quellcode-Dateien, die eine `main(...)` -Funktion enthalten. Diese Dateien repräsentieren die späteren Programme, die auf Funktionalitäten aus den `include`- und `lib`-Ordnern zurückgreifen. Dabei sollte für jede spätere Binärdatei ein Unterordner angelegt werden, in dem sich die entsprechende Quellcode-Datei befindet.

4.1.2. Der Ablauf eines Build-Prozesses unter CMake

Mit `CMake` selbst können keine Binärdateien erzeugt werden. `CMake` dient vielmehr dazu, Quellcode-Dateien für die unterschiedlichsten Verwendungen vorzubereiten: es können neben `Makefiles` auch *XCode*-, *Eclipse*- oder gar *Visual Studio*-Projekte erzeugt werden.

Für die Entwicklung von `binSpector` wurde auf eine Entwicklungsumgebung wie XCode oder Eclipse verzichtet, sodass diese Fähigkeit von `CMake` nicht weiter beleuchtet wird. Im Folgenden wird der Ablauf eines Build-Prozesses durch erzeugte *Makefiles* näher erläutert.

Vorbereitung der Build-Umgebung

Damit `CMake` verwendet werden kann, muss die Datei `CMakeLists.txt` angepasst werden.[23] Listing 4.1 zeigt die globale `CMakeLists.txt`-Datei für das `binSpector`-Framework.[24]

```
1 PROJECT(binSpector)
2 CMAKE_MINIMUM_REQUIRED(VERSION 2.8.7)
3 SET( CMAKE_COLOR_MAKEFILE ON )
4 SET( CMAKE_VERBOSE_MAKEFILE ON )
```

[23] Ein zuverlässiges und hilfreiches Tutorial zur Erstellung einer `CMake`-Struktur und der notwendigen `CMakeLists.txt`-Dateien findet man bei Kiefer (2013).

[24] Weiterführend befinden sich Listings der `CMakeLists.txt` der Unterordner `docs`, `lib` und `tools` im Anhang E.

```
5  SET( CMAKE_INCLUDE_CURRENT_DIR TRUE )
6
7  IF( APPLE )
8          SET( PROGNAME binSpector )
9          SET( MACOSX_BUNDLE_ICON_FILE binSpector.icns )
10         SET( MACOSX_BUNDLE_SHORT_VERSION_STRING 0.1-alpha )
11     SET( MACOSX_BUNDLE_VERSION 0.1-alpha )
12     SET( MACOSX_BUNDLE_LONG_VERSION_STRING Version 0.1-alpha )
13 ELSE( APPLE )
14         SET( PROGNAME binSpector )
15 ENDIF( APPLE )
16
17 # Instruct CMake to run moc automatically when needed.
18 SET(CMAKE_AUTOMOC ON)
19 # Find includes in corresponding build directories
20 SET(CMAKE_INCLUDE_CURRENT_DIR ON)
21 # Find the QtWidgets library
22 FIND_PACKAGE(Qt5Core REQUIRED)
23
24 INCLUDE_DIRECTORIES("include")
25
26 ADD_SUBDIRECTORY(docs)
27 ADD_SUBDIRECTORY(lib)
28 ADD_SUBDIRECTORY(tools/binSpector)
```

Listing 4.1: Die Datei `binSpector/CMakeLists.txt`

Eine `CMake`-Datei besteht aus mehreren `Key(Value)`-Tupeln. Der `Value` kann dabei aus mehreren Zeichenketten - getrennt durch ein Leerzeichen - bestehen. Die globale `CMake`-Datei des `binSpector`-Frameworks kann dabei in 5 Abschnitte unterteilt werden:

1. Die Projektbezeichung und Anforderungen an `CMake`;
2. Betriebssystem-spezifische Angaben (hier die Angaben, die OSX benötigt, um die entstehende App korrekt zu erstellen);
3. Angaben zu weiteren Bibliotheken (hier die Angaben, die für die Verwendung von Qt5 mit `CMake` benötigt werden);
4. Die Angabe, in der sich die *Header*-Dateien befinden (dies ermöglicht einen einfachen `#include` in den Quellcode-Dateien, da der Suchbereich `INCLUDE_DIRECTORIES(...)` auf den Ordner „include“ gesetzt wird.);

5. Angaben zu Ordnern, in denen sich weitere `CMakeLists.txt`-Dateien befinden (diese werden beim Aufruf von `CMake` abgearbeitet).

Durch `binSpector/docs/CMakeLists.txt` wird gemäß Abschnitt 4.1.1 und Listing E.6 die Dokumentation vorbereitet. Die Datei `binSpector/lib/CMakeLists.txt` erstellt gemäß Listing E.7 aus den angegebenen `*.cpp`-Quellcode-Dateien eine Bibliothek. Bei der Erstellung dieser Bibliothek wird ebenfalls die `Qt5_Widgets`-Bibliothek verwendet, um die jeweiligen Abhängigkeiten aufzulösen.

Bei der Erstellung des `binSpector`-Frameworks wird die Bibliothek `binSpectorLib` im Anschluss von der Datei `binSpector/tools/binSpector/CMakeLists.txt` gemäß Listing E.8 zum Linken verwendet. Damit das Framework erfolgreich kompiliert werden kann, müssen die *Header*- und *Source*-Dateien durch das Qt5-Programm Meta-Object-Compiler (moc) vorbereitet werden. Das Programm moc „wertet die Deklaration von slots und signals in Header-Dateien aus und erzeugt ein zusätzliches .cpp-Modul, welches die notwendige Routing-Funktionalität enthält“ (Pawelczak 2013a: S. 87). Sobald die `CMakeLists.txt`-Dateien erstellt sind, kann der eigentliche Build-Prozess gestartet werden.

4.1.3. Kompilieren des Frameworks

Um das Framework zu kompilieren, sind die Schritte gemäß Listing 4.2 notwendig. Nach dem Kompilieren des Framework kann binSpector wie im nachfolgenden Kapitel 4.2 erläutert, verwendet werden.

```
1  # Setzen des Installations-Pfades
2  export DESTINATION=$HOME/Developer
3  # Klonen des Repository
4  git clone git@github.com:gismo141/binSpector $DESTINATION/binSpector
5  # Zum geklonten Verzeichnis wechseln
6  cd $DESTINATION/binSpector
7  # Das build-Verzeichnis erstellen
8  mkdir build && cd build
9  # Das build-Verzeichnis vorbereiten (moc, library, etc.)
10 cmake ..
11 # Die Dokumentation, die Bibliothek und binSpector kompilieren
12 make
```

Listing 4.2: Befehle zum Kompilieren von binSpector

4.2. Die Verwendung von binSpector

Das Programm `binSpector` ist in C++ unter Verwendung der grafischen Bibliothek Qt5 geschrieben. Qt ist eine *open-source*-Bibliothek, die für die Betriebssysteme OSX, iOS, Windows, Linux und Android angepasste grafische Elemente bietet. Dadurch können grafische Oberflächen unabhängig vom späteren Betriebssystem programmiert werden. Nach dem Kompilieren entspricht die grafische Oberfläche den Systemstandards des Betriebssystems, auf dem das Programm ausgeführt wird.

Die Programmiersprache C++ eignet sich auf Grund ihrer objekt-orientierten Programmierung sehr gut, um `binSpector` modular zu entwickeln und einfach zu erweitern.

4.2.1. Der grafische Aufbau von binSpector

Wenn `binSpector` gestartet wird, öffnet sich im Vordergrund gemäß Abbildung 4.2 ein `QFileDialog`. Dieser Dialog dient dazu, eine Binärdatei auszuwählen, die von `binSpector` im weiteren Verlauf analysiert werden soll. Der Anwender kann ebenfalls eine abgespeicherte Analyse öffnen. Abgespeicherte Analysen werden von `binSpector` stets als Ordner mit der Endung `*.binsp` (`binSpector`-Projekt) abgespeichert. Hat der Anwender eine Datei ausgewählt, wird das Framework mit seiner Hauptansicht geöffnet.

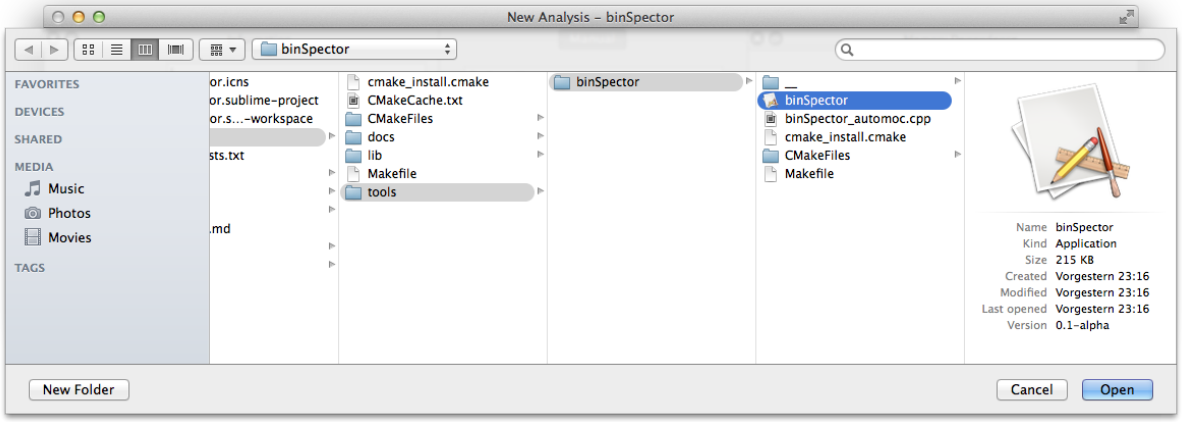


Abbildung 4.2: Dialog „Öffnen eines Projekts/Binärdatei“

Die Hauptansicht besteht gemäß Abbildung 4.3 aus einem zentralen Hauptfenster, einer Menüleiste und jeweils einem Dock an der linken und rechten Seite des Hauptfensters. Diese Docks können in der Größe verändert und getrennt bewegt oder geschlossen werden. Das linke Dock wird im Folgenden als *binary*-Dock bezeichnet; das rechte Dock ist das *visualizer*-Dock. Beide Docks bieten verschiedene Funktionen, die in Tabs gestaffelt werden können.

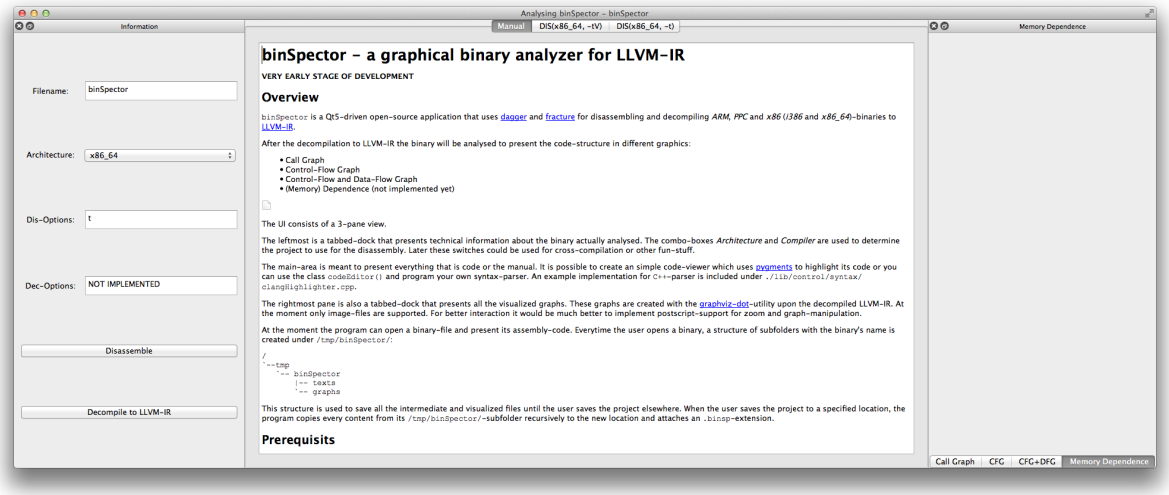


Abbildung 4.3: Hauptansicht von binSpector

Die Menüleiste

Bei OSX wird die Menüleiste stets getrennt vom Programm am oberen Bildschirmrand angezeigt. Über die Menüleiste kann der Benutzer Analyse-Projekte anlegen, speichern, laden und schließen. Ebenfalls können verschiedene Einstellungen bezüglich des Objektcodes vorgenommen werden. Es wird empfohlen, jeden Menüeintrag mit einem Tastaturkürzel zu versehen.

Als Menüleiste wird eine `QMenuBar` verwendet, die bereits folgende `QMenu`'s zur Verfügung stellt:

- **File** (Funktionen zum Datei- und Projektmanagement)
- **Edit** (Bearbeiten von Analysen)
- **View** (Darstellung von `binSpector` verändern)
- **Help** (Hilfestellungen zur Verwendung von `binSpector`)

Das zentrale Hauptfenster

Das Hauptfenster bietet Platz für die aktuell durchgeführte Analyse. Die Standarddarstellung ist der aktuell betrachtete Objektcode in Assembler. Wurde im *binary*-Dock die Architektur des vorliegenden Objektcodes angegeben oder durch *binSpector* korrekt bestimmt, so kann die resultierende LLVM-IR im Hauptfenster dargestellt werden. Durch den Aufruf der Hilfe wird eine Anleitung zur Verwendung von `binSpector` im Hauptfenster dargestellt.

Das *binary*-Dock

In diesem Dock werden gemäß Abbildung 4.3 alle Funktionen aufgelistet, die mit der Objektdatei an sich in Verbindung gebracht werden können. Beispielhaft sind Ausgabefenster, die den Inhalt des `Comment`-Blocks anzeigen oder Einstellfenster, um die Architektur anzugeben, für die der Objektcode kompiliert wurde. Die in diesem Fenster dargestellten Informationen sind nicht generiert, sondern werden aus der vorhandenen Binärdatei extrahiert.

Das *visualizer*-Dock

Das *visualizer*-Dock ermöglicht gemäß Abbildung 4.3 die grafische Darstellung des analysierten Objektcodes. Zum aktuellen Zeitpunkt wird bei der Analyse ein Kontrollflussgraph angezeigt. Ein Kontrollflussgraph stellt den Ablauf des Objektcodes grafisch dar. Bei der Weiterentwicklung des Programms ist es denkbar, die Graphen mit unterschiedlich vielen Details anzuzeigen. Die Detailtiefe wird vom Benutzer definiert, der dadurch einen besseren Gesamtüberblick über das analysierte Programm erhält.

4.2.2. Disassemblieren einer Binärdatei

Um eine Binärdatei zu disassemblieren, muss in `binSpector` eine Binärdatei geöffnet sein. Diese kann entweder beim Start von `binSpector` oder über den Menüeintrag *File->New Binary-Analysis* oder das Tastenkürzel `CMD+N` ausgewählt werden. Es ist möglich, `*.app` -Bundles zu öffnen. Dabei handelt es sich um eine Ordnerstruktur, die unter Mac OSX verwendet wird, um Anwendungen mit zusätzlichen Dateien zu bestücken. Wurde eine Datei ausgewählt, überprüft `binSpector`, ob es sich um eine `*.app`-Datei handelt - wenn ja, wird die darin enthaltene Binärdatei unter `<Name-des-Programms>.app/Contents/MacOS/<Name-des-Programs>` geladen.

Anschließend wird im *binary*-Dock der Programmname sowie die Architektur angezeigt, für die sie kompiliert wurde. Diese Informationen werden aus dem UNIX-Tool namens `file` extrahiert. Momentan wird stets x86_64 bevorzugt, wenn es sich um eine sogenannte *fat*-Binärdatei handelt. Eine *fat*-Binärdatei enthält mehrere kompilierte Versionen des Programmcodes, um verschiedene Architekturen zu unterstützen. Die automatische Auswahl kann vom Anwender stets überschrieben werden; dafür muss aus der `QComboBox` nur die gewünschte Architektur ausgewählt werden.

Im Anschluss können die Parameter angegeben werden, die zur Disassemblierung verwendet werden sollen. Für die Disassemblierung wird das OSX-Programm `otool` verwendet. Die Parameter lauten standardmäßig `tV` und produzieren eine disassemblierte Ausgabe von Textstellen und symbolisch disassemblierten Operanden aus der Text-Sektion der

Binärdatei.[25]

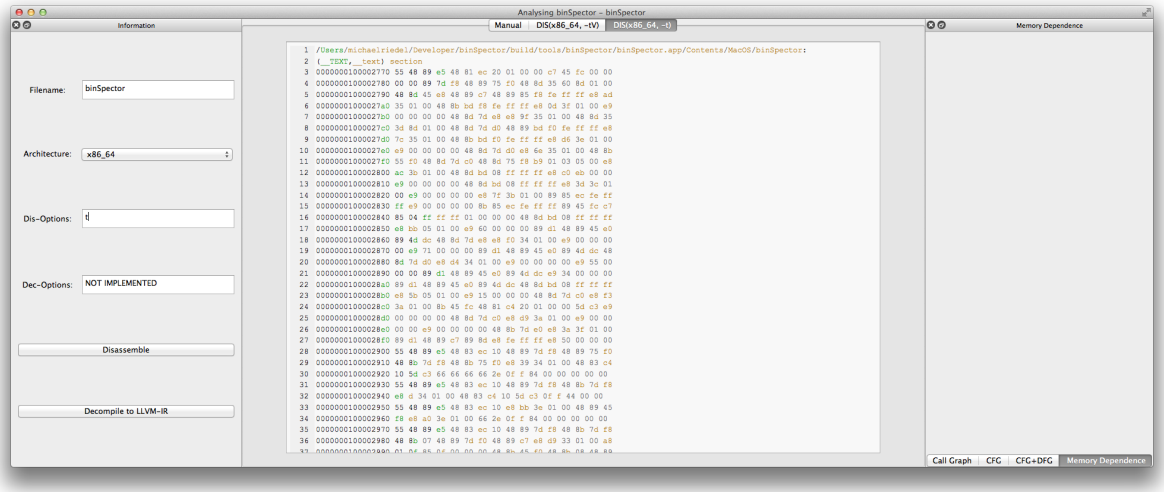


Abbildung 4.4: Disassemblierung der Anwendung `binSpector` mit den Optionen `t`

Durch einen Klick auf den `QPushButton` „Disassemble“ wird die Disassemblierung mit den angegebenen „Dis-Options“ und der ausgewählten Architektur gestartet. Je nach Größe der Binärdatei dauert die Disassemblierung und Darstellung mehrere Sekunden, in diesem Zeitraum bleibt der `QPushButton` „Disassemble“ blau hinterlegt. Sobald die Disassemblierung abgeschlossen ist, wird der Assembler-Code mit dem GNU is not UNIX (UNIX-ähnliches Betriebssystem) (GNU)-Programm `c++filt` analysiert, um die Funktionsnamen gemäß der *Name-Mangling*-Konvention aufzulösen und die Zeilen nach 110 Zeichen mit dem UNIX-Programm `fold` umzubrechen. Anschließend wird das Python-Skript *Pygments* verwendet, um den Assembler-Code farbig zu markieren und als `*.html`-Datei zu speichern.[26] Diese wird dann gemäß Abbildung 4.4 im Hauptfenster dargestellt.

4.2.3. Dekompilieren zu LLVM-IR

Diese Funktionalität ist momentan nicht implementiert, weil die Projekte *Fracture* und *Dagger* bisher keine verifizierbare Ausgabe liefern.

Angedacht ist, dass das Dekompilieren nahezu dem Ablauf zum Disassemblieren entspricht. Nachdem die Binärdatei geladen wurde, muss der Anwender die Optionen zum Dekompilieren im `QPlainTextEdit`-Feld „Dec-Options“ angeben und die Architektur auswählen. Durch den Klick auf den `QPushButton` „Decompile to LLVM-IR“ wird die Dekom-

[25] Weitere Optionen sind in den `manpages` von `otool` aufgelistet.

[26] Durch das kontinuierliche Speichern der Zwischendateien wird dem Verlust einer Analyse vorgebeugt. Sollte das Programm `binSpector` abstürzen, können die bereits analysierten Dateien aus dem Ordner `/tmp/binSpector/<Name-des-analysierten-Programms>/` extrahiert werden.

pilierung gestartet.

Zum aktuellen Zeitpunkt wird statt der Dekompilierung eine Disassemblierung durchgeführt. Für die erzeugte Ausgabe wird gemäß Abbildung 4.5 die Syntax-Färbung für LLVM-IR verwendet.

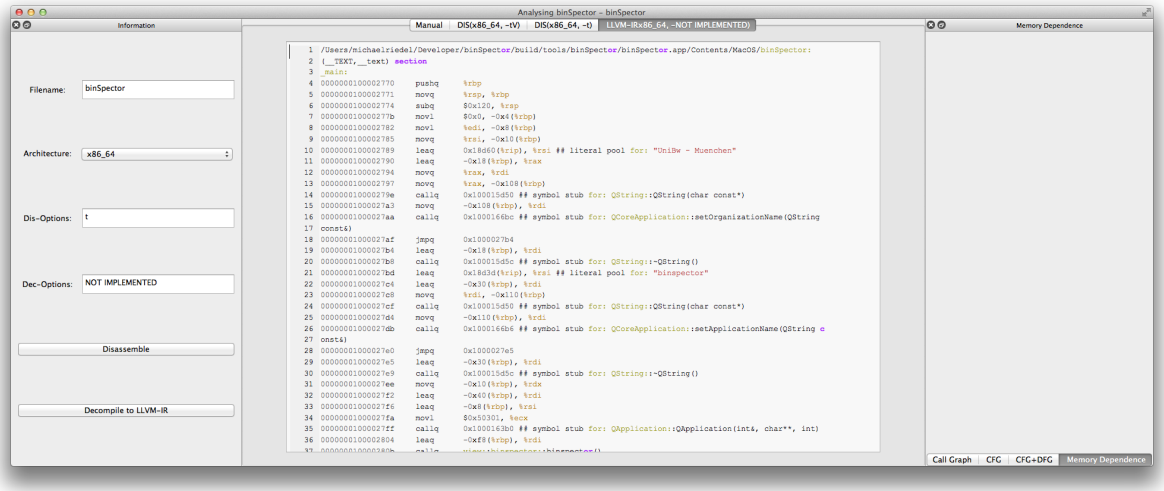


Abbildung 4.5: Disassemblierung der Anwendung `binSpector` mit der Syntax-Färbung für LLVM-IR

Es ist angedacht, dass die Dekompilierung ähnlich wie die Disassemblierung durch ein externes Programm durchgeführt wird. Die Bedingungen dafür sind, dass jede Datei, die entsteht, im Ordner `/texts/` mit einem eindeutigen Namen abgespeichert wird. Ein eindeutiger Dateiname muss sich wie folgt zusammensetzen: `<verwendete-Architektur>+<verwendete-Optionen>+.+<Dateiendung-des-Resultats` (zum Beispiel: `x86_64.tV.asm`).

4.2.4. Abspeichern und erneutes Öffnen eines Projekts

Ein analysiertes Projekt wird temporär unter `/tmp/binSpector/<Name-des-analysierten-Programms>` abgelegt. Durch den Menüeintrag *File->Save Project* (`CMD+S`) kann die aktuelle Analyse an einem gewünschten Ort abgespeichert werden. Dabei werden alle Analysedaten aus dem temporären Ordner in den Zielordner kopiert.

Durch *File->Open Project* (`CMD+O`) kann anschließend der abgespeicherte `*.binsp`-Ordner ausgewählt und wieder geöffnet werden.

4.3. Die programmiertechnische Struktur von binSpector

`binSpector` ist als modulares Framework aufgebaut, damit es kontinuierlich im Forschungsbereich der wissenschaftlichen Einrichtung für Betriebssysteme und Rechnerarchitekturen (WE 6) der Universität der Bundeswehr in München weiterentwickelt werden kann. Um

den zukünftigen Nutzern die Entwicklung und Benutzung von `binSpector` zu erleichtern, werden in den folgenden Abschnitten die Struktur des Frameworks und der verwendeten Tools und Projekte näher erläutert.

4.3.1. Die Aufteilung der Namespaces

Das Framework wird gemäß dem *Model-View-Controller*-Pattern entwickelt. Dies ermöglicht eine Aufteilung in Projekte (*Control*), in Daten (*Model*) und in grafische Oberflächen (*View*), um mit den Objektdateien zu interagieren. Um das Framework thematisch zu gliedern, werden verschiedene Namespaces verwendet. Diese Namespaces werden auch im Dateisystem durch die Verwendung von Ordnern repräsentiert, wodurch eine homogene Struktur möglich wird.

4.3.2. Ablauf des Programmstarts

Zum Programmstart wird ein Objekt der `binSpector`-Klasse gemäß Listing 4.3 initialisiert und anschließend dargestellt.

```
1  #include <iostream>
2
3  #include <QApplication>
4  #include "view/binspector.h"
5
6  int main(int argc, char **argv)
7  {
8      QApplication::setOrganizationName("UniBw - Muenchen");
9      QApplication::setApplicationName("binspector");
10     QApplication app(argc, argv);
11
12     view::binspector mainWindow;
13     mainWindow.show();
14     return app.exec();
15 }
```

Listing 4.3: Die `main()`-Funktion von `binSpector`

Im Konstruktor der `view::binspector`-Klasse wird anschließend gemäß Abbildung 4.3 die Hauptansicht mit den Menüeinträgen, der Hilfe zu `binSpector` sowie den *visualizer*- und *binary*-Docks initialisiert und dargestellt.

Das *visualizer*-Dock wird mit 4 Visualisierungen als Tabs initialisiert:

- der Call-Graph (`view::visualizer::callGraph`)
- der Controlflow-Graph (`view::visualizer::controlFlowGraph`)
- der CFG-DFG (`view::visualizer::controlFlowAndDataFlowGraph`)
- Memory-Dependence (`view::visualizer::memoryDependence`)

Das *binary*-Dock wird mit dem `view::binary::basicInfo`-Tab geladen. Dieser Tab ermöglicht das Disassemblieren und Dekompilieren. Bei der Initialisierung des `view::binary::basicInfo`-Tabs wird der *codeViewer* geladen, der wiederum den `control::Disassembler` und `control::Decompiler` instantiiert. Ab diesem Moment ist `binSpector` einsatzbereit.

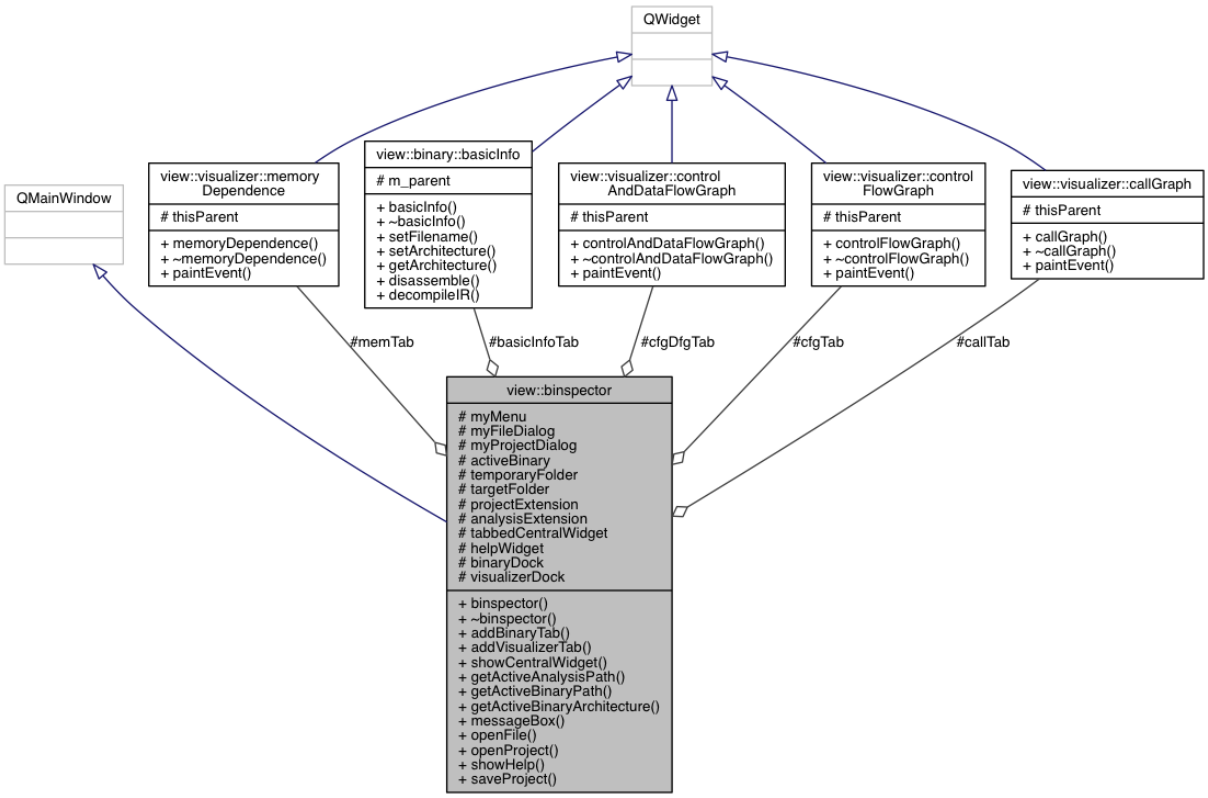


Abbildung 4.6: Kollaborationsdiagramm zur `view::binspector`-Klasse

Abbildung 4.6 zeigt das Kollaborationsdiagramm der `view::binspector`-Klasse. Anhand der *Doxygen*-Dokumentation von `binSpector` können die aufgerufenen Funktionen und Parameter analysiert und visualisiert werden.

4.3.3. Der Entwicklungsstand von binSpector

In der vorliegenden Version ist `binSpector` in der Lage, eine gewünschte Binärdatei mit den vom Anwender angegebenen Optionen zu disassemblieren und das Ergebnis farbig darzustellen. Analysen lassen sich als Projekte abspeichern und können wieder geladen werden.

Bisher konnten die vorgestellten Projekte *Fracture* oder *Dagger* noch nicht erfolgreich in das `binSpector`-Framework implementiert werden. Auf Grund dessen sind die Visualisierungen ebenfalls ohne Funktion, da zur Erstellung der Graphen eine LLVM-IR-Datei benötigt wird.

5. Fazit und empfohlene Weiterentwicklung am binSpector-Framework

Die Präsenz von Technik, die uns umgibt, hat im letzten Jahrzehnt stark zugenommen. Stets werden neue Apps auf unseren Mobilgeräten veröffentlicht, aktualisiert oder durch andere Produkte ausgetauscht. Im Computerbereich werden teilweise bereits im Jahrestakt neue Betriebssysteme veröffentlicht. Bei so schnellen Veränderungen ist es kaum möglich, alle Sicherheitslücken eines Systems zu schließen, ohne bei Aktualisierungen weitere zu öffnen.

Da viele Softwarelösungen als *closed-source*-Produkte veröffentlicht werden, ist es schwer möglich, Sicherheitslücken zu identifizieren, geschweige denn, sie zu schließen. Eine mögliche Lösung ist das Disassemblieren solcher Anwendungen. Auf Grund der maschinennahen Struktur des Assembler-Codes ist eine Analyse sehr mühsam und erfordert viele Kenntnisse über die vorliegende Prozessorarchitektur.

Das LLVM-Projekt ist eine vielseitige Plattform im *open-source*-Bereich, die es ermöglicht, verschiedene Compiler und Prozessorarchitekturen auf der gemeinsamen „Zwischensprache“ LLVM-IR zu vereinen. Die Syntax von LLVM-IR ähnelt stark der C-Syntax und ermöglicht damit ein besseres Verständnis als Assembler. Das Ziel ist es, einen Decompiler zu entwickeln, der Maschineninstruktionen in LLVM-IR zuverlässig umwandelt.

Seit zirka 2 Jahren ist ein wachsendes Interesse im Bereich der Dekompilierung zu vermerken. In der vorliegenden Studienarbeit wurden 3 Projekte vorgestellt, die explizit in diesem Bereich forschen. Insbesondere die Rüstungsindustrie hat großes Interesse an Weiterentwicklungen und finanziert entsprechende Projekte. Zum Beispiel wird das Projekt *McSema* von DARPA finanziert, *Fracture* wird an den *Draper Laboratories* für das amerikanische Verteidigungsministerium entwickelt.

Das `binSpector`-Framework ist meine Antwort auf die eingangs gestellte Forschungsfrage: „Wie kann Objektcode in eine Zwischensprache wie LLVM-IR übersetzt und somit bequem visualisiert und analysiert werden?“ Das `binSpector`-Framework bietet die Möglichkeit, verschiedene Projekte zur Dekompilierung zu verwenden, zu vergleichen und zu visualisieren. Das Projekt `binSpector` ist im Bereich der visuellen Analyse von LLVM-IR-Code einzigartig, weil bisher kein *open-source* Projekt diese Nische als Framework abdeckt.

Das Projekt ist zukunftsträchtig, weil zusammen mit dem LLVM-Projekt eine mächtige Plattform geboten wird, die für viele Betriebssysteme und Prozessorarchitekturen die

notwendige Unterstützung bietet.

Die zukünftige Entwicklung von `binSpector` sollte im Sinne der Integration der Projekte wie *Dagger* und *Fracture* fortgeführt werden. Im aktuellen Stadium ist `binSpector` eine visuell ansprechende und leicht zu verstehende Anwendung. Die Einfachheit der Anwendung sollte auch weiterhin einen wichtigen Stellenwert einnehmen. Sobald die korrekte Umwandlung von Maschinencode in LLVM-IR möglich ist, sind kreativen Weiterentwicklungen keine Grenzen gesetzt.

Zukünftige Projekte bieten sich im Bereich der Visualisierung von LLVM-IR-Code sowie im Bereich der Portierung von Binärdateien auf unterschiedliche Architekturen an. Ebenfalls ist es denkbar, aus dem generierten LLVM-IR-Code eine korrekte Dekompilierung in C/C++/Objective-C zu erhalten.

6. Glossar

Bytecode Quellcode, der in eine maschinenunabhängige Zwischensprache übersetzt wurde. Bei der Ausführung von Bytecode wird ein Interpreter benötigt, der den Bytecode in Objektcode umwandelt. Diese Übersetzung nennt man Just in Time (JIT)-Compilation.

JIT-Compiler Ein JIT-Compiler ist ein Programm, das zur Laufzeit einen Bytecode in Objektcode übersetzt und diesen ausführt. Ein Beispiel ist die Programmiersprache Java mit der Java Virtual Machine (JVM).

Namespace Ein Namespace (engl. für Namensraum) definiert den Gültigkeitsbereich von Funktionen und Variablennamen. Namensräume können ineinander geschachtelt werden, um den Quelltext systematisch zu separieren. Dadurch können stets die gleichen Namen für Funktionen und Variablen verwendet werden, ohne das es zu Komplikationen kommt.

Name-Mangling C++ ist eine objektorientierte Programmiersprache, die wie Java eine Überladung von Funktionen erlaubt. Funktionsüberladung bedeutet, dass Funktionsnamen mehrfach verwendet werden können. Die einzige Bedingung ist, dass die Funktion sich entweder in ihren Übergabeparametern oder in dem Namespace unterscheidet, in dem sie deklariert ist. Bei der Übersetzung des Quellcodes werden die Funktionen in eindeutige Namen umgewandelt. Dabei wird der Funktionsname mit den Namen der Übergabeparameter und Datentypen encodiert. Dieses Verfahren wird gemäß IBM Corporation (2003) als *name-mangling* bezeichnet und ist je nach Compiler unterschiedlich implementiert. Diese Umwandlung ist notwendig, weil die ersten C++ Compiler den Quellcode zuerst in die Sprache C umwandelten. In C dürfen Funktionsnamen nur einmalig vergeben werden. Das Verfahren kann im Quelltext durch den `extern "C"`-Bezeichner verhindert werden.

Objektcode Objektcode ist der auf einen Maschinentyp übersetzte Quellcode, er kann direkt ausgeführt werden. Objektcode kann nur auf den Architekturen ausgeführt werden, für die er kompiliert wurde.

Literaturverzeichnis

Aho, *Alfred V.*; *Lam*, *Monica S.*; *Sethi*, *Ravi* und *Ullman*, *Jeffrey D.* (2007): *Compilers: Principles, Techniques and Tools (for Anna University), 2/e*. 2. Aufl. Pearson Education - Addison Wesley.

Bach, *Maurice J* (1986): *The design of the UNIX operating system*. Bd. 1. Prentice-Hall Englewood Cliffs, NJ.

Bougacha, *Ahmed*; *Aubey*, *Geoffroy*; *Collet*, *Pierre*; *Coudray*, *Thomas*; *Salwan*, *Jonathan* und *de la Vieuville*, *Amaury* (2013): *Dagger - Decompiling to IR*. URL: `http://llvm.org/devmtg/2013-04/bougacha-slides.pdf` (besucht am 15. 09. 2014).

Dinaburg, *Artem* und *Ruef*, *Andrew* (2014): *Static Translation of x86 Instructions to LLVM*. URL: `https://www.trailofbits.com/resources/McSema.pdf` (besucht am 14. 09. 2014).

Eagle, *Chris* (2008): *The IDA pro book: the unofficial guide to the world's most popular disassembler*. No Starch Press.

IBM Corporation (2003): *Name Mangling*. URL: `http://publib.boulder.ibm.com/infocenter/macxhelp/v6v81/index.jsp?topic=/com.ibm.vacpp6m.doc/language/ref/clrc01name_mangling.htm` (besucht am 13. 09. 2014).

Katz, *Deby* (2014): *15-745: Optimizing Compilers*. URL: `http://www.cs.cmu.edu/afs/cs.cmu.edu/academic/class/15745-s14/public/lectures/` (besucht am 07. 08. 2014).

Kernighan, *Brian W*; *Ritchie*, *Dennis M* und *Ejeklint*, *Per* (1988): *The C programming language*. Bd. 2. prentice-Hall Englewood Cliffs.

Kiefer, *Mirko* (2013): *CMake by Example*. URL: `http://mirkokiefer.com/blog/2013/03/cmake-by-example/` (besucht am 13. 09. 2014).

Lattner, *Chris* (2007): *The LLVM Compiler Infrastructure*. URL: `http://llvm.org` (besucht am 07. 08. 2014).

Mikushin, *Dmitry* (2013): *Visualizing code structure in LLVM*. URL: `http://icsweb.inf.unisi.ch/cms/images/stories/ICS/slides/llvm-graphs.pdf` (besucht am 15. 09. 2014).

Moll, *Simon* (2011): „Decompilation of LLVM IR“. Saarbrücken.

Pawelczak, *Dieter* (2013a): *Maschinenorientiertes Programmieren 2*. Skriptum zur gleichnamigen Lehrveranstaltung WT13 4. Auflage.

— (2013b): *Programmerzeugungssysteme*. Skriptum zur gleichnamigen Lehrveranstaltung.

QuantAlea GmbH (2014): *Retargetability*. URL: `https://www.quantalea.net/media/_doc/2/7/manual/index.html?LLVMandNVVM.html` (besucht am 14. 09. 2014).

Segal, *Loren* (2009): *Writing Your Own Toy Compiler Using Flex, Bison and LLVM*. URL: `http://gnuu.org/2009/09/18/writing-your-own-toy-compiler/` (besucht am 14. 09. 2014).

Trieu, *Richard* (2013): *Comparison of Diagnostics between GCC and Clang*. URL: `https://gcc.gnu.org/wiki/ClangDiagnosticsComparison` (besucht am 14. 09. 2014).

Abkürzungsverzeichnis

CFG-DFG	Control- and Dataflow Graph
DARPA	Defense Advanced Research Projects Agency
GCC	GNU C-Compiler
GNU	GNU is not UNIX (UNIX-ähnliches Betriebssystem)
IR	Intermediate Representation
JIT	Just in Time
JVM	Java Virtual Machine
LLVM	Eigenname, ehemals: Low Level Virtual Machine
MC	LLVM Machine Code Project
moc	Meta-Object-Compiler
SIMD	Single Instruction, Multiple Data
SSA	Static Single Assignment
SSE	Streaming SIMD Extensions

Abbildungsverzeichnis

Listingverzeichnis

Tabellenverzeichnis

Anhang

A. Ausgaben zur Kompilierung eines Programms mit LLVM-Tools

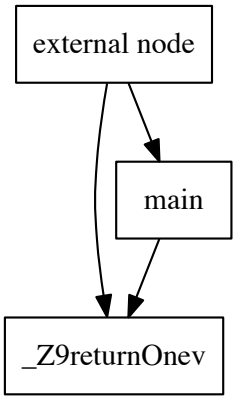
```
; ModuleID = 'main.cpp'
target datalayout = "e-p:64:64:64-i1:8:8-i8:8:8-i16:16:16-i32:32:32
        -i64:64:64-f32:32:32-f64:64:64-v64:64:64-v128:128:128-a0:0:64
        -s0:64:64-f80:128:128-n8:16:32:64-S128"
target triple = "x86_64-apple-macosx10.9.0"

; Function Attrs: nounwind ssp uwtable
define i32 @main(i32, i8**) #0 {
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
  %5 = alloca i8**, align 8
  store i32 0, i32* %3
  store i32 %0, i32* %4, align 4
  store i8** %1, i8*** %5, align 8
  ret i32 141
}

attributes #0 = { nounwind ssp uwtable "less-precise-fpmad"="false"
        "no-frame-pointer-elim"="true" "no-frame-pointer-elim-non-leaf"
        "no-infs-fp-math"="false" "no-nans-fp-math"="false"
        "stack-protector-buffer-size"="8" "unsafe-fp-math"="false"
        "use-soft-float"="false" }

!llvm.ident = !{!0}

!0 = metadata !{metadata !"Apple LLVM version 5.1 (clang-503.0.40)
        (based on LLVM 3.4svn)"}
```

Listing A.1: Kompletter LLVM-IR-Code nach Kompilierung aus C-Source

```
1         .section        __TEXT,__text,regular,pure_instructions
2         .macosx_version_min 10, 9
3         .globl          _main
4         .align          4, 0x90
5 _main:                                  ## @main
6         .cfi_startproc
7 ## BB#0:
8         pushq           %rbp
9 Ltmp0:
10        .cfi_def_cfa_offset 16
11 Ltmp1:
12        .cfi_offset %rbp, -16
13        movq            %rsp, %rbp
14 Ltmp2:
15        .cfi_def_cfa_register %rbp
16        movl            $0, -4(%rbp)
17        movl            %edi, -8(%rbp)
18        movq            %rsi, -16(%rbp)
19        movl            $141, %eax
20        popq            %rbp
21        retq
22        .cfi_endproc
23
24
25 .subsections_via_symbols
```

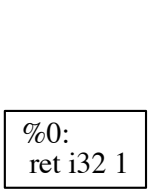
Listing A.2: Assemblercode nach Kompilierung aus LLVM-IR

B. Beispiele zur Visualisierung von Listing 2.1

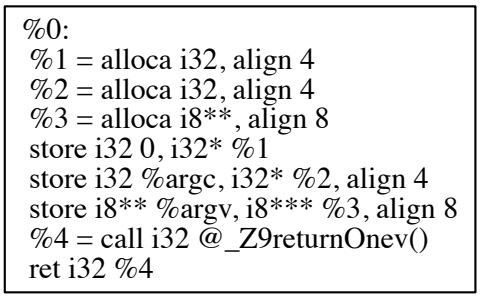


Call graph

Abbildung B.1: Call-Graph von `main.ll`



CFG for '_Z9returnOnev' function



CFG for 'main' function

Abbildung B.2: Controlflow-Graphen von `main.ll`

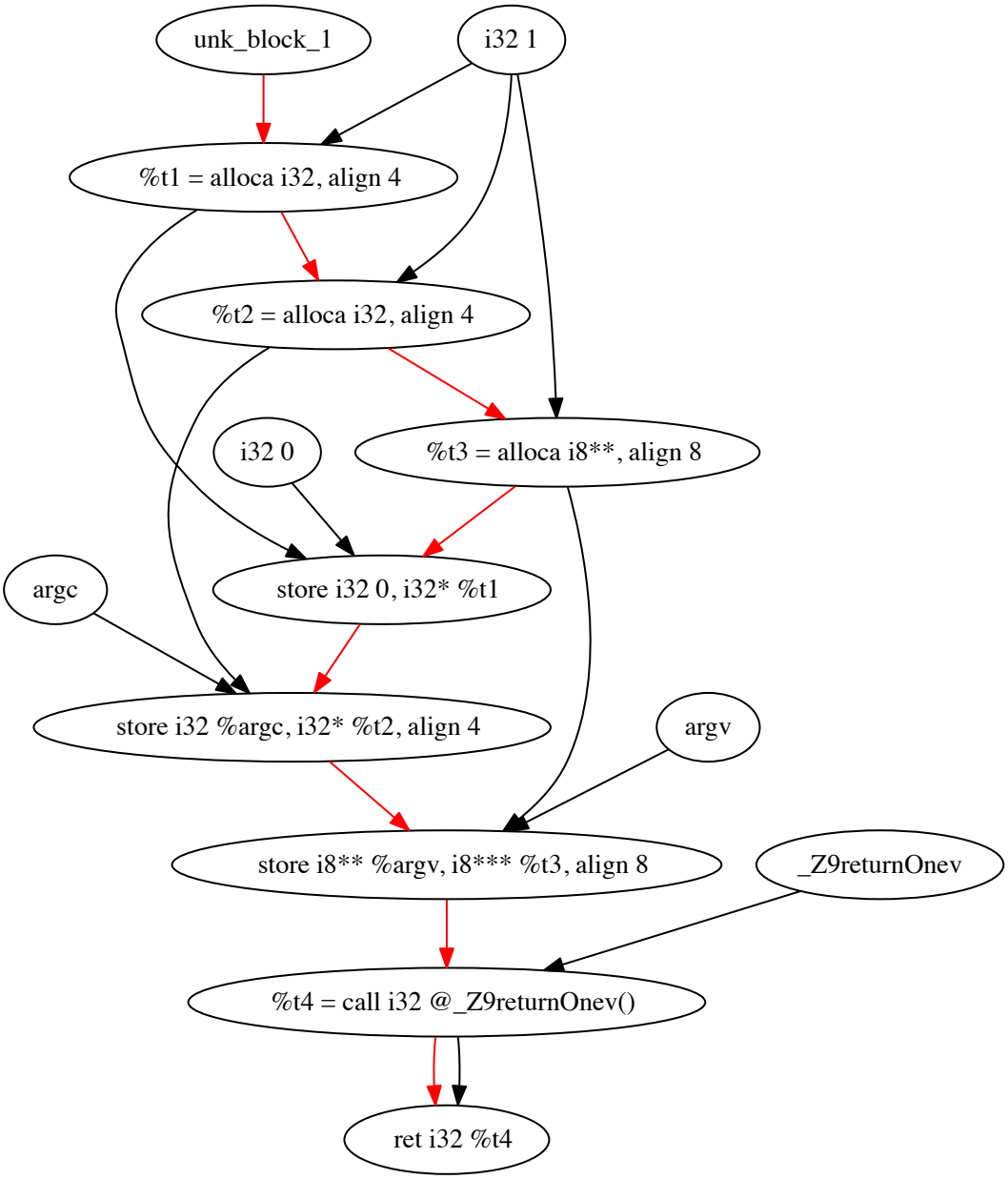


Abbildung B.3: CFG-DFG von `main.ll`

C. Generierter LLVM-IR-Code von *Dagger*

Der folgende LLVM-IR-Code wurde aus der kompilierten Datei `main.out` aus Abschnitt 2.2 mit dem Programm `llvm-dec` aus dem Projekt *Dagger* gemäß Listing C.3 generiert.

```
~/Developer/dagger/build/bin/llvm-dec main.out -o main.dagger.ll
```

Listing C.3: Befehl zur Dekompilierung mit *Dagger*

```
; ModuleID = 'output'

%regset = type { i16, i16, i32, i16, i16, i16, i16, i64, i64, i64, i64, \
  i64, i64, i64, i64, i64, i16, i64, i64, i64, i64, i64, i64, i64, i64, \
  i64, i64, i64, i64, i64, i64, i64, i64, i32, i32, i32, i32, i32, i32, \
  i32, i32, i80, i80, i80, i80, i80, i80, i80, i16, i16, i16, i16, i16, \
  i16, i16, i16, i64, i64, i64, i64, i64, i64, i64, i64, i64, i64, i64, \
  i64, i64, i64, i64, i64, i80, i80, i80, i80, i80, i80, i80, i80, i512, \
  i512, i512, i512, i512, i512, i512, i512, i512, i512, i512, i512, i512, \
  i512, i512, i512, i512, i512, i512, i512, i512, i512, i512, i512, i512, \
  i512, i512, i512, i512, i512, i512, i512 }

define void @main_init_regset(%regset*, i8*, i32, i32, i8**) {
  %6 = ptrtoint i8* %1 to i64
  %7 = zext i32 %2 to i64
  %8 = add i64 %6, %7
  %9 = sub i64 %8, 8
  %10 = inttoptr i64 %9 to i64*
  store i64 -1, i64* %10
  %11 = getelementptr inbounds %regset* %0, i32 0, i32 15
  store i64 %9, i64* %11
  %12 = zext i32 %3 to i64
  %13 = getelementptr inbounds %regset* %0, i32 0, i32 11
  store i64 %12, i64* %13
  %14 = ptrtoint i8** %4 to i64
  %15 = getelementptr inbounds %regset* %0, i32 0, i32 14
  store i64 %14, i64* %15
  ret void
}
```

```
30
31 define i32 @main_fini_regset(%regset*) {
32   %2 = getelementptr inbounds %regset* %0, i32 0, i32 7
33   %3 = load i64* %2
34   %4 = trunc i64 %3 to i32
35   ret i32 %4
36 }
37
38 define void @fn_100000F80(%regset* noalias nocapture) {
39 entry_fn_100000F80:
40   %RIP_ptr = getelementptr inbounds %regset* %0, i32 0, i32 13
41   %RIP_init = load i64* %RIP_ptr
42   %RIP = alloca i64
43   store i64 %RIP_init, i64* %RIP
44   %EIP_init = trunc i64 %RIP_init to i32
45   %EIP = alloca i32
46   store i32 %EIP_init, i32* %EIP
47   %IP_init = trunc i64 %RIP_init to i16
48   %IP = alloca i16
49   store i16 %IP_init, i16* %IP
50   %RBP_ptr = getelementptr inbounds %regset* %0, i32 0, i32 8
51   %RBP_init = load i64* %RBP_ptr
52   %RBP = alloca i64
53   store i64 %RBP_init, i64* %RBP
54   %RSP_ptr = getelementptr inbounds %regset* %0, i32 0, i32 15
55   %RSP_init = load i64* %RSP_ptr
56   %RSP = alloca i64
57   store i64 %RSP_init, i64* %RSP
58   %ESP_init = trunc i64 %RSP_init to i32
59   %ESP = alloca i32
60   store i32 %ESP_init, i32* %ESP
61   %SP_init = trunc i64 %RSP_init to i16
62   %SP = alloca i16
63   store i16 %SP_init, i16* %SP
64   %SPL_init = trunc i64 %RSP_init to i8
65   %SPL = alloca i8
66   store i8 %SPL_init, i8* %SPL
```

```
67    %EBP_init = trunc i64 %RBP_init to i32
68    %EBP = alloca i32
69    store i32 %EBP_init, i32* %EBP
70    %BP_init = trunc i64 %RBP_init to i16
71    %BP = alloca i16
72    store i16 %BP_init, i16* %BP
73    %BPL_init = trunc i64 %RBP_init to i8
74    %BPL = alloca i8
75    store i8 %BPL_init, i8* %BPL
76    %RAX_ptr = getelementptr inbounds %regset* %0, i32 0, i32 7
77    %RAX_init = load i64* %RAX_ptr
78    %RAX = alloca i64
79    store i64 %RAX_init, i64* %RAX
80    %EAX_init = trunc i64 %RAX_init to i32
81    %EAX = alloca i32
82    store i32 %EAX_init, i32* %EAX
83    %AX_init = trunc i64 %RAX_init to i16
84    %AX = alloca i16
85    store i16 %AX_init, i16* %AX
86    %AL_init = trunc i64 %RAX_init to i8
87    %AL = alloca i8
88    store i8 %AL_init, i8* %AL
89    %1 = lshr i64 %RAX_init, 8
90    %AH_init = trunc i64 %1 to i8
91    %AH = alloca i8
92    store i8 %AH_init, i8* %AH
93    %RDI_ptr = getelementptr inbounds %regset* %0, i32 0, i32 11
94    %RDI_init = load i64* %RDI_ptr
95    %RDI = alloca i64
96    store i64 %RDI_init, i64* %RDI
97    %EDI_init = trunc i64 %RDI_init to i32
98    %EDI = alloca i32
99    store i32 %EDI_init, i32* %EDI
100   %RSI_ptr = getelementptr inbounds %regset* %0, i32 0, i32 14
101   %RSI_init = load i64* %RSI_ptr
102   %RSI = alloca i64
103   store i64 %RSI_init, i64* %RSI
```

```
104   br label %bb_100000F80
105
106 exit_fn_100000F80:                                ; preds = %bb_100000F80
107   %2 = load i64* %RAX
108   store i64 %2, i64* %RAX_ptr
109   %3 = load i64* %RBP
110   store i64 %3, i64* %RBP_ptr
111   %4 = load i64* %RDI
112   store i64 %4, i64* %RDI_ptr
113   %5 = load i64* %RIP
114   store i64 %5, i64* %RIP_ptr
115   %6 = load i64* %RSI
116   store i64 %6, i64* %RSI_ptr
117   %7 = load i64* %RSP
118   store i64 %7, i64* %RSP_ptr
119   ret void
120
121 bb_100000F80:                                     ; preds = %entry_fn_100000F80
122   %EIP_0 = trunc i64 4294971264 to i32
123   %8 = trunc i64 4294971264 to i32
124   %IP_0 = trunc i64 4294971264 to i16
125   %9 = trunc i64 4294971264 to i16
126   %RIP_1 = add i64 4294971264, 1
127   %EIP_1 = trunc i64 %RIP_1 to i32
128   %IP_1 = trunc i64 %RIP_1 to i16
129   %RBP_0 = load i64* %RBP
130   %RSP_0 = load i64* %RSP
131   %10 = sub i64 %RSP_0, 8
132   %11 = inttoptr i64 %10 to i64*
133   store i64 %RBP_0, i64* %11
134   %RSP_1 = sub i64 %RSP_0, 8
135   %ESP_0 = trunc i64 %RSP_1 to i32
136   %12 = trunc i64 %RSP_1 to i32
137   %SP_0 = trunc i64 %RSP_1 to i16
138   %13 = trunc i64 %RSP_1 to i16
139   %SPL_0 = trunc i64 %RSP_1 to i8
140   %14 = trunc i64 %RSP_1 to i8
```

```
141   %RIP_2 = add i64 %RIP_1, 3
142   %EIP_2 = trunc i64 %RIP_2 to i32
143   %IP_2 = trunc i64 %RIP_2 to i16
144   %EBP_0 = trunc i64 %RSP_1 to i32
145   %15 = trunc i64 %RSP_1 to i32
146   %BP_0 = trunc i64 %RSP_1 to i16
147   %16 = trunc i64 %RSP_1 to i16
148   %BPL_0 = trunc i64 %RSP_1 to i8
149   %17 = trunc i64 %RSP_1 to i8
150   %RIP_3 = add i64 %RIP_2, 5
151   %EIP_3 = trunc i64 %RIP_3 to i32
152   %IP_3 = trunc i64 %RIP_3 to i16
153   %RAX_0 = load i64* %RAX
154   %18 = trunc i64 %RAX_0 to i32
155   %RAX_1 = zext i32 141 to i64
156   %AX_0 = trunc i32 141 to i16
157   %19 = trunc i64 %RAX_1 to i16
158   %AL_0 = trunc i32 141 to i8
159   %20 = trunc i64 %RAX_1 to i8
160   %21 = lshr i32 141, 8
161   %AH_0 = trunc i32 %21 to i8
162   %22 = lshr i64 %RAX_1, 8
163   %23 = trunc i64 %22 to i8
164   %RIP_4 = add i64 %RIP_3, 7
165   %EIP_4 = trunc i64 %RIP_4 to i32
166   %IP_4 = trunc i64 %RIP_4 to i16
167   %24 = add i64 %RSP_1, -4
168   %25 = inttoptr i64 %24 to i32*
169   store i32 0, i32* %25
170   %RIP_5 = add i64 %RIP_4, 3
171   %EIP_5 = trunc i64 %RIP_5 to i32
172   %IP_5 = trunc i64 %RIP_5 to i16
173   %RDI_0 = load i64* %RDI
174   %EDI_0 = trunc i64 %RDI_0 to i32
175   %26 = add i64 %RSP_1, -8
176   %27 = inttoptr i64 %26 to i32*
177   store i32 %EDI_0, i32* %27
```

```
178   %RIP_6 = add i64 %RIP_5, 4
179   %EIP_6 = trunc i64 %RIP_6 to i32
180   %IP_6 = trunc i64 %RIP_6 to i16
181   %RSI_0 = load i64* %RSI
182   %28 = add i64 %RSP_1, -16
183   %29 = inttoptr i64 %28 to i64*
184   store i64 %RSI_0, i64* %29
185   %RIP_7 = add i64 %RIP_6, 1
186   %EIP_7 = trunc i64 %RIP_7 to i32
187   %IP_7 = trunc i64 %RIP_7 to i16
188   %RSP_2 = add i64 %RSP_1, 8
189   %ESP_1 = trunc i64 %RSP_2 to i32
190   %SP_1 = trunc i64 %RSP_2 to i16
191   %SPL_1 = trunc i64 %RSP_2 to i8
192   %30 = sub i64 %RSP_2, 8
193   %31 = inttoptr i64 %30 to i64*
194   %RBP_1 = load i64* %31
195   %EBP_1 = trunc i64 %RBP_1 to i32
196   %BP_1 = trunc i64 %RBP_1 to i16
197   %BPL_1 = trunc i64 %RBP_1 to i8
198   %RIP_8 = add i64 %RIP_7, 1
199   %EIP_8 = trunc i64 %RIP_8 to i32
200   %IP_8 = trunc i64 %RIP_8 to i16
201   %RSP_3 = add i64 %RSP_2, 8
202   %32 = inttoptr i64 %RSP_2 to i64*
203   %RIP_9 = load i64* %32
204   %ESP_2 = trunc i64 %RSP_3 to i32
205   %SP_2 = trunc i64 %RSP_3 to i16
206   %SPL_2 = trunc i64 %RSP_3 to i8
207   %EIP_9 = trunc i64 %RIP_9 to i32
208   %IP_9 = trunc i64 %RIP_9 to i16
209   store i8 %AH_0, i8* %AH
210   store i8 %AL_0, i8* %AL
211   store i16 %AX_0, i16* %AX
212   store i16 %BP_1, i16* %BP
213   store i8 %BPL_1, i8* %BPL
214   store i32 141, i32* %EAX
```

```
215   store i32 %EBP_1, i32* %EBP
216   store i32 %EDI_0, i32* %EDI
217   store i32 %EIP_9, i32* %EIP
218   store i32 %ESP_2, i32* %ESP
219   store i16 %IP_9, i16* %IP
220   store i64 %RAX_1, i64* %RAX
221   store i64 %RBP_1, i64* %RBP
222   store i64 %RDI_0, i64* %RDI
223   store i64 %RIP_9, i64* %RIP
224   store i64 %RSI_0, i64* %RSI
225   store i64 %RSP_3, i64* %RSP
226   store i16 %SP_2, i16* %SP
227   store i8 %SPL_2, i8* %SPL
228   br label %exit_fn_100000F80
229 }
230
231 ; Function Attrs: noreturn nounwind
232 declare void @llvm.trap() #0
233
234 define i32 @main(i32, i8**) {
235   %3 = alloca %regset
236   %4 = alloca [8192 x i8]
237   %5 = getelementptr inbounds [8192 x i8]* %4, i32 0, i32 0
238   call void @main_init_regset(%regset* %3, i8* %5, i32 8192, i32 %0, i8** %1)
239   call void @fn_100000F80(%regset* %3)
240   %6 = call i32 @main_fini_regset(%regset* %3)
241   ret i32 %6
242 }
243
244 attributes #0 = { noreturn nounwind }
```

Listing C.4: Generierter LLVM-IR-Code von Dagger

D. Installation von LLVM, Clang und Fracture unter Mac OSX

```
# Installation der XCode command-line-tools
xcode-select --install
# Installation von autoconf und automake
brew install autoconf automake
# Setzen des Installationspfades
export DESTINATION=$HOME/Developer
# Installation von LLVM und Clang
cd $DESTINATION
git clone https://github.com/draperlaboratory/llvm llvm
cd llvm/tools
git clone https://github.com/draperlaboratory/clang clang
cd ..
./configure --enable-debug-symbols --prefix=/usr/local \
    --build=x86_64-apple-darwin13.3.0
make -j16
sudo make install
# Installation von Fracture
cd $DESTINATION
git clone https://github.com/draperlaboratory/fracture.git fracture
cd fracture
export CXXFLAGS="-std=c++11 -stdlib=libc++ \
    -I/Applications/Xcode.app/Contents/Developer\
    /Toolchains/XcodeDefault.xctoolchain/usr/lib/c++/v1"
./autoconf/AutoRegen.sh
./configure --enable-debug-symbols --with-llvmsrc=$DESTINATION/llvm/ \
    --with-llvmobj=$DESTINATION/llvm/
make -j16
```

Listing D.5: Kompilieren von Fracture

E. Die Erstellung eines CMake-Projekts am Beispiel von binSpector

```
# add a target to generate API documentation with Doxygen
FIND_PACKAGE(Doxygen)
IF(DOXYGEN_FOUND)
        CONFIGURE_FILE(
                ${CMAKE_CURRENT_SOURCE_DIR}/Doxyfile.in
                ${CMAKE_CURRENT_BINARY_DIR}/Doxyfile @ONLY
        )
        ADD_CUSTOM_TARGET(doc ALL
                ${DOXYGEN_EXECUTABLE} ${CMAKE_CURRENT_BINARY_DIR}/Doxyfile
                WORKING_DIRECTORY ${CMAKE_CURRENT_BINARY_DIR}
                COMMENT "Generating API documentation with Doxygen" VERBATIM
        )
ENDIF(DOXYGEN_FOUND)
```

Listing E.6: Die Datei `binSpector/docs/CMakeLists.txt`

```
INCLUDE_DIRECTORIES("control")
INCLUDE_DIRECTORIES("view")
ADD_LIBRARY(binSpectorLib
        ./control/codeEditor.cpp
        ./control/Disassembler.cpp
        ./control/syntax/clangHighlighter.cpp
        ./view/binary/basicInfo.cpp
        ./view/guiFunctions.cpp
        ./view/mainWidget/codeViewer.cpp
        ./view/mainWidget/help.cpp
        ./view/visualizer/callGraph.cpp
        ./view/visualizer/controlAndDataFlowGraph.cpp
        ./view/visualizer/controlFlowGraph.cpp
        ./view/visualizer/memoryDependence.cpp
        ./view/binspector.cpp
)

QT5_USE_MODULES(binSpectorLib Widgets)
```

Listing E.7: Die Datei `binSpector/lib/CMakeLists.txt`

```
1  INCLUDE_DIRECTORIES("../../include")
2
3  SET( _MOC_HDRS
4          ../../include/control/codeEditor.h
5          ../../include/control/Disassembler.h
6          ../../include/control/syntax/clangHighlighter.h
7          ../../include/view/binary/basicInfo.h
8          ../../include/view/guiFunctions.h
9          ../../include/view/mainWidget/codeViewer.h
10         ../../include/view/mainWidget/help.h
11         ../../include/view/visualizer/callGraph.h
12         ../../include/view/visualizer/controlAndDataFlowGraph.h
13         ../../include/view/visualizer/controlFlowGraph.h
14         ../../include/view/visualizer/memoryDependence.h
15         ../../include/view/binspector.h)
16
17 QT5_WRAP_CPP( _MOC_SRCS ${_MOC_HDRS} )
18
19 IF( APPLE )
20         ADD_EXECUTABLE(
21                 ${PROGNAME} MACOSX_BUNDLE main.cpp ${_SRCS} ${_MOC_SRCS}
22         )
23     ADD_CUSTOM_COMMAND( TARGET ${PROGNAME} POST_BUILD
24             COMMAND mkdir ARGS -p
25             ${CMAKE_CURRENT_BINARY_DIR}/${PROGNAME}.app/Contents/Resources
26             COMMAND cp ARGS
27             ../../../${MACOSX_BUNDLE_ICON_FILE}
28             ${CMAKE_CURRENT_BINARY_DIR}/${PROGNAME}.app/Contents/Resources
29             COMMAND cp ARGS
30             ../../../*.md
31             ${CMAKE_CURRENT_BINARY_DIR}/${PROGNAME}.app/Contents/Resources
32     )
33 ELSE( APPLE )
34         ADD_EXECUTABLE(
35                 ${PROGNAME} main.cpp ${_SRCS} ${_MOC_SRCS}
36         )
```

```
37 ENDIF( APPLE )
38
39 TARGET_LINK_LIBRARIES(${PROGNAME} ${QT_LIBRARIES} binSpectorLib)
40 QT5_USE_MODULES(${PROGNAME} Widgets)
```

Listing E.8: Die Datei `binSpector/tools/binSpector/CMakeLists.txt`