Besondere Lernleistung - Dokumentation

Inokentiy Babushkin

February 12, 2015

Inhalt

1 Grundlagen

Die vorliegende Besondere Lernleistung hat das Ziel, ein Softwareprojekt umfassend zu planen und umzusetzen. Dieses Dokument beschreibt Zielsetzung, Aufbau und konkrete Aspekte der Umsetzung, wobei der Hauptaugenmerk auf Algorithmen, Vorgehensweisen und Designentscheidungen liegen wird.

2 Eine kurze Einführung in Reverse Engineering und Dekompilierung

2.1 Grundlegende Funktionsweise von CPU und die Abstraktion der Assemblersprache

[1] Die CPU, also der Prozessor eienes Computersystems unterteilt jedes mögliche Programm in sogenannte Instruktionen, die die kleinste unteilbare Einheit eines Programms darstellen. Es werden verschiedene Instruktionsklassen unterschieden. Allgemein ist jede Instruktion nach folgendem Muster aufgebaut: Sie

[1] Die hier beschriebene Syntax ist die Intel-Syntax, die unter der x86(-64) Architektur am weitesten verbreitet ist.

besteht aus 2 bis 3 Tokens (Wörtern), wobei der erste den Namen der Instruktion, der diese eindeutig kennzeichnet, darstellt. Im Assembler-Jargon wird dieser als *Mnemonic* bezeichnet. Die folgenden 1 bis 2 Tokens sind sogenannte *Operanden*, die noch weiter in *Source-* und *Destination-* Operand unterteilt werden. Diese sind immer eine Referenz auf Daten, auf die die Instruktion angewandt wird. Beispielsweise gibt es Instruktionen für Grundrechenarten und bitlogische Operationen, wobei folgende Darstellungsform verwendet wird:

$$instruction\ destination[, source]$$

2.1.1 Einige häufig anzutreffende Instruktionen

- *add:* Addiert Source- und Destination-Operand und speichert das Ergebnis in Destination.
- *sub:* Subtrahiert den Source- vom Destination Operand und speichert dort das Ergebnis.
- *mov:* Kopiert die Daten aus dem Source- in den Destination-Operand.
- *push:* Nimmt nur einen Source-Operanden und legt diesen auf dem Stack ab.
- *pop:* Nimmt nur einen Destination-Operanden und legt in ihm das letzte Element auf dem Stack ab, was seiner Größe entspricht. Wenn also ein 4 Byte großer Speicherbereich den Destination-Operanden darstellt, werden die obersten 4 Byte auf dem Stack in ihm abgelegt und der Stack entsprechend verkleinert.
- *cmp:* Führt einen Vergleich der Operanden durch und setzt entsprechende CPU-Flags. Diese sind in einem speziellen EFLAGS-Register zu finden und sind Boolsche Werte, die über das Ergebnis von Vergleichen etc. Auskunft geben. Es existiert eine Reihe von Instruktionen, die in Abhängigkeit vom Zustand bestimmter Flags ausgeführt werden.
- *Sprünge:* Der einfachste Sprung, *jmp*, wird immer ausgeführt und springt zur Adresse / zum Label in seinem einzigen Operanden. Alle anderen Sprnginstruktionen fragen vor der tatsächlichen Sprunginstruktion die CPU-Flags ab, sind also sogenannte *bedingte Sprünge*. Deren Instruktions-Mnemonics fangen in der Regel mir einem *j* an.
- *call* Ruft die durch den einzigen Operanden identifizierte Funktion auf.

Diese Liste ist natürlich bei weitem nicht vollständig, da sie nur dazu gedacht ist, die dem Programm beigefügten Test-Cases und Beispiele für vollständige Neulinge verständlicher zu gestalten. Um ein umfassendes Grundwissen über Programmierung und Verständnis von Assembler (zwecks Reverse-Engineering) zu erlangen, empfehlen sich die unter den Literaturverweisen [1] und [2] referenzierten Wikibooks.

2.1.2 Funktionen

Zum weiteren Verständnis des Projekts ist eine Erklärung von Funktionen in Assembler vonnöten. Eine Funktion ist, genau wie in Hochsprachen, ein Codeabschnitt, der blockweise ausgeführt wird, wobei innerhalb eines solchen Blocks Verzweigungen möglich sind. Eine Funktion kann Argumente (Parameter) annehmen, welche für ihre Funktionsweise notwendig sind und meistens über den Stack übergeben werden. Auch kann eine Funktion Variablen mit lokal begrenzter Gültigkeit auf dem eigenen Stackframe, dem für jede aufgerufene Instanz individuellen Stackabschnitt, ablegen. Dabei muss jede Funktion ihren Stackframe selbst initialisieren und - je nach Aufrufkonvention - wieder deinitialisieren. Gleichzeitig legt die Definition der Funktion die Reihenfolge der Parameter auf dem Stack fest. Auch dies ist nur eine sehr kurze Einführung in die grundlegenden Konzepte, die für das Verständnis von Assembler-Programmen relevant sind. Auch hier sei nochmal auf [1] und [2] verwiesen.

2.2 Compiler, Decompiler und Disassembler

Jede Hochsprache, die bis jetzt entwickelt wurde, hatte das Ziel, dem Programmierer ein effizientes Werkzeug bereitzustellen, welches verhältnismäßig einfache Softwareentwicklung möglich machen sollte, ohne ihn in seiner Handlungsfreiheit zu beschränken. Gleichzeitig soll der gesamte Prozess der Entrwicklung für den Programmierer so transparent wie möglich sein. Tatsächlich hat sich dieser Ansatz bewährt und erfüllt die Zielvorgaben. Damit die in einer solchen Sprache formalisierten Algortihmen auch von einem Prozessor ausgeführt werden können, muss der Quellcode erst in eine für den Computer angemessene Form überführt werden, das heißt in die Assemblersprache beziehungsweise ihre Representation als einzelne Bytes im Speicher. Genau dies ist die Aufgabe eines sogenannten Compilers: Dieses Programm nimmt Hochsprachen-Quellcode als Eingabe und gibt entweder Assemblerprogramme oder direkt ausführbare Binärdateien zurück, die das selbe Programm repräsentieren. Dabei verlieren die so umgewandelten Programme jedoch viele Informationen, die für die Ausführung unerheblich sind, zu ihrem Verständis jedoch entscheidend beitragen: Kommentare, Klassenprototypen, Variablennamen und viele weitere Details, die das Lesen von Quellcode durch menschliche Nutzer vereinfachen sollen. Wenn ein Programm jedoch nur in Binärform vorliegt, ist die Analyse - je nach Komplexität des Analyseziels - kompliziert und erfordert fundierte Fachkenntnisse über die Funktionsweise der CPU und grundlegende Konzepte der Assemblerprogrammerung, zumal die erwähnten Vereinfachungen und Hilfestellungen der Hochsprachen nicht verfügbar sind. Stattdessen besitzt die Assemblersprache - wie im vorangehenden Abschnitt eräutert - keine Kontrollstrukturen im eigentlichen Sinn, was es schwierig macht, den Kontrollfluss eines Programms, das in dieser Form vorliegt, zu verstehen. Die tatsächliche Umwandlung einer Binärdatei in ein Assemblerlisting, welches grundlegende Abstraktionen über einzelne Instruktion der CPU bereiststellt, ist Aufgabe eines Disassemblers, der aus diesem Grund ein zentrales Wekzeug eines Reverse Engineers darstellt. Durch die allgemeine Ausrichtung eines Compilers, nämlich schnell und effizient arbeitenden Code zu generieren und dabei auf (hürdenfreie) Menschenlesbarkeit unter Umständen zu verzichten, wird diese Art der Analyse zusehends zeitintensiv. Aus diesem Grund gibt es auch einige Ansätze, diesen Prozess komplett zu automatisieren, so dass keine oder kaum Nutzrinteraktion notwendig ist, wovon sich die Entwickler eine enorme Beschleunigng des Prozesses erhoffen. Diese technischen Lösungen sind als Decompiler bekannt, da sie - im Idealfall - die Funktionsweise eines Compilers umkehren. Da diese meist über eine für ein Programm überaus komplexe Aufgabe erfüllen, sind es auch in der Regel kommerzielle Produkte, die auf diesem Gebiet verhältnismäßig weite Verbreitung finden, wobei auch eine Anzahl von Open-Source-Projekten sich mit dieser Zielsetzung befassen.

3 Problemantik

Das erklärte Ziel dieses Projekts ist es, ein Programm zu entwickeln, welches es ermöglicht, Software teilweise automatisch zu dekompilieren. Da moderne Software immer kompexer wird, wird es entsprechend schwieriger, in binärer Form vorliegende Softwareprodukte so umzuformen, dass sie ein tiefgehendes Verständnis und darauf basierende Modifikation erlauben. Dies betrifft nicht nur die Untersuchung von fremder Software, sondern auch die Wiederherstellung von Quellcode aus alten, beendeten oder sonstwie des Quellcodes verlustig gewordenen Projekten. Aus diesem Grund werden bereits seit einiger Zeit Ansätze unternommen, diesen Vorgang so weit wie möglich zu automatisieren. Die Hauptargumente für eine solche Vorgehensweise sind, dass die Wiederherstellung von Kontrollfluss, Variablen, Funktionen und weiteren HLL[2]-spezifischen Strukturen ohnehin nicht komplett möglich ist, das Ergebnis also bloß eine Annäherung an den Originalcode sein kann. Gleichzeitig ist der Vorgang - von Menschen durchgeführt - sehr zeitraubend, erfordert viel Fachwissen und möglichst detaillierte Kenntnisse über die Funktionsweise der zu untersuchenden Software. All dies lässt es vorteilhaft erscheinen, die recht "undankbare" Aufgabe des Reverse-Engineerings auf ein automatisiertes System zu verlagern, vorausgesetzt das Ziel ist der komplette Quellcode des zu untersuchenden Produkts. Diese Systeme beruhen auf verschiedenen theoretischen Ansätzen, Programme zu beschreiben und Muster in ihnen zu erkennen, die in ihrer Komplexität stark variieren. Dabei sind mehrer Programmkategorien zu unterscheiden:

[2]High Level Language, also Programmiersprache mit hohem Abstraktionsgrad, bekannte Beispiele für kompilierte HLL's sind C, C++ und Java

1. Decompiler, deren Ziel es ist, ein Programm komplett wiederherzustellen, die meistens eine non-interaktive Arbeitsweise erfordern.

2. Programme, die bestimmte Aspekte des zu analysierenden Codes wiederherstellen oder eine Art Anleitung für den Benutzer generieren, wie er am besten verfährt, um ein umfassendes Verständnis vom zu untersuchenden System zu erlangen.

Das Resultat der vorliegenden Besonderen Lernleistung ist in die zweite Kategorie einzuordnen, wobei es aus drei Submodulen besteht, die unabhängig arbeiten und verschiedene Bereiche des Analysegebiets abdecken. Die Entscheidung, keinen kompletten Decompiler anzustreben wurde aus mehreren Gründen getroffen: Zum Einen ist automatisierte Programmanalyse recht genau, kann jedoch trotz allem Fehler erzeugen. Deswegen ist es deutlich günstiger, eine transparente Arbeitsweise und eine enge Zusammenarbeit zwischen Benutzer und System zu ermöglichen, da auf diese Weise Fehler deutlich schneller gefunden und behoben werden können, was die Qualität der Arbeit des Nutzers wesentlich verbessert und ihm eine enorme Zeitersparnis ermöglicht. Zum Anderen sind viele Funktionen und Features in einem solchen Projekt machbar und können auch mehr oder weniger einfach integriert werden, obwohl sie kaum Nutzen erbringen, da das Verhältnis zwischen Ergebnisqualität und Programmkomplexität zu schlecht ist und die erbrachten Resultate zu fragil. Folglich ist es sinnvoll, sich auf die wesentlichen Aufgabenstellungen zu beschränken, so dass der Aufwand in Relation zum erbrachten Nutzen steht.

4 Zielsetzung

Wie bereits im letzten Abschnitt erwähnt, handelt es sich bei diesem Projekt um ein Analyseframework zur teilweisen Dekompilierung von Software, wobei auch eine Vielzahl weiterer Reverse-Engineering[3]-Aufgaben mit seiner Hilfe vereinfacht werden können. Dabei wird die Ausgabe von IDA Pro[4] als Grundlage der Analyse verwendet. Dies mag zuerst als Nachteil erscheinen, da es den Arbeitsverlauf des Nutzers einschränkt, wobei bedacht werden muss, dass der Analyse- und Input-Parsing-Code strikt getrennt wird, was es ermöglicht, andere Eingabeformate mithilfe weiterer Module zu akzeptieren, ohne Änderungen in der Programmmechanik umsetzen zu müssen. Die Umsetzung des Moduls für IDA wurde zuerst durchgeführt, da diese keine wesentliche Verarbeitung der Eingabe erfordert, um die für die Programmmechanik notwendigen Informationen bereitzustellen, weswegen ein solches Modul sich besonders gut als Demonstrationsbeispiel eignet. Für mehr Informationen zum Projektaufbau und eine graphische Darstellung sei auf den gleichnamigen Abschnitt verwiesen.

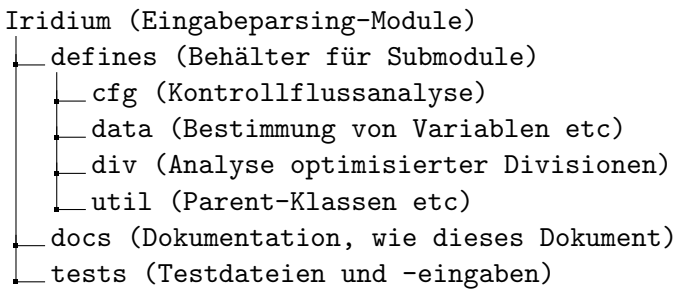
4.1 Projektstruktur

Um eine Erweiterung im Sinne des vorhergehenden Abschnittes zu gewährleisten, ist es besonders wichtig, eine sinnvole Aufteilung des Codes in einzelne Module durchzuführen. Auch sonst gehört es zu einem "guten Stil", Programme in Sinneinheiten zu koppeln. Wie bereits erwähnt, wird deswegen das Projekt grob in zwei zentrale Teile unterteilt: Die tatsächlichen Analysemodule, die alle eine Eingabe in einem bestimmten Format erwarten, und Module, die den Zweck erfüllen, Daten in diesem Format bereitzustellen und diese aus verschiedenen Eingabeformaten generieren. Die Ordnerstruktur sieht aus, wie in Figur 1 dargestellt und wird primär verwendet, um das Projekt zu strukturieren. An der Wurzel der Ordnerstruktur liegen die einzelnen Module, die Eingabe verarbeiten und in ein Format umwandeln, welches von den Analysemodulen verwendet wird. Diese wiederum befinden sich innerhalb des Moduls *defines*, welches pro Submodul einen Unterordner besitzt. Jedes dieser Submodule kann sowohl alleinstehend, also als Script, als auch als Modul benutzt bzw. imporiert werden. Auf diese Weise sind die einzelnen Komponenten fähig, miteinander in Kontakt zu treten und auf die Funktionalität anderer Module zuzugreifen, jedoch ohne zu sehr voneinander abhängig zu sein, so dass Erweiterung und Modifikation sehr einfach zu bewerkstelligen ist. Beispielsweise sind so ohne Weiteres andere Eingabeparsing-Module denkbar, als die bereits mitgelieferten. Ebenfalls kann

[3]Der Prozess, Software oder andere technische Systeme in ihrem Aufbau und Verhalten zu analysieren, um sie zu reproduzieren, zu modifizieren oder das dabei gewonnene Wissen weiterzuverwenden, z.B. um Anti-Malware Routinen zu entwickeln

[4]Ein Disassembler, der viele verschiedene Eingabeformate unterstützt und nicht nur interaktive Arbeit ermöglicht, sondern auch Assemblerlistings etc. generieren kann.

die Analyse um weitere Aspekte des zu untersuchenden Programms erweitert werden, sollte dies sich als notwendig herausstellen.



Figur 1: Die Ordnerstruktur des Projekts

4.2 Kontrollflussanalyse

Einer der zentralen Aspekte moderner wie historischer Programme ist die Möglichkeit, Code in Abhängigkeit von Bedingungen ausführen zu können. Die am weitesten bekannten Beispiele sind verschiedene Arten Bedingter Anweisungen, Schleifen u.ä. Diese Beziehungen zwischen den einzelnen Teilen des Codes wiederherstellen zu können ist eine der Hauptaufgaben dieses Projekts. Dabei ist diese Zielsetzung keineswegs trivial, zumal Strukturen linear und nicht linear verschachtelt[5] sein können etc. Als Grundlage des verwendeten Algorithmus diente die in [3] beschriebene Vorgehensweise, die jedoch um fortgeschrittenere Bedingungsanalyse erweitert wurde. Dies war notwendig, da z.B. Bedingte Anweisungen nicht nur eine, sondern mehrere über logische Operatoren verknüpfte Bedingungen besitzen können. Dies führt dazu, dass Strukturen erst erkannt werden können, wenn diese komplexen Bedingungen reduziert wurden.

4.3 Analyse von Variablen und anderen Daten

Daten bilden neben dem Code selbst den zweiten Hauptbestandteil eines Programms, weswegen das Layout der Variablen, Argumente/Parameter und Rückgabewerte von großem Interesse für den Reverse-Engineer sein sollte. Gleichzeitig sind Arrays und Variablen nicht immer trivial zu erkennen, weswegen sich einige einfache, dennoch effektive Heuristiken anbieten, die in diesem Projekt umfassend realisiert wurden. Die eigentliche Problematik liegt darin, dass Compiler nicht alle Daten auf dem Stack ablegen, aber gleichzeitig den Stack nicht nur zum Speichern lokaler Variablen verwenden. Folglich muss unterschieden werden, welche Speicherbereiche Variablen abbilden, welchen Datentyp diese besitzen und ob diese gegebenenfalls Pointer darstellen. Aufgrund der oben beschrieben Umstände handelt es sich bei den Analyseergebnissen nur um eine Annäherung an die Realität, beispielsweise verwenden optimisierende Compiler zum Speichern lokaler Variablen häufig CPU-Register [6] , die von dem vorliegenden Modul nicht berücksichtigt werden, da auch so viel Arbeit in zweifelhafte Ergebnisse investiert werden würde, zumal Register auch für Arithmetik und Argumentübergabe verwendet werden können.

4.4 Behandlung von Integerdivision

Die Integerdivision ist, ebenso wie die Modulodivision, eine verhältnismäßig häufig verwendete Rechenarten in Programmen, bzw. ist meistens von zentraler Bedeutung für die Resultate der Ausführung. Allerd-

[5] Strukturen in einem CFG bestehen in der Regel aus zwei oder Mehr Teilen, wobei sogenannte primitive Teile einfache Knoten des Kontrollflussgraphen sind. In der Praxis können jedoch auch nich-primitive Teile, also andere Strukturen an dieser Stelle auftreten. Wenn eine Struktur komplett eine andere Teilstruktur ersetzt, wird der Begriff "linear verschachtelt" verwendet. Gleichzeitig ist es möglich, dass Strukturen verschiedener Ebenen den gleichen Knoten verwenden, was als nichtlineare Verschachtelung bezeichnet wird. Beide Begriffe sind vom Autor selbst geprägt.

[6] Ein Speicherbereich, der direkt an die CPU angeschlossen ist und sehr viel schnelleren Zugriff erlaubt, allerdings in seiner Größe und Anzahl begrent ist. Jede Variation der Assemblersprache definiert eine bestimmte Anzahl sogenannter Universalregister, die auf bestimmte Weise benannt werden und spezifisch für die jeweilige Prozessorarchitektur sind. Für die x86-Prozessorarchitektur sind dies EAX, EBX, ECX, EDX, ESI, EDI, EBP und ESP. Diese sind 32 Bit breit, fassen also 4 Byte und können auch teilweise angesprochen werden. Dabei sind EBP und ESP mit einer besonderen Bedeutung versehen: Sie zeigen auf Anfang und Ende des aktuellen Stackframes und können nicht vom Programm anderweitig benutzt werden.

ings sind diese Operationen auch sehr ressourcenintensiv, weswegen seit den späten 1990er Jahren die entsprechenden Prozessorinstruktionen kaum noch verwendet wurden, zumal die Division durch Konstanten durch Multiplikation und Shifts ersetzt werden kann, was deutlich schnellere Ausführung ermöglicht[4], wodurch gerade Schleifen beschleunigt werden können. Gleichzeitig wird dadurch die eigentliche Bedeutung des Codes verschleiert, was zu Hürden bei der Analyse durch Menschen führt. Auch hier ist eine automatisierte Lösung also naheliegend, zudem noch nicht allgemein verbreitet[7]. Gleichzeitig ist die Berechnung des Divisors aus den vorliegenden Konstanten durchaus möglich und ausreichend beschrieben bzw. erklärt[5].

5 Umsetzung

Im Folgenden werden die verwendeten Algorithmen im Detail beschrieben, auch Informationen zur Implementation sind hier wiedergegeben.

5.1 Kontrollflussanalyse

Wie bereits in Abschnitt 4.2 beschrieben, ist die Kontrollflussanalyse eine verhältnismäßig komplexe Aufgabe, da das zu untersuhende Objekt eine Vielzahl an Formen annehmen kann, beispielsweise arrangieren Compiler den generierten Assemblercode nicht immer auf eine direkt erwartete Weise. Aus diesem Grund wurde im Laufe der Planung vorliegender Software sehr schnell klar, dass die Analyse des Programms auf Codeebene nicht erfolgreich sein kann, da die Strukturen kaum noch erkannt werden, wenn sie verschachtelt vorliegen. Eine Untersuchung der Fachliteratur ließ erkennen, dass die Analyse des Kontrollflusses deutlich effektiver ist, wenn dieser als Graph betrachtet wird. Per Definition ist ein Kontrollfussgraph (CFG) ein gerichteter Graph G, der über eine Menge Knoten V, eine Menge gerichteter Kanten E und einen Wurzelknoten r mit $r \in V$ verfügt [6].

$$G(V, E, r)$$

Aufgrund der Konzeption von Codeflusssteuerung auf Machienenebene kann zudem davon ausgegangen werden, dass jeder Knoten 0 bis 2 Nachfolger hat, allerdings eine beliebige Anzahl Vorgänger, wobei nur der Wurzelknoten keinen Vorgänger haben darf. Die Verwendung einer solchen Datenstruktur zur Representation des Programms hat eine Reihe von Vorteilen, wobei die wichtigsten eine deutlich einfachere Analyse möglicher Pfade vom Start- zum Endknoten und die Unabhängigkeit vom Codelayout durch den Compiler im Speicher sind. Die Software geht bei der Analyse folgendermaßen vor: Die Knoten werden nacheinander bearbeitet, wobei die Reihenfolge der Postorder-Traversierung des Tiefensuchbaumes des Graphen entspricht. Ist der aktuelle Knoten der Anfang einer simplen Struktur, die es zu erkennen gilt, werden die dazugehörigen Knoten und Kanten extrahiert und durch einen Knoten ersetzt, der Informationen über die enthaltene Struktur speichert, und welcher als nächstes analysiert wird. Sobald alle Knoten auf diese Weise behandelt wurden und alle notwendigen Graphenreduktionen stattgefunden haben, können die Informationen über erkannte Schleifen, Verzweigungen etc. in strukturierter Form dargestellt werden. Das aktuelle Featureset umfasst vor- und nachprüfende Schleifen, Verzweigungen mit beliebig vielen alternativen Pfaden und alle dem Autor bekannten Möglichkeiten der Verschachtelung. Seit neuestem werden auch komplexe Bedingungen in allen möglichen Kontexten erkannt und reduziert, was ein Novum darstellt, da der dazu verwendete Algorithmus bs jetzt nicht in der dem Autor bekannten Fachliteratur beschrieben wurde. Sprungbefehle wie goto, break, continue und in manchen Fällen return werden aufgrund der Herangehensweise nicht erkannt, wobei eine entsprechende Änderung in zukünftigen Versionen durchaus möglich ist.

5.1.1 Algorithmusbeschreibung und Implementationsdetails

Das Programm lässt sich in mehrere Abschnitte unterteilen, die sich verschiedenen Phasen der Analyse zuordnen lassen: Generieren des Graphen und die Analyse des Graphen selbst, die sich wiederum in zwei Abschnitte unterteilbar ist: Bestimmung der Analysereihenfolge der Knoten und die Analyse jedes einzelnen Knotens nach dieser Reihenfolge.

[7] Eine Recherche des Autors ergab, dass z.B. der Hex-Rays Decompiler "ohne Probleme" mit solchen Konstrukten umgehen kann, da dieser allerdings ein komerzielles und noch dazu recht teures Produkt darstellt, kann vermutlich nicht vonn allgemeiner Verbreitung gesprochen werden.

1. **Generierung des Graphen aus Assemblercode:** Die Unterteilung des Assemblerlistings erfolgt an sogenannten Trennpunkten zwischen Codezeilen. Diese erfüllen mindstens eine der nachfolgenden Bedingungen:

   (a) Die vorherige Instruktion ist ein bedingter oder unbedingter Sprung.

   (b) Die nachfolgende Instruktion mit einem Label versehen, also Ziel eines oder mehrer Sprünge.

   (c) Die nachfolgende Instruktion ist ein Vergleich.

   Wenn auf diese Weise alle Knoten des Graphen bestimmt sind, werden mögliche Übergänge von Knoten zu Knoten, also Kanten des Graphen, generiert, die *aktiv* oder *passiv* sein können. *Aktive* Kanten sind Kanten, die tatsächlich ausgeführten Sprüngen entsprechen, während *passive* Kanten keinem Sprung entsprechen, sondern einem Übergang zum nächsten Befehl, der in einem anderen Knoten liegt.

2. **Bestimmung der Analysereihenfolge für die Knoten des Graphen:** Hierfür wird zuerst der Tiefensuchbaum des in Schritt 1 erstellten Graphen berechnet.[8] Dieser wird dann per Postorder[9] traversiert. Die sich dabei ergebende Reihenfolge ist für die Analyse sehr gut geeignet, da auf diese Weise die innersten Strukturen zuerst erkannt und reduziert werden, so dass kein Knoten, der am Anfang einer Struktur steht, analysiert wird, während der Graph eine Erkennung der entsprechenden Struktur unmöglich macht.

3. **Analyse der einzelnen Knoten nach der in Schritt 2 bestimmten Reihenfolge:** Zuerst wird überprüft, ob der aktuelle Knoten Anfang einer simplen Struktur sein kann. Als simple Strukturen werden alle elementaren Konstrukte bezeichenet, die von Hochsprachen definiert werden: vor- und nachprüfende Schleifen, lineare Abfolgen beliebiger Strukturen und if-then / if-then-else -strukturen, also Bedingte Anweisungen. Das Programm unterscheided die folgenden Strukturen:

   (a) *if-then:* Eine bedingte Anweisung, ohne else-Klausel.

   (b) *if-then-else:* Eine bedingte Anweisung, mit else-Klausel.[10]

   (c) *while-loop:* Eine vorprüfende Schleife, entspricht sowohl der while- als auch der for-Schleife, da diese im Grunde genommen identisch vom Compiler behandelt werden.

   (d) *do-loop:* Eine nachprüfende Schleife.

   (e) *condition:* Eine komplexe Bedingung.

   (f) *block:* Eine beliebig lange, lineare Abfolge von sowohl primitiven, als auch Struktur-Knoten, ausgenommen Bedingungen.[11]

   Ist eine solche Sruktur aufzufinden, werden ihre Teile aus dem Graphen entfernt und ein Knoten wird in den Graphen an entsprechender Stelle eingefügt, der die Informationen über sie speichert. Sollte dies jedoch nicht der Fall sein, wird überprüft, ob zuerst eine komplexe Bedingung reduziert werden kann, falls das möglich ist, wird danach ein erneuter Versuch einer Strukturreduktion durchgeführt. Sollte eine Struktur gefunden worden sein, wird ihr Knoten vor dem nächsten Knoten in der Reihenfolge aus Schritt 2 betrachtet, indem die entsprechende Analysefunktion rekursiv ausgeführt wird. Konkret werden komplexe Bedingungen folgendermaßen untersucht: Zunächst wird überprüft, ob es sich bei der hypothetischen Bedingung um die Bedingung einer vorprüfenden Schleife handeln kann.

[8] Ein Tiefensuchbaum oder Depth-First-Search-Tree (DFS-Tree) wird generiert, indem ein Graph per Tiefensuche traversiert wird und die dabei besuchten Knoten auf eine bestimmte Weise in einem Baum abgelegt werden. Eine genauere Beschreibung des Algorithmus kann unter [7] eingesehen werden.

[9] Die Postorder-Traversierung ist eine Methode, für einen Baum eine eindeutige Reihenfolge der Knoten zu bestimmen. Dabei handelt es sich um einen rekursiven Algorithmus, der auf jeden Knoten Teilbaum im Baum angewendet wird. Dabei wird das Ergebnis der Postorder-Traversierung der Kindknoten zuerst an die Liste der Knoten angefügt, danach der eigentliche Knoten. Auf diese Weise wird der ganze Baum durchlaufen, wobei die Wurzel immer das letzte Element der Ergebnismenge darstellt. Mehr Details sind unter [8] zu finden.

[10] Bedingte Anweisungen, die mindestens ein *else if* enthalten, werden als verschachtelte *if-then-else* bzw. *if-then*-Srukturen angesehen, da dies die Analyse deutlich vereinfacht und zudem für den Compiler keinen Unterschied darstellt.

[11] Tatsächlich kann es vorkommen, dass Bedingungen direkt in einem Block platziert werden, wenn dieser Block der einzige Knoten in einer nachprüfenden Schleife ist. Dies liegt an der Art, auf die nachprüfende Schleifen erkannt werden und wird *nicht* als Bug angesehen.

Dies wird über eine Suche nach Backedges, die den aktuellen Knoten zum Ziel haben, bewerkstelligt. Sollten solche Knoten vorhanden sein, werden alle Pfade (s. u), die diesen Knoten enthalten, aus der Analyse ausgeschlossen. Danach werden die möglichen Pfade für den Kontrollfluss ausgehend vom aktuellen Knoten berechnet. Nun wird nach einem Knoten mit speziellen Eigenschaften gesucht: Dem ersten gemeinsamen Knoten aller gefundenen Pfade, der nicht dem Anfangsknoten entspricht. Sollte es hingegen so sein, dass eine Schleifenbedingung betrachtet wird, werden nur die Pfade für die Bestimmung des Knotens herangezogen, die nicht den Schleifenkörper erreichen und dementsprechend keine Backedges enthalten, wie anfangs beschrieben. Der gefundene Knoten ist nicht mehr zur Struktur und Bedingung gehörig. Die Knoten, die vor ihm in den Pfaden auftreten, gehören hingegen dazu, werden anhand der Anzahl ihrer Nachfolger als Bedingungen oder Code identifiziert. Die Knoten, die zur Bedingung gehören, also zwei Nach- folger besitzen, werden zu einem Knoten reduziert, der die komplexe Bedingung representiert. Dies geschieht jedoch nur, wenn keiner der Knoten in der Bedingungstruktur, außer der Anfangsknoten, Vorgänger außerhalb der Struktur besitzt. Auf diese Weise wird verhindert, dass Teile von großen und komplexen Bedingungen als Strukturen reduziert werden, was die Reduktion sehr unperformant und nicht gerade nützlich machen würde. Derzeit ist Code in der Entwicklung, der die Analyse von bereits reduzierten Bedingungen übernehmen soll. Dies würde es ermöglichen, logische Verknüpfungen zwischen den Teilbedingungen zu erkennen und ein weiteres arbeitsintensives Detail des Reverse-Engineerings zu automatisieren.

5.2 Datenanalyse

Bei diesem Modul liegt der Hauptaugenmerk auf der Möglichkeit, nicht nur addressierte Speicherbereiche aus dem Code zu extrahieren, wie es z.B. in IDA Pro geschieht, sondern auch aus dem Kontext der Speicheraufrufe auf die Funktion der entsprechenden Speicherabschnitte zu schließen. Dabei sei erwähnt, dass solche Verfahren grundsätzlich nur Heuristiken darstellen, da nicht jeder Speicherabschnitt, aus dem z.B. ein DWORD[12] gelesen wird, automatisch einer Integervariable gleichgesetzt werden kann. So verwenden moderne Compiler die push[13]- und pop[14]-Instruktionen nicht mehr für Funktionsaufrufe, da diese zu viele Ressourcen verbrauchen. Stattdessen wird am Anfang der Funktion mehr Speicher alloziert als für die lokalen Variablen notwendig, so dass Parameter direkt in den Speicher geschrieben werden, bevor eine Funktion aufgerufen wird. Darüber hinaus werden vermehrt sogenannte Canarys auf dem Stack hinterlegt, die sicherstellen sollen, dass kein Stackoverflow stattfindet, bzw. Schäden am Stackframe direkt erkannt werden. Dabei handelt sich um "freien" Speicher, der zwischen den lokalen Variablen und dem für Parameter reservierten Platz alloziert wird und auf seine Integrität überprüft wird, wenn die Funktion zurückkehrt. Auf diese Weise wird es viel schwieriger, die Rückkehradresse durch Nutzereingaben zu verändern. Solche Canary-Werte können auch zwischen Arrays und anderen Variablen eingefügt werden, was z.B. die Längenbestimmung von Arrays erschwert, zumal bei diesen selten direkte Addressierungen der Elemente stattfinden, was es sehr schwierig macht, ihre Grenzen genau zu bestimmen. Diese und andere Aspekte der Kompilierung, bei denen Informationen verlorengehen, erschweren die Analyse und Wiederherstellung mitunter sehr, was gleichermaßen für Menschen und Programme gilt, wobei letztere weniger trugschlussanfällig zu sein scheinen, da sie das Analyseergebnis schrittweise aufbauen, so dass mitunter Informationen nur unvollständig sind, jedoch nur selten objektiv falsch.

5.3 Optimisierte Intergerdivision

Die derzeitige Umsetzung dieses Moduls basiert auf praktischen Erfahrungen des Autors und von Nutzern der Reverse-Engineering-Stackexchange Website [5].

[12] 4 byte (32 bit) großer Speicherabschnitt, meist ein Integer

[13] push alloziert Speicher auf dem Stack und speichert dort den übergebenen Operanden

[14] pop entfernt den zuletzt auf den Stack geschobenen Wert und speichert ihn im übergebenen Operanden

5.3.1 Herleitung des Algorithmus

Aus den Ausführungen in [5] folgt, dass folgende Formel zum Wiederherstellen des Divisors benutzbar ist:

$$d = \left\lceil \frac{2^{bitness+rshift}}{x + 2^{bitness}} \right\rceil$$

Die mathematische Herleitung ist verhältnismäßig verworren und nicht gerade klar, weswegen sie hier, um Fehler zu vermeiden, nicht wiedergegeben wird. Dabei ist d der gesuchte Divisor, *bitness* ist die Breite der verwendeten Register, $rshift$ die Anzahl Stellen, um die das Ergebnis verschoben wird. Folglich bleibt es dem Code noch übrig diese Werte aus dem Assebly-Listing zu extrahieren. Um den genauen Ablauf dieses Prozesses zu steuern, werden Grenzwerte festgelegt, wie weit die einzelnen Instruktionen voneinander entfernt sein dürfen, damit der gegebene Codeabschnitt noch als Division betrachtet wird. Sollten die automatisch verwendeten Werte nicht zum gewünschten Ergebnis führen, kann der sogenannte *interaktive Modus* verwendet werden, der den Nutzer die Konstanten von Hand eingeben lässt und das Ergebnis der Berechnungen ausgibt.

6 Ausblick

Trotz aller in den vorherigen Abschnitten beschriebenen Vorzügen des Projekts kann nicht von einem kompletten und fehlerfreien Produkt gesprochen werden. Konkret stehen noch folgende Punkte aus:

- Die Dokumentation muss verbessert werden.
- Die Analyse reduzierter Bedingungen muss umgesetzt werden.
- Das CFG-Modul muss restrukturiert werden, um es stylistisch zu verbessern.
- Gegebenenfalls sollte eine Unterstützung von *break*, *continue*, *goto* und *return* in die Kontrollflussanalyse eingebunden werden.

Gleichzeitig ist das Projekt durchaus recht weit seiner Fertigstellung fortgeschritten, unter anderem, da alle Features einem sehr gründlichen System an Testläufen unterzogen wurden, die auch auf Integration mit dem Restsystem ausgerichtet sind, was bedeutet, dass Tests, die einzelnen Bausteinen eines Moduls gelten so konzipiet sind, dass aus ihren Ergebnissen auf das Verhalten des Systems in veränderten Kontexten geschlossen werden kann. Auf diese Weise wird verhndert, dass einzelne Fehler in bestimmten Eingabekonfigurationen auftreten können. Dies ist besonders für das Kontrollflussanalysemodul wichtig, da dieses mit Abstand am komplexesten ist, weswegen auch zukünftige Erweiterungen sehr streng auf ihre Interoperabilität geprüft werden müssen.

Bibliotheksreferenz

[1] http://en.wikibooks.org/wiki/X86_Assembly. Eingesehen am 08.02.2015.

[2] http://en.wikibooks.org/wiki/X86_Disassembly. Eingesehen am 08.02.2015.

[3] Misha Sharir. Structural analysis: A new approach to flow analysis in optimizing compilers. Technical report, New York University, Department of Computer Sciences, 1979.

[4] Torbjörn Granlund and Peter L. Montgomery. Division by invariant integers using multiplication. 1991.

[5] http://reverseengineering.stackexchange.com/questions/1397/how-can-i-reverse-optimized-integer-division-modulo-by-constant-operations. Eingesehen am 15.11.2014.

[6] http://de.wikipedia.org/wiki/Kontrollflussgraph. Eingesehen am 4.11.2014.

[7] http://de.wikipedia.org/wiki/Tiefensuche. Eingesehen am 31.01.2015.

[8] http://de.wikipedia.org/wiki/Binärbaum. Eingesehen am 31.01.2015.