# Besondere Lernleistung - Dokumentation

Inokentiy Babushkin

December 23, 2014

## 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 Problemantik

Da moderne Software immer kompexer wird, wird es entsprechend schwieriger, in binärer Form vorliegende Softwareprodukte so umzuformen, dass sie ein tiefgehendes Verständnis und Modifikation erlauben. Aus diesem Grund werden bereits seit einiger Zeit Ansätze unternommen, diesen Vorgang so weit wie möglich zu automatisieren, da die Wiederherstellung von Kontrollfluss, Variablen, Funktionen und weiteren HLL[1]-spezifischen Strukturen ohnehin nicht komplett möglich ist, das Ergebnis also bloß eine Annäherung an den Originalcode sein kann, obwohl der Vorgang - von Menschen durchgeführt - sehr zeitraubend ist. 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: Decompiler, deren Ziel es ist, ein Programm komplett wiederherzustellen, die meistens eine non-interaktive Arbeitsweise erfordern, und Programme, die bestimmte Aspekte des zu analysierenden Codes wiederherstellen oder eine Art Anleitung für den Benutzer generieren, wie er dabei am besten verfährt. Das Resultat der vorliegenden Besonderen Lernleistung ist in die zweite Kategorie einzuordnen, wobei es aus drei Modulen besteht, die unabhängig arbeiten und verschiedene Bereiche des Analysegebiets abdecken.

## 3 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[2]-Aufgaben mit seiner Hilfe vereinfacht werden kann. Dabei wird die Ausgabe von IDA Pro 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 Verarbeitung der Eingabe erfordert, um die für die Programmmechanik notwendigen Informationen bereitzustellen.

### 3.1 Kontrollflussanalyse

Einer der zentralen Aspekte moderner wie historischer Programme ist die Möglichkeit, Code in Abhängigkeit von Bedingungen ausführen zu können. Diese Beziehungen wiederherstellen zu können ist eine der Hauptaufgaben dieses Projekts. Dabei ist diese Zielsetzung keineswegs trivial, zumal

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

[2] 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

[3] Strukturen in einem CFG bestehen in der Regel aus zwei oder Mehr Teilen, wobei sogenannte primitive Teile einfache Knoten des Graphen sind. In der Praxis können jedoch auch nich-primitive Teile, also andere Strukturen an dieser Stelle auftreten. Wenn eine Struktur komplett einen Teil 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.

Strukturen linear und nicht linear verschachtelt[3] sein können etc. Als Grundlage des verwendeten Algorithmus diente die in (1) beschriebene Vorgehensweise.

### 3.2 Analyse von Variablen und anderen Daten

Daten bilden den zweiten Hauptbestandteil eines Programms, weswegen das Layout der Variablen, Argumente/Parameter, Rückgabewerte von großem Interesse für den Benutzer sein sollte. Gleichzeitig sind Arrays und Variablen nicht immer trivial zu erkennen, weswegen sich einige einfache, dennoch effektive Heuristiken anbieten.

### 3.3 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. Allerdings sind diese Operationen auch sehr ressourcenintensiv, weswegen seit den späten 1990er Jahren die entsprechenden Prozessorinstruktionen kaum noch verwendet werden, zumal die Division durch Konstanten durch Multiplikation und Shifts ersetzt werden kann, was deutlich schnellere Operationen sind(2), wodurch die Ausführung gerade in Schleifen beschleunigt werden kann. 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[4].

## 4 Umsetzung

Im Folgenden werden die verwendeten Algorithmen im Detail beschrieben.

### 4.1 Kontrollflussanalyse

Wie bereits in Abschnitt 3.1 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 (3).

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

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.

[4] 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.

#### 4.1.1 Algorithmusbeschreibung und Implementationsdetails

Das Programm lässt sich in mehrere Abschnitte unterteilen:

1. Generierung des Graphen aus Assemblercode: Die Unterteilung des Assemblerlistings erfolgt an sogenannten Trennpunkten zwischen Codezeilen. Diese sind dadurch gekennzeichnet, dass entweder die vorherige Zeile mehrere Nachfolger besitzt, also ein bedingter Sprung ist, oder die nächste Zeile über Sprünge direkt erreicht werden kann, also ein Label ist. Diese Fälle können gleichzeitig auftreten. Um besser Code von Bedingungen unterscheiden zu können, wird das Listing zusätzlich an den Stellen aufgeteilt, wo die nachfolgende Zeile ein Vergleich ist, was im Verlauf der Analyse teilweise rückgängig gemacht wird. Sobald 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 dem nicht-Ausführen eines bedingten Sprungs oder einem anderen Übergang zum nächsten Befehl.

2. Bestimmung der Analysereihenfolge für die Knoten des Graphen: Hierfür wird zuerst der Tiefensuchbaum des in Schritt 1 erstellten Graphen berechnet. Dieser wird dann per Postorder traversiert. Die sich dabei ergebende Reihenfolge der Knoten wird für die Analyse verwendet.

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 elementare Konstrukte bezeichenet, die von Hochsprachen definiert werden: vor- und nachprüfende Schleifen, lineare Abfolgen beliebiger Strukturen und if-then / if-then-else Konstrukte. 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 das nicht der Fall sein, wird überprüft, ob zuerst eine komplexe Bedingung reduziert werden kann, an diese Analyse schließt sich der Versuch einer Strukturreduktion angefügt. Sollte eine Struktur gefunden worden sein, wird ihr Knoten vor dem nächsten Knoten in der Reihenfolge aus Schritt 2 betrachtet. Konkret werden komplexe Bedingungen folgendermaßen untersucht: Zunächst wird überprüft, ob es sich bei der hypothetischen Bedingung um die Bedingung einer vorprüfenden Schleife handelt. 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. 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 und Code identifiziert, die Knoten, die zur Bedingung gehören, werden zu einem Knoten reduziert, was allerdings nur geschieht, wenn keiner der Knoten in der Struktur, außer der Anfangsknoten, Vorgänger außerhalb der Struktur besitzt. Auf diese Weise werden versehentlich keine Teilstrukturen reduziert.

### 4.2 Datenanalyse

Bei diesem Modul liegt der Hauptaugenmerk auf der Möglichkeit, nicht nur addressierte Speicherbereiche aus einer Analyse des Codes 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[5] gelesen wird, automatisch einer Integervariable gleichgesetzt werden kann. So verwenden moderne Compiler die push[6]- und pop[7]-Instruktionen nicht mehr für Funktionsaufrufe, da diese zu viele

[5] 4 byte (32 bit) großer Speicherabschnitt, meist ein Integer
[6] push alloziert Speicher auf dem Stack und speichert dort den übergebenen Operanden
[7] pop entfernt den zuletzt auf den Stack geschobenen Wert und speichert ihn im übergebenen Operanden

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. 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. 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.

### 4.3 Optimisierte Intergerdivision

Die derzeitige Umsetzung dieses Moduls basiert auf praktischen Erfahrungen des Autors und von Nutzern der Reverse-Engineering-Stackexchange Website(4).

#### 4.3.1 Herleitung des Algorithmus

Essentiell für das Verständnis, wie der Computer bei optimisierter Division rechnet, ist es wichtig zu verstehen, dass Division durch die Multiplikation mit dem Kehrwert esetzt werden kann:

$$\frac{n}{r} = n * \frac{1}{r}$$

Dabei ist es allerdings nicht möglich, einen Bruch $\frac{1}{r}$ mit $r \ni \mathbb{N}$ als einen Integer zu speichern, weswegen es nahe liegt, beide Werte mit einer Konstante zu multiplizieren. In der Regel wird $2^{32}$ als Faktor verwendet, da dies gut zur verwendeten Registergröße passt.

Folglich kann der folgende Zusammenhang verwendet werden:

$$d = \frac{y+x}{r} * \frac{n}{y}, x = r - (y \mod r)$$

Dabei ist $y$ der verwendete Faktor, hier $2^{32}$, wobei dies eigentlich jede Zahl $2^n$ sein kann, allerdings ist die verwendete Konstante - wie bereits erwähnt - aufgrund der 32-Bit breiten Register besonders bequem, $x$ wir die sogenannte "Magic number" darstellen, die der Compiler als Konstante verwenden wird, $n$ ist die Variable, die dividiert werden soll. Aus diesen Ausführngen folgt:

$$d = \frac{n}{r} + \frac{xn}{ry}$$

Wobei der zweite Summand die Abweichung, die möglichst klein werden soll, darstellt.

## Bibliotheksreferenz


[1] M. Sharir, "Structural analysis: A new approach to flow analysis in optimizing compilers,"

[2] T. Granlund and P. L. Montgomery, "Division by invariant integers using multiplication,"

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

[4] http://reverseengineering.stackexchange.com/questions/1 can-i-reverse-optimized-integer-division-modulo-by-constant-operations. Eingesehen am 15.11.2014.