Proseminar

Reverse Engineering – Java Decompiler

Jan Möller

31. Januar 2014

Technische Universität Dortmund Fakultät Informatik

Lehrstuhl 14 Software Engineering – Prof. Dr. Jan Jürjens

betreut durch: Sebastian Pape

Jan Möller

Jan.Moeller2@tu-dortmund.de
Matrikelnummer: 147674
Studiengang: Bachelor Angewandte Informatik

Proseminar
Thema: Reverse Engineering – Java Decompiler

Eingereicht: 31. Januar 2014

Betreuer: Sebastian Pape

Prof. Dr. Jan Jürjens Lehrstuhl 14 Software Engineering
Fakultät Informatik
Technische Universität Dortmund
Otto-Hahn-Straße 14
44227 Dortmund

Ehrenwörtliche Erklärung

Ich erkläre hiermit ehrenwörtlich, dass ich die vorliegende Arbeit selbstständig angefertigt habe. Sämtliche aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht.

Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und noch nicht veröffentlicht.

Dortmund, den 31. Januar 2014 _____Jan Möller_____

Unterschrift

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Wofür „Reverse Engineering“?

1.1 Motivation

Reverse Engineering (engl. rekonstruieren) beschreibt das Vorgehen, ein fertiges System oder Endprodukt systematisch zu zerlegen, um damit Erkenntnisse über den ursprünglichen Konstruktionsvorgang und eingesetzte Techniken bzw. Materialien zu gewinnen [WIK13]. Dieser Vorgang kann in unterschiedlichen Gebieten zum Einsatz kommen, wie zum Beispiel bei montierten Maschinen oder auch bei Hard- und Software. Während bei Maschinen und Hardware das Rekonstruieren mit einem höheren Aufwand verbunden ist (exaktes Zerlegen der Einzelteile), kann aus Software, wenn der Entwickler nicht auf ein besonderes Gegenwirken achtet, teilweise mühelos der ursprüngliche Quelltext gewonnen werden [KEP10]. Genau diese Leichtigkeit stellt ein hohes Sicherheitsrisiko dar, da mögliche Konkurrenz programmierte Technologien kopieren und somit geistiges Eigentum stehlen oder auch manipulieren kann, um Schaden zu verursachen.

Das Thema kann jedoch auch ohne einen böswilligen Hintergedanken zum Einsatz kommen. Man stelle sich ein Unternehmen vor, welches im gesamten Vertrieb eine bestimmte Software verwendet, welche nur von dem Entwickler betreut wird. Bei einem Szenario in dem dieser Entwickler nun insolvent geht kann die Software jedoch nicht mehr gewartet werden. In diesem Falle benötigt das Unternehmen Zugriff auf die Programmieranweisungen um zum Beispiel eigenständig Sicherheitslücken zu beheben. Ebenso verwenden Hersteller von Antivirensoftware Decompiler, um die Verhaltensweisen von Computer Viren zu analysieren und Gegenmaßnahmen zu entwerfen.

Welche Möglichkeiten und Vorgehensweise es zur Rückgewinnung von Quellcode vor allem im Hinblick auf die oft genutzte Programmiersprache Java gibt, wird in dieser Arbeit beschrieben. Ebenso werden Alternativen aufgezeigt, mit denen sich ein Entwickler vor Manipulation und Diebstahl schützen kann.

1.2 Compiler – Decompiler

Um die Struktur einer Programmiersprache zu verstehen muss zunächst der Zusammenbau einer solchen aufgezeigt werden. Diese Grundlage ist nötig, um die Voraussetzungen einer Rekonstruktion von Quellcode greifbar zu machen.

Um eine Anwendung zu programmieren schreibt der Entwickler den Quellcode, bestehend aus Funktionen, Definitionen und Anweisungen in Form von Klartext auf, welcher der Syntax der gewählten höheren Programmiersprache entspricht. Dieser Programmcode ist an dieser Stelle noch für Menschen lesbar nicht jedoch für Maschinen. Um diesen Code nun in

Maschinensprache umzuwandeln, sodass daraus ein ausführbares Programm wird, ist ein sogenannter „Compiler“ (engl. to compile – zusammenstellen) notwendig. Ein Compiler übersetzt eine Programmiersprache in Maschinensprache [DUD13].

Das Gegenstück zu einem Compiler ist der „Decompiler“. Der Zweck dieses Programms besteht darin, den Vorgang eines normalen Compilers umzudrehen und somit aus Maschinencode erneut den Quellcode einer höheren Programmiersprache zu gewinnen [KEP10].

Ein möglicher Decompiler von Javacode ist der Java Decompiler, auf den im späteren Verlauf dieser Arbeit genauer eingegangen wird.

2 Die Architektur von Java

2.1 Struktur von der Java Virtual Machine

In dieser Arbeit wird der Fokus des Reverse Engineering auf die Programmiersprache Java gesetzt. Java als Programmiersprache wird heutzutage auf vielen unterschiedlichen Gerätetypen verwendet, dessen Grund vor allem in der besonderen Struktur – dem Bytecode für die spezielle Laufzeitumgebung – zu finden ist. Anders als eine systemnahe Programmiersprache wie zum Beispiel „C“ muss für die Ausführung von kompilierten Java Programmen eine weitere Software installiert sein, die sogenannte „Java Runtime Environments“ (kurz JRE, dt.: „Java Laufzeit Umgebung“). Diese Programmgrundlage hat auf der einen Seite den Nachteil, dass ein möglicher Nutzer von Java Anwendungen mehr Zeit zum endgültigen Starten benötigt, da zunächst die „JRE“ installiert werden muss, bevor ein Programm ausführbar wird. Auf der anderen Seite hat diese Laufzeit Umgebung jedoch den erheblichen Vorteil der Plattformunabhängigkeit, da Java Programme ohne größere Umwege direkt auf unterschiedlichen Systemen lauffähig sind, sofern Java entsprechend verfügbar bzw. portiert ist.

Die Java Virtual Machine (JVM) fungiert als ein Adapter. Sie führt die Java Programme aus und übersetzt den Javacode in die eigentliche Maschinensprache. Damit dies möglich wird, übersetzt der Java Compiler den Quelltext eines Programms in den sogenannten „Java Bytecode“ [SOI13].

Genau dieser Zwischenschritt ermöglicht jedoch einfachere Ansätze des Reverse Engineerings, als dass es der eigentliche Maschinencode tut. Wie dies genau funktioniert, wird im nächsten Kapitel aufgezeigt [K04].

2.2 Kompilieren für die JVM

Wie bereits in dem vorhergehenden Abschnitt erwähnt wird Java Quelltext mit dem Java Compiler in den „Bytecode“ übersetzt und ist damit auch lesbar für die JVM. Dieser Bytecode wird in Form von „.class“ Dateien generiert, welche im weiteren Sinne Anweisungen für die Laufzeit Umgebung beinhalten.

Eine Bytecode Anweisung hat im Allgemeinen die folgende Form:

<index> <opcode> [<operand1> [<operand2> …]] [<comment>]

Abbildung 2.1: Bytecode Aufbau

Der Platzhalter „index“ steht hierbei für eine Art „byte offset“[1] von dem Beginn einer Methode. Opcode steht für eine Abkürzung, die für den Menschen einfach zu lesen ist und einen Befehl des Java Bytecode präsentiert. Zusätzlich können n – Operanden übergeben werden, je nach Verwendung sehr wenige oder auch beliebig viele. Der letzte Platzhalter „comment“ ist optional und bietet die Möglichkeit Kommentare zu der Anweisung abzulegen, welche vom Compiler erzeugt werden [O13].

Folgendes Beispiel zeigt den Bytecode einer Java Methode mit der Signatur „public void start()“, welche von einer einfachen Stoppuhr Klasse den Zähler auslöst. Hierbei wird die aktuelle Systemzeit abgefragt und in einer Variablen gespeichert.

```
0: aload_0
1: invokestatic #17;   //Method java/lang/System.currentTimeMillis:<>J
4: putfield       #23;  //Field start:J
7: return
```

Abbildung 2.2: Beispiel Methode in Bytecode

Der Start der Methode wird mit „aload_0“ eingeleitet und das Ende mit einem „return“ definiert. Zusätzlich ist der Aufruf der Java Funktion zum Abrufen der aktuellen Systemzeit sichtbar, welcher durch den automatisch generierten Kommentar deutlich wird. Ebenso wird die Variablenbelegung mit „putfield“ dargestellt.

[1] Zu der Basisadresse wird ein Versatz („offset“) addiert. Die Summe ergibt dann die tatsächliche Speicheradresse.

3 Bytecode Analyse

3.1 Visualisierung von Bytecode und Funktionen

Um einen Ansatz für das Reverse Engineering zu haben, muss der Java Bytecode einer kompilierten Java Klasse aus der „.class Datei“ sichtbar und somit greifbar gemacht werden. Dies funktioniert mit einer Anwendung von Java, die bereits in dem Java Development Kit (kurz JDK) mitgeliefert wird. Diese findet sich in dem entsprechenden Installationsordner unter dem Dateipfad „bin/javap.exe“ [O13].

Mit einem kurzen Kommandozeilen Befehl (unter Windows „javap –c <ClassName>“) wird der Bytecode offenbart. Im folgenden Kapitel soll der genaue Vorgang zur Rekonstruktion von Java Quelltext anhand einer einfachen Beispielklasse Schritt für Schritt nachvollzogen werden. Dafür wurde eine Klasse „Stoppuhr“ angelegt, die Methoden zum Starten und Stoppen eines Zeitabschnitts und zur Rückgabe des Zeitraums bereitstellt. Im Anschluss des Reverse Engineering Vorgangs wird der Ausgangscode gezeigt, um einen Vergleich zu den Ergebnissen zu ermöglichen und etwaige Abweichungen aufzudecken und anzusprechen.

```
C:\Program Files (x86)\Java\jdk1.6.0_38\bin>javap -c Stoppuhr
Compiled from "Stoppuhr.java"
public class Hilfsfunktionen.Stoppuhr extends java.lang.Object{
public Hilfsfunktionen.Stoppuhr();
  Code:
   0:   aload_0
   1:   invokespecial   #11; //Method java/lang/Object."<init>":()V
   4:   return

public void start();
  Code:
   0:   aload_0
   1:   invokestatic    #17; //Method java/lang/System.currentTimeMillis:()J
   4:   putfield        #23; //Field start:J
   7:   return

public void end();
  Code:
   0:   aload_0
   1:   invokestatic    #17; //Method java/lang/System.currentTimeMillis:()J
   4:   putfield        #26; //Field ende:J
   7:   return

public long result();
  Code:
   0:   aload_0
   1:   getfield        #26; //Field ende:J
   4:   aload_0
   5:   getfield        #23; //Field start:J
   8:   lsub
   9:   lreturn
```

Abbildung 3.1: Beispielklasse in Bytecode

Die Visualisierung zeigt nun die beinhalteten Konstruktoren, Methoden und referenzierte Klassen an. Hierbei fällt bereits auf, dass die ursprünglich vergebenen Bezeichner des Entwicklers in Bytecode vollständig bestehen bleiben.

An dieser Stelle wurden also bereits die ersten groben Informationen einer kompilierten Klasse rekonstruiert. Man kann durch diese Informationen festhalten, dass die Klasse eine Unterklasse der allgemeinen „Object Class“ und dem Package „Hilfsfunktionen“ zugeordnet ist, einen Konstruktor beinhaltet und sich aus drei Methoden mit den Signaturen „public void start()“, „public void end()“ und „public long result()“ zusammensetzt.

Die Kommentarbereiche der einzelnen Bytecode Anweisungen geben zudem erste Erkenntnisse über die verwendeten Funktionen anderer Java Klassen, in diesem Falle der Zeitabfrage über „System.currentTimeMillis()“. Zudem offenbart der Kommentarbereich erste verwendete Variablennamen, wie man an dem Feld #26 mit dem Bezeichner „ende“ und dem Feld #23 mit der Kennzeichnung „start“ erkennen kann. Darüber hinaus wird durch die „lreturn“ Anweisung der Methode „result()“, welche sich von dem normalen „return“ Methodenabbruch unterscheidet, sichtbar, dass hier ein Objekt von dem Datentyp „long“ zurückgegeben wird. Neben den hier gesehenen „return“ und „lreturn“ gibt es in dem Java Bytecode Befehlssatz noch eine ganze Reihe weiterer Methodenendpunkte, wie zum Beispiel „ireturn“ für einen Integer oder auch „areturn“ für eine Objektreferenz als Rückgabewert.

Die weiteren Befehle aus dieser Beispielklasse haben folgende Funktionen:

aload_<n>	Lädt die Referenz einer lokalen Variable, wobei <n> der Index zu dem lokalen Variablenfeld des aktuellen Bereichs ist.
invokespecial	Befehl um eine Instanz aufzurufen. Wird hierbei für den Konstruktor der Klasse „Stoppuhr“ benötigt.
invokestatic	Funktion um eine statische Klassenmethode aufzurufen. In dem Beispiel wird die aktuelle Systemzeit abgefragt.
putfield	Setzt den Wert eines Feldes. Diese Funktion wird genutzt um die „start“ und „ende“ Variable zu setzen.
getfield	Liefert den Wert eines Feldes. In der Methode „result()“ wird dies genutzt, um an die gemessenen Werte zu gelangen.
lsub	Subtrahiert zwei Werte des Datentyps „long“. Wie bei „return“ gibt es auch hierbei unterschiedliche „sub“ Befehle (zB. „isub“ für Integer). In der Beispielklasse wird hiermit die Differenz zwischen dem Start und Ende berechnet.

Tabelle 3.1: Befehle und deren Bedeutungen aus der Beispielklasse [O13]

Nachdem nun die einzelnen Befehle der Methoden von Hand analysiert wurden, liegt ein grober Bauplan der Vorgehensweise von den einzelnen Methoden vor. Durch die

gesammelten Informationen lässt sich eine Klasse „Stoppuhr“ mit den gleichen funktionalen Möglichkeiten erstellen, ohne den eigentlichen Quelltext zu kennen. Es ist sogar möglich, die ursprünglichen Namen der Methoden und Variablen zu verwenden und sich damit eine bessere Vorstellung der letztlichen Bedeutungen zu schaffen. Dies ist vor allem für größere und umfangreichere Klassen nötig.

```
package Hilfsfunktionen;

public class Stoppuhr {

    private long start, ende;

    public void start()
    {
        this.start = System.currentTimeMillis();
    }

    public void end()
    {
        this.ende = System.currentTimeMillis();
    }

    public long result()
    {
        return ende-start;
    }
}
```

Abbildung 3.2: Ausgangsklasse "Stoppuhr"

Im Gegensatz zu der sehr einfach gehaltenen Beispielklasse bestehen größere Projekte, vor allem die von denen sich mögliche Konkurrenten einen Vorteil durch Reverse Engineering erhoffen, aus deutlich komplexeren Strukturen ohne einen strikt linearen Kontrollfluss. Funktionen bestehen oft aus (zum Teil mehrfach verschachtelten) Schleifen, Fallunterscheidungen und mehreren möglichen Abbruchpunkten. Durch solche Konstrukte wird die Analyse des Bytecodes erschwert, da nicht jede Zeile direkt durch die entsprechende Bedeutung des Befehls ersetzt werden kann. Der Reverse Engineerer muss also die Logik von Sprüngen und den dahinter verborgenen Schleifen erkennen.

Welche Vorgehensweisen zur erweiterten Kontrollflussanalyse es gibt, wird in dem folgenden Abschnitt thematisiert und erklärt.

3.2 Zurückgewinnung des Kontrollflusses

Den Kontrollfluss eines linearen Programms zurück zu gewinnen gestaltet sich um einiges leichter als bei Anwendungen mit Verzweigungen und Sprüngen innerhalb der Anweisungen.

Im besten Fall kann einfach jeder Bytecode Befehl in die entsprechende Java Methode transformiert werden. Verzweigungen und Wiederholungen stellen bei der Rekonstruktion von Java Quelltexten bereits eine Hürde dar, da es keine direkten Entsprechungen in dem Bytecode Befehlssatz gibt. Es bedarf daher einer genauen Analyse des kompilierten Codes, um solche Abweichungen des linearen Ablaufs zu entdecken.
Als erster Indikator einer Schleife oder Verzweigung ist der „goto <index>“ Befehl zu verstehen, welcher einen Sprung (ähnlich wie bei Assembler) zu einer bestimmten Codezeile darstellt.

3.2.1 Schleifen

Der folgende Bytecode Ausschnitt zeigt eine Java For-Schleife mit einer Zählervariablen.

```
public void schleife();
  Code:
    0:   iconst_0
    1:   istore_1
    2:   goto    15
    5:   getstatic       #15; //Field java/lang/System.out:Ljava/io/PrintStream;
    8:   iload_1
    9:   invokevirtual   #21; //Method java/io/PrintStream.println:(I)V
    12:  iinc    1, 1
    15:  iload_1
    16:  iconst_3
    17:  if_icmplt       5
    20:  return
```

Abbildung 3.3: Schleife in Bytecode

Die einzelnen Befehle haben folgende Bedeutungen:

Index 0 und 16: iconst_0 / inconst_3	Platziert einen Integer mit dem Wert „0“ (bzw. „3“) oben auf dem Variablen Stack.
Index 1: istore_1	Speichert den Integer vor obersten Stelle des Stacks in Variable Nummer 1.
Index 2: goto 15	Springt zu der Zeile mit dem Index 15 und dabei überspringt Index 5 bis 12.
Index 8 und 15: iload_1	Liest den aktuellen Wert der Variable Nummer 1 und platziert diesen auf dem Stack an oberster Stelle.
Index 12: iinc 1, 1 (iinc <index>,<const>)	Erhöht die Variable Nummer 1 um den Wert „1“.
Index 17: if_icmplt 5	Vergleicht die zwei obersten Variablen auf dem Stack. Falls die erste kleiner ist als die zweite, so wird zu der Zeile mit dem Index „5“ gesprungen.

Tabelle 3.2: Bedeutungen der Bytecode Befehle einer Schleife [O13]

Der eigentliche „Kopf“ der For-Schleife wird in den Zeilen mit den Index Nummern 12 bis 17 bearbeitet. An dieser Stelle wird die Zählervariable um den entsprechenden Wert erhöht und auf den Stack platziert. Anschließend wird eine Konstante mit dem festgelegten Vergleichswert (in diesem Beispiel der Wert „3“) auf dem Stack angelegt. Nun werden diese beiden Werte auf dem Stack verglichen, um zu entscheiden, ob die Abbruchbedingung erfüllt ist. Falls dies nicht der Fall ist wird zu Index 5 gesprungen.

Dieser Bereich, definiert durch die Index Nummern 5 bis 9, stellt den inneren Teil der Schleife dar welcher bei dem Durchlauf ausgeführt wird.

```
public void schleife() {
    for (int i = 0; i < 3; i++) {
        System.out.println(i);
    }
}
```

Abbildung 3.4: Quelltext der Schleife

Grundsätzlich kann die Identifizierung von einer Schleife in Java Bytecode wie folgt durchgeführt werden:

1. „goto“ Anweisung finden
2. Existiert in diesem referenzierten Bereich eine Vergleichsoperation?
3. Führt der Vergleich zu einem Sprung in vorhergehende Code Blöcke?

Diese Herangehensweise bietet einen möglichen Ansatz zur Findung von einfachen Schleifen. Zugleich muss jedoch erwähnt werden, dass das Spektrum von Schleifen in Java groß ist und es daher auch Wiederholungen nach einem anderen Schema geben kann [SOI13].

3.2.2 Bedingungen

Ebenso wie Schleifen beeinflussen Bedingungen den linearen Ablauf einer Methode. Je nach Eintreten einer Voraussetzung variiert der Verlauf und es kommt zu Verzweigungen. Die nachfolgende Abbildung zeigt eine einfache Fallunterscheidung in Bytecode Befehlen.

```
public void fallunterscheidung(int);
  Code:
    0:   iload_1
    1:   ifge    15
    4:   getstatic       #15; //Field java/lang/System.out:Ljava/io/PrintStream;
    7:   ldc     #31; //String negativ
    9:   invokevirtual   #33; //Method java/io/PrintStream.println:(Ljava/lang/St
ring;)V
   12:   goto    23
   15:   getstatic       #15; //Field java/lang/System.out:Ljava/io/PrintStream;
   18:   ldc     #36; //String positiv
   20:   invokevirtual   #33; //Method java/io/PrintStream.println:(Ljava/lang/St
ring;)V
   23:   return
```

Abbildung 3.5: Fallunterscheidung in Bytecode

Die eigentliche Fallunterscheidung geschieht in der Zeile mit Index 1. Der Befehl „ifge 15“ prüft, ob die Variable auf dem Stack größer („g“ = engl. „greater“) oder gleich („e“ = engl. „equal“) dem Wert „0“ ist. Falls dies so ist wird zu dem Index 15 gesprungen. Ist dies jedoch nicht der Fall (der vergleichende Wert also kleiner als „0“), wird der Bereich 4 bis 12 ausgeführt. Der Befehl bei Index 12 („goto 23“) stellt hierbei sicher, dass der andere Fall übersprungen wird, da diese Bedingung nicht erfüllt ist.

Ebenfalls zu erkennen ist an diesem Ausschnitt, dass Strings in Java Bytecode unverschlüsselt gespeichert werden. Die entsprechenden Ausgaben („positiv“ und „negativ“) liegen als Klartext vor [SOI13].

Es gibt neben dem „ifge“ Vergleichsoperator noch weitere Möglichkeiten zwei Werte zu vergleichen. Neben unterschiedlichen Verhältnisprüfungen existieren auch Befehle die prüfen, ob eine Variable gleich oder ungleich „null“ ist [O13].

Die Erkennung von Fallunterscheidungen kann nach diesem Schema erfolgen:

1. Bedingungsanweisung finden
2. Gibt es „goto“ Anweisungen (=Fälle)?
3. Zusammengehörige Bereiche selektieren und zusammenfassen

3.2.3 Java Decompiler

Der Java Decompiler ist ein Programm, welches die einzelnen Schritte zur Rekonstruktion einer Java Klasse automatisch durchführt und als Ergebnis einen funktional gleichen Quelltext liefert. Die Anwendung ist in der Programmiersprache C++ geschrieben und unterstützt Java bis zu der Version 7 [JDP13].

Ohne Einarbeitungszeit in die Materie des Reverse Engineering ermöglicht der Java Decompiler somit auch technisch weniger visierten Benutzern eine Umwandlung von .class Dateien in Quellcode. Die folgende Gegenüberstellung zeigt die ursprüngliche Klasse Stoppuhr (Abb. 3.6) und die erzeugte Umwandlung mit dem Java Decompiler (Abb. 3.7).

Das Ergebnis einer Dekompilierung mit dem Java Decompiler ähnelt der Ausgangsklasse an vielen Stellen und unterscheidet sich nicht in der Ausführung. Wie bereits in den vorhergehenden Kurzbeispielen festgehalten, werden die Variablennamen, Textsequenzen und Bezeichnungen komplett wiederhergestellt.

Auch die Schleifensignaturen und Verzweigungen konnten rekonstruiert werden.

Nur die Kommentare (eingeleitet durch „//“) werden nicht angezeigt, da diese beim Kompilierungsvorgang nicht mit übersetzt werden.

```java
public class Stoppuhr {

    //globale Variablen
    private long start, ende;

    public void start()
    {
        this.start = System.currentTimeMillis()
    }

    public void end()
    {
        this.ende = System.currentTimeMillis();
    }

    public long result()
    {
        return ende-start;
    }

    //Test Schleife
    public void schleife() {
        for (int i = 0; i < 3; i++) {
            System.out.println(i);
        }
    }

    //Test Fallunterscheidung
    public  void fallunterscheidung(int i){
        if(i<0)
            System.out.println("negativ");
        else
            System.out.println("positiv");
    }

}
```

Abbildung 3.7: Quelltext original

```java
import java.io.PrintStream;

public class Stoppuhr
{
  private long start;
  private long ende;

  public void start()
  {
    this.start = System.currentTimeMillis();
  }

  public void end()
  {
    this.ende = System.currentTimeMillis();
  }

  public long result()
  {
    return this.ende - this.start;
  }

  public void schleife()
  {
    for (int i = 0; i < 3; i++) {
      System.out.println(i);
    }
  }

  public void fallunterscheidung(int i)
  {
    if (i < 0) {
      System.out.println("negativ");
    } else {
      System.out.println("positiv");
    }
  }
}
```

Abbildung 3.6: Quelltext Ausgabe von Java Decompiler

4 Fazit und Ausblick

4.1 Einschränkungen des Decompilens

Bei den bisher gezeigten Ansätzen des Reverse Engineering von Java Klassen offenbaren sich mögliche Einschränkungen und Sicherheitsrisiken des Decompilens.

Da Variablennamen und Bezeichner von Methoden auf den ersten Blick komplett wiederherstellbar sind, lässt sich eine Rekonstruktion nicht verhindern.

Die genauen Bezeichnungen unterstützen zudem den Reverse Engineerer bei dem besseren Verstehen der Befehlsabfolgen, sodass die ursprüngliche Logik eindeutig wird. Einzig die fehlenden Kommentare, die gegebenenfalls eine weitere Verständnishilfe für den eigentlichen Entwickler ist, beeinträchtigen diesen Vorgang. An genau dieser Stelle, das Verstehen von Code, setzt eine Technologie zum Schutz des Quelltextes an, ein Quelltextverschleierer (engl. Obfuscator) [K04].

Ein Quelltextverschleierer verändert nur die äußere Form des Programms, nicht jedoch die eigentlichen Funktionen. So werden bei der Verwendung einer solchen Anwendung unter anderem den Variablen und Methoden zufällig generierte Namen zugeteilt. Ebenso werden die eigentlichen Werte von Konstanten verschleiert, indem der eigentliche Wert mit einer Funktion umschrieben wird (zum Beispiel der Wert „2“ dargestellt als „$2*(3^2-8)$“).

Das logische Verstehen eines so veränderten Codes wird dadurch, obwohl der eigentliche Bytecode wiederherstellbar ist, schwer bis unmöglich gemacht.

4.2 Fazit und Ausblick über die Verwendung von Reverse Engineering

Zusammenfassend, unter Betrachtung der Möglichkeiten des Reverse Engineering von Java, kann gesagt werden, dass sich vor allem Entwickler mit dieser Thematik beschäftigen sollten, die kostenintensive Projekte bearbeiten. Naiv erstellte Programme mit eindeutigen Bezeichnern ermöglichen ein einfaches Decompilen ohne vorausgesetzte Fachkenntnisse, wie das Programm „Java Decompiler“ eindrucksvoll darlegt. Um den Programmcode besser zu schützen empfiehlt es sich daher auf weitere Programme, wie zum Beispiel einen Quelltextverschleierer, wert zu legen.

In Anbetracht der heutigen immer weiter voranschreitenden Digitalisierung, stellt das Thema des Reverse Engineering vor allem für spezialisierte Software Unternehmen eine ernstzunehmende Gefahr dar. Die entwickelten Algorithmen und Methoden können unter Umständen von Konkurrenzunternehmen kopiert werden, ohne eigene Entwicklungskosten zu generieren.

Außerdem können mögliche Softwarefehler von kriminell gesinnten Personen dazu genutzt werden, Schaden anzurichten und sensible Daten auszuspähen. Dies könnte vor allem dann der Fall sein, wenn der Programmierer Zugangsdaten als Klartext in Strings abgespeichert hat, welche wie bereits festgestellt, nicht verschlüsselt werden.

Alles in allem müssen sich vor allem Entwickler mit dem Thema des Reverse Engineering genau befassen, da sowohl jetzt als auch in der Zukunft weiter Programme in Java geschrieben werden, welche (wie auch andere Programmiersprachen) nicht für jeden offengelegt werden sollten.

Literaturverzeichnis

[JDP13] http://jd.benow.ca/. „Java Decompiler Project“. Stand 16.12.2013.

[K04] Alex Kalinovsky. „Covert Java: Techniques for Decompiling, Patching, and Reverse Engineering“. Sams Verlag 2004.

[KEP10] H. Kreisel, B. Engleder, C. Pernsteiner. „Decompiler - Der Stand der Technik im Jahr 2008“. GRIN Verlag GmbH 2010.

[O13] http://docs.oracle.com/javase/specs/jvms/se7/html/. „The Java® Virtual Machine Spezification“. Stand 16.12.2013.

[SOI13] http://www.dcs.ed.ac.uk/teaching/cs1/CS1/Bh/Notes/JavaByteCode.pdf. „Compilation I: Java Byte Code“. School of Informatics, University of Edingburgh. Stand 16.12.2013.

[WIK13] http://de.wikipedia.org/wiki/Reverse_Engineering. Stand 16.12.2013.

[DUD13] http://www.duden.de/rechtschreibung/Compiler. Stand 16.12.2013.