Reverse Engineering: Java Decompiler

Jan Möller

Proseminar

6. Februar 2014

Inhaltsverzeichnis

Wofür Reverse Engineering?

- Was ist Reverse Engineering?
  - Zerlegung von fertigen Systemen
    - Hardware
    - Software
  - Informationsgewinnung über:
    - Den Konstruktionsvorgang
    - Die verwendeten Materialien (Quellcode)

Wofür Reverse Engineering?

- Mögliche Anwendungsgebiete:
  - Industriespionage
  - Nachträgliche Wartung von Software
  - Antivirus Software

Wofür Reverse Engineering?

- Programme bestehen aus Quellcode
  - Funktionen, Definitionen und Anweisungen
  - Für den Menschen lesbar, jedoch nicht für einen Computer
- Compiler:
  - Quellcode → Maschinensprache
- Decompiler:
  - Maschinensprache → Quellcode

Die Java Architektur

- Java Runtime Environment (JRE)
  - Nachteil: Installation → Mehraufwand für den Nutzer
  - Vorteil: Plattformunabhängigkeit
- Java Virtual Machine (JVM)
  - Adapter, der Java Programme in die eigentliche Maschinensprache übersetzt → Java Bytecode
  - → Ermöglicht **einfaches** Reverse Engineering

Die Java Architektur

- Bytecode Anweisungen

```
<index> <opcode> [ <operand1> [ <operand2> ... ] ] [<comment>]
```

  - Index: „byte offset“ für die Speicheradresse
  - Opcode: Java Bytecode Befehl
  - Operand: Parameter
  - Comment: vom Compiler erzeugte Kommentare

Die Java Architektur

- Beispiel einer Bytecode Befehlsfolge:

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

- 0: Methodenstart
- 1: Statischer Methodenaufruf
- 4: Setzen einer Variable
- 7: Ende der Methode

Bytecode Analyse

- Visualisierung von Bytecode mit „javap.exe“

```
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
```

Bytecode Analyse

- Darstellung enthält:
  - Konstruktoren
  - Methoden
  - Referenzierte Klassen

```
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
```

- Bereits erkennbar:
  - Ursprüngliche Bezeichner sind unverändert
  - Package Zuordnung
  - Rückgabewerte
  - Variablennamen

Bytecode Analyse

- Bedeutungen der Befehle:

```
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
```

- aload_<n>
  → Laden einer referenzierten Variable
- invokespecial
  → Instanz aufrufen (Konstruktor)
- invokestatic
  → Statische Klassenmethode aufrufen
- putfield
  → Wert eines Feldes setzen
- getfield
  → Liefert den Wert eines Feldes
- lsub
  → Subtrahiert zwei Werte des Datentyps „long“

Bytecode Analyse

- Ausgangsklasse

```
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;
    }
}
```

Bytecode Analyse

- Lineare Abläufe sind einfach zu rekonstruieren
  - Befehle schrittweise übersetzen
- Schwieriger:
  - Schleifen
    → es gibt keinen „loop“ Befehl im eigentlichen Sinne
  - Bedingungen
    → Verzweigungen

Bytecode Analyse: Schleifen

- Einfaches Beispiel in Bytecode

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

```
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
```

- 0 & 16: Integer mit dem Wert „0“ bzw. „3“ auf dem Stack platzieren
- 1: Stack Variable abspeichern in Feld 1
- 2: Springt direkt zu Index 15 ohne Index 5 bis 12 zu beachten
- 8 & 15: Liest den aktuellen Wert von Feld 1 und platziert diesen auf den Stack
- 12: Erhöht Variable aus Feld 1 um den Wert „1“
- 17: Vergleicht die obersten Variablen auf dem Stack

Bytecode Analyse: Schleifen

- Identifizierung von Schleifen:

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

Bytecode Analyse: Bedingungen

- Einfaches Beispiel in Bytecode

```
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
```

- Fallunterscheidung bei Index 1: „ifge 15“
  - „ge“ → greater equal ➔ größer gleich
  - Prüft, ob die Zahl auf dem Stack größer oder gleich „0“ ist
  - Ist dies der Fall: Sprung zu Index 15
  - „Else-Bereich“: Index 4 bis 12
- Es fällt auf: Strings liegen im **Klartext** vor!

Bytecode Analyse: Bedingungen

- Erkennung von Fallunterscheidungen

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

Java Decompiler

- Programm zur automatischen Ausführung Rekonstruktion
  ➔ Für jeden Anwender geeignet
- Liefert funktional gleiche Quelltexte
- Java Decompiler ist in C++ geschrieben worden
- Unterstützt Java bis zu der Version 7

Live Demo

Schutzmöglichkeiten

- Obfuscator
  - Verschleiert den Quelltext → Einschränkung der Lesbarkeit
  - Bezeichner werden zufällig generiert
  - Keine Änderung der Funktionalität

- Wichtige Daten nicht als Klartext verwenden
  ➔ Strings verschlüsseln

Fazit

- Reverse Engineering spielt eine große Rolle bei der voranschreitenden Digitalisierung
  → sowohl positive, als auch negative Rolle

- Entwickler sollten sich mit dieser Thematik befassen
  → Gegenmaßnahmen wählen
  → Sensible Daten schützen