# Gliederung

1. Einleitung
2. Ziel
3. Aufbau und Komponenten
   1. Befehl ausführen
   2. Verhalten aufzeichnen
   3. Verhalten analysieren
   4. Auswertung speichern
4. Funktionen
   1. Daten sammeln
   2. Ausführbare Datei emulieren
5. Fazit & Ausblick
6. Anhang
7. Quellenverzeichnis

# Einleitung

Der Prozess des Software Reverse Engineerings beschäftigt sich mit der Nachkonstruktion vorhandener Produkte. Zum Erreichen des Ziels müssen verschiedene Eigenschaften von Software-Produkten Verstanden und extrahiert werden. Zu diesen Eigenschaften gehören beispielsweise der Aufbau, der Programmfluss, die Struktur von Objekten und die Funktion von Codesegmenten.

Hierfür können verschiedene Techniken angewandt werden. Die gängigste Vorgehensweise besteht darin, einen Disassemblierer (IDA, Hopper, objdump) zu verwenden und die Funktion von Codesegmenten anhand der erkannten Maschinenbefehle nachzuvollziehen. Dieser Vorgang ist sehr zeitintensiv und erfordert viel Erfahrung, vor allem, wenn aus der untersuchende Software hilfreiche Symboltabellen entfernt wurde, welche die in der Programmiersprache verwendeten Variablen- und Funktionsnamen enthalten können.

Um das Verstehen von Objekt-Dateien zu erleichtern, wird aktuell versucht aus dem schwer lesbaren Maschinencode für den Menschen leichter lesbaren (Pseudo-)Code zu generieren. Dieser Vorgang des Decompilierens befindet sich jedoch aufgrund vieler Schwierigkeiten bei der Umsetzung noch in einer frühen Phase. Das Problem besteht darin, dass zwischen mehreren Maschinenbefehlen, die auch an unterschiedlicher Stelle stehen können, ein Zusammenhang erkannt werden muss, der dem Benutzer leicht verstehbar ausgegeben werden muss. Ebenso können Missverständnisse entstehen, wenn Maschinenbefehle direkt in lesbaren Code umgewandelt werden oder erst zusammengefasst werden. Durch kleine Fehler in der Übersetzung kann die Darstellung des Codes verwirrender als der Maschinencode sein oder es können wichtige Schritte des Programms verloren gehen. Bei aktuellen Decompilern fällt auf, dass der Maschinencode lediglich Zeile für Zeile übersetzt wird. So werden beispielsweise Jump-Anweisungen nicht in eine besser verstehbare If-Anweisung umgewandelt, sondern direkt mit der im Binärcode entsprechenden Sprungadresse ausgegeben. Dadurch entsteht kein Mehrwert, weshalb diese simple Übersetzung als überflüssig angesehen werden kann.

Ein weiterer Bereich, der zum Software Reverse Engineering hinzugezählt werden kann, ist die Emulation. Hierbei wird eine Maschine durch Software nachgebildet und Maschinencode nur virtuell ausgeführt. Meist wird diese Technik benutzt um Maschinencode auf einem nicht mit diesem Code kompatiblen System auszuführen. Dies wird erreicht, indem der Maschinencode geparst wird und anschließend durch Datenbanken, welche die Bedeutung des Maschinencodes enthalten, die entsprechenden Änderungen am nachgebildeten System vorgenommen werden. So kann Software, bzw. Maschinencode, ohne direktem Zugriff zur Hardware ausgeführt werden. Die Vorteile zeichnen sich zum einen durch die Möglichkeit aus, nicht kompatiblen Bytecode auf einer Maschine laufen zu lassen und zu nutzen. Zum anderen bietet die Emulation die Möglichkeit Software schrittweise auszuführen und die Veränderungen zu beobachten und nachzuvollziehen. Auch Schadcode kann ohne Bedenken untersucht werden, weil die Maschine lediglich durch einen Prozess simuliert wird, welche durch beenden dieses Prozesses aus dem Speicher verschwindet.

Das schwierigste und zeitintensivste bei der Programmierung eines solchen Emulator besteht aus der Erstellung der benötigten Datenbanken. Es muss für jeden auf der Maschine verfügbaren Befehl ein Datensatz erstellt werden, der das Verhalten der Maschine beschreibt. Hierfür müssen große Dokumente, die sogenannten Befehlssatzarchitekturen, durchgearbeitet und für den Emulator abgespeichert werden. Genau dieses Problem ist der Gegenstand dieser Arbeit und soll durch automatische Generierung der benötigten Daten beseitigt werden.

# Ziel

Im Vordergrund dieser Arbeit steht die Untersuchung der Möglichkeit Informationen über Maschinenbefehle für einen Emulator automatisch zu generieren und für eine Emulation einer beliebigen ausführbaren Binärdatei zu nutzen. Hierfür müssen vorab die benötigten Komponenten mit ihren Funktionen definiert werden. Anschließend muss festgelegt werden, wie und mit welchen Hilfsmitteln diese Funktionen implementiert werden.

Zuerst stellt sich die Frage, wie diese Daten ohne weitere Kenntnisse ermittelt werden können. Dies kann erreicht werden, indem auf der zu untersuchenden Maschine alle möglichen Befehle ausgeführt werden und die Zustände des Systems vor und nach dem jeweiligen Befehl aufgezeichnet und untersucht werden. Nach der Analyse muss die Auswertung für den Emulator aufbereitet und abgespeichert werden. Nach erfolgreicher Abarbeitung aller Befehle soll es möglich sein, eine beliebige ausführbare Binärdatei zu emulieren und jeden einzelnen Schritt nachvollziehen zu können. Im Optimalfall soll der Benutzer über eine Schnittstelle sowohl mit dem Emulator, als auch mit der der emulierten Datei kommunizieren können.

Bei der Verwirklichung des Ziels treten verschiedene Probleme auf, die möglicherweise zeitaufwendig lösbar sind. Auch für die Geschwindigkeit des Sammelns der Daten, bzw. der Emulation einer Datei, muss auf eine effiziente Implementierung geachtet werden, welche meist viel Zeit benötigt. Aufgrund der begrenzten Zeit wurde in dieser Arbeit die Priorität auf die Kernfunktionen des Programms gelegt. Genauere Beschreibungen zu Problemen und Vernachlässigungen bei der Programmierung finden sich in den folgenden Kapiteln an den jeweiligen Stellen. Ebenso wurde bei der Programmierung nur auf einer 32 Bit x86-Architektur gearbeitet. Das Vorgehen sollte jedoch auf andere Maschinen übertragbar sein.

# Aufbau und Komponenten

## Befehl ausführen

Es gibt mehrere Möglichkeiten einen beliebigen Maschinenbefehl mithilfe des Bytecodes auszuführen. Um das Auftreten unerkannter Abhängigkeiten zu vermeiden, ist es in jedem Fall erforderlich, einen neuen, unberührten Prozess zu erzeugen und zu untersuchen. Wird der Befehl in ein und demselben Prozess ausgeführt und untersucht, so kann sich der Befehl aufgrund zuvor ausgeführter Befehle anders verhalten, als in einer neu initialisierten Prozessumgebung. Beispielsweise wenn ein Register durch eine zuvor ausgeführte nicht deterministische Funktion geändert wurde und von dem untersuchten Befehl verwendet wird, entsteht eine Abhängigkeit, die fast unmöglich zu erkennen ist.

### Bytecode durch Cast eines Funktionszeigers ausführen

Eine mögliche Lösung ist den Bytecode in einen ausführbaren Speicherbereich zu laden und anschließend auszuführen[[1]](#footnote-1).

/\* copy code to executable buffer \*/

void**\*** buf **=** mmap**(**0**, sizeof(**code**),** PROT\_READ**|**PROT\_WRITE**|**PROT\_EXEC**,**

MAP\_PRIVATE**|**MAP\_ANONYMOUS**, -**1**,** 0**);**

memcpy**(**buf**,** code**,** **sizeof(**code**));**

/\* run code \*/

**((**void **(\*)** **(**void**))**buf**)();**

**Listing 1: Beliebigen Bytecode ausführen**

Hier wird die Variable *code* vom Type *char[]* in einen ausführbaren Bereich kopiert und anschließend ausgeführt. Mit der Funktion *mmap*[[2]](#footnote-2) können Dateien oder Geräte in den dynamischen Speicher geladen werden. Mit den in Listing 1 verwendeten Argumenten gibt die Funktionen einen Zeiger auf einen neu allokierten Speicherbereich mit der Länge des Strings zurück, welcher gelesen, beschrieben und ausgeführt werden kann. Die Adresse wird hierbei vom System bestimmt. In Zeile 4 wird mithilfe der Funktion *memcpy*[[3]](#footnote-3) der Bytecode in den Puffer kopiert und schließlich in Zeile 6 ausgeführt. Dies erfolgt durch einen Cast der Variablen *buf* auf einen Funktionszeiger ohne Rückgabewert und ohne Parameter und anschließendem Aufruf der Funktion durch zwei runde Klammern.

Diese Vorgehensweise bringt jedoch mehrere Probleme mit sich. Auch wenn der Code in einem neuen Prozess ausgeführt werden würde, sind mehrere Funktionen zur Vorbereitung nötig, die das Ergebnis verfälschen können. Ein weiteres Problem ist der Funktionsaufruf eines dynamisch allokierten Speicherbereichs. Zum einen erfolgen bei einem Funktionsaufruf weitere Änderungen des Systems, die mit aufgezeichnet werden würde. Zum anderen ist die Adresse erst zur Laufzeit bekannt, wodurch Schwierigkeiten bei der Aufzeichnung entstehen können, da nicht klar ist an welchen Stellen der Prozess angehalten werden muss. Ebenfalls wird der Prozess in einen „Segmentation Fault“ hineinlaufen, falls kein Return-Befehl im Bytecode enthalten ist, welcher auch Änderungen nach sich zieht, die nicht zum eigentlich untersuchten Befehl gehören.

### Bytecode durch Modifikation einer Vorlage ausführen

Eine andere und deutlich einfachere Möglichkeit besteht daraus, einen Prozess mithilfe einer ausführbaren Datei zu starten, in der der zu untersuchende Befehl einfach in der Main-Funktion hartkodiert vorzufinden ist. Die Adresse des Befehls ist bekannt, wodurch das Anhalten des Prozesses an den gewünschten Stellen vereinfacht wird. Folgendermaßen kann dabei vorgegangen werden: Es wird eine Programm-Schablone erstellt, indem eine C-Datei mit ausreichend „No Operation“-Befehlen erstellt und kompiliert wird.

int main**(**int argc**,** const char**\*** argv**[])** **{**

asm volatile**(**"nop"**);**

**}**

**Listing 2: Vorlage „template.c“**

Durch Inline-Assembly wird in der Main-Funktion Platz geschaffen. Die Main-Funktion enthält (abgesehen von Speicherverwaltungsfunktionen) vorerst nur „No Operation“-Befehle, welche eine Länge von einem Byte haben. Um alle Befehle untersuchen zu können muss die Zeile 2 insgesamt 15-mal vorkommen, da dies der maximalen Länge eines x86-Befehls entspricht.

080483eb **<**main**>:**

80483eb**:** 55 **push** %ebp

80483ec**:** 89 e5 **mov** %esp**,%ebp**

80483ee**:** 90 **nop**

...

80483fc**:** 90 **nop**

80483fd**:** b8 00 00 00 00 **mov** $0x0**,%eax**

8048402**:** 5d **pop** %ebp

8048403**:** c3 **ret**

**Listing 3: Disassembly der kompilierten Vorlage**

Nach der Kompilierung erhält man das Ergebnis in Listing 3. Hier wird die Start- und Endadresse des Blocks mit den „No Operation“-Befehlen benötigt, um den gewünschten Befehl in die Datei einzufügen und den Prozess an den entsprechenden Stellen zu stoppen. Hier kann man den Zeilen 4 und 7 die Adressen entnehmen. Die Startadresse lautet 0x80483ee und die Endadresse 0x80483fd. Die Differenz der beiden Adresse ergibt die erwarteten 15 Bytes.

Um einen beliebigen Befehl jetzt in einer blanken Main-Funktion auszuführen, wird eine Funktion benötigt, welche die Datei entsprechend anpasst.

bool cpu\_x86\_32**::**build\_exe**()** **{**

ifstream templ**;**

ofstream exe**;**

templ**.**open**(**"./template"**,** ios**::**in **|** ios**::**ate **|** ios**::**binary**);**

exe**.**open**(**"./execute"**,** ios**::**out **|** ios**::**trunc **|** ios**::**binary**);**

**if(!**templ**.**is\_open**())** **{**

cerr **<<** "ERROR: Failed to open ./template!" **<<** endl**;**

**return** **false;**

**}**

**if(!**exe**.**is\_open**())** **{**

cerr **<<** "ERROR: Failed to open ./execute!" **<<** endl**;**

**return** **false;**

**}**

int size **=** templ**.**tellg**();**

templ**.**seekg**(**0**);**

char lead**[**INSTR\_POS**];**

templ**.**read**(**lead**,** **sizeof(**lead**));**

exe**.**write**(**lead**,** **sizeof(**lead**));**

templ**.**seekg**((**int**)**templ**.**tellg**()** **+** len**);**

exe**.**write**((**char**\*)**instr**,** len**);**

char trail**[**size **-** templ**.**tellg**()];**

templ**.**read**(**trail**,** **sizeof(**trail**));**

exe**.**write**(**trail**,** **sizeof(**trail**));**

templ**.**close**();**

exe**.**close**();**

**if(**chmod**(**"./execute"**,** S\_IRWXU

**|** S\_IRGRP

**|** S\_IXGRP

**|** S\_IROTH

**|** S\_IXOTH**)** **==** **-**1**)** **{** // Change file permissons to 755

cerr **<<** "ERROR: Failed to change file permissions!" **<<** endl**;**

**return** **false;**

**}**

**return** **true;**

**}**

**Listing 4: Funktion zur Anpassung der Vorlage[[4]](#footnote-4)**

In den Zeilen 2 bis 13 werden die benötigten Dateien vorbereitet, indem zwei Dateiströme erstellt werden[[5]](#footnote-5). Zum einen muss die zuvor erstellte Vorlage gelesen werden, zum anderen wird eine neue Datei erstellt, bzw. überschrieben, welche den gewünschten Maschinenbefehl enthalten soll. Beide Dateien werden durch das Flag *ios::binary* byteweise bearbeitet und durch das Flag *ios::ate* wird der Lesezeiger der Vorlage an das Ende der Datei gesetzt. Dadurch kann in Zeile 15 die Größe der Datei bestimmt werden. In Zeile 16 wird der Lesezeiger wieder an den Anfang des Stroms gesetzt. Anschließend wird über den Puffer *lead* alles vor dem zu bearbeitenden Bereich in die neue Datei geschrieben. Das Define *INSTR\_POS* entspricht dem Offset der ersten „No Operation“-Anweisung der Main-Funktion, welches äquivalent zu den ersten 12 Bits der Startadresse aus Listing 3 ist. Während in Zeile 23 der gewünschte Befehl *instr* in die neue Datei geschrieben wird, wird in Zeile 22 der Lesezeiger der Vorlage um die Länge *len* des Befehls weiterverschoben. In den Zeilen 25 bis 27 wird äquivalent zu den Zeilen 18 bis 20 der Rest der Vorlage in die neue ausführbare Datei übertragen. Nach dem Schließen der Dateien werden in Zeile 32 noch die Zugriffsrechte der neuen Datei *execute* geändert, damit diese auch im weiteren Verlauf des Prozesses ausgeführt werden kann. Nach Abarbeitung der Funktion *build\_exe* kann nun ein neuer Prozess furch *fork* und *exec* kreiert werden.

Diese Lösung kommt dem Ziel nahe, jedoch ist bei einigen Befehlen eine Vorbereitung des Systems durch andere Befehle erforderlich, weil sie sonst einen „Segmentation Fault“ auslösen. Dadurch können benötigte Befehle nicht untersucht werden, weshalb die Datenbank für die spätere Emulation Lücken aufweisen würde. Um dieses Problem zu lösen, müssten Kombinationen zwischen zwei und mehreren Befehlen erstellt werden und das Verhalten richtig erkannt werden. Ein stumpfes kombinieren beliebiger Befehle oder Bytes kann jedoch zu so vielen Möglichkeiten führen, dass der Aufbau der Datenbank nicht in endlicher Zeit möglich ist. Durch Unterscheidung des zurückgegebenen Signalcodes des Prozesses lässt sich erkennen, ob der Prozess aufgrund einer Schutzverletzung oder einer illegalen Instruktion abgestürzt ist. So können zumindest nicht benötigte Bytecodefolgen aussortiert werden.

Ein Beispiel dieses Problems ist der Befehl *pushl -0x4(%ecx)* mit dem Bytecode 0xff71fc. Der Befehl legt den 32-Bit Wert an der Adresse in *ecx* mit dem Offset -0x4 auf dem Stack ab. Schnell wird klar, dass ein „Segmentation Fault“ bei einer ungültigen Adresse im Register *ecx* auftritt. Bei der Untersuchung des Bytecode 0xff zeigt der Disassembler an, dass es sich um einen nicht existenten Befehl handelt. Der Prozess terminiert aufgrund einer „Illegal Instruction“.

Dieses Verhalten kann auch in dem üblichen Fall eintreten, falls ein zu untersuchender Befehl kürzer als 15 Bytes ist. Hier kann dennoch durch Überprüfung des Program Counters festgestellt werden, ob trotzdem etwas ausgeführt wurde. Wie dies funktioniert und auf was zu achten ist wird im nächsten Kapitel deutlich.

## Verhalten aufzeichnen

Wurde nun ein Maschinenbefehl ausgeführt, so wurden an der Maschine Änderungen vorgenommen. Dabei handelt es sich in erster Linie um Änderungen an den Registern und dem Arbeitsspeicher. Um die Werte organisiert zu speichern und zu verwalten, bietet es sich an, eine Klasse zum Erzeugen von Objekten, welche das System und die jeweiligen Zustände abbilden, zu erstellen. Zum Speichern der Werte ist es vorteilhaft, vorgefertigte Datenstrukturen wie Listen oder Hash Maps zu verwenden. Die Datei *cpu\_x86\_32.h* beinhaltet eine mögliche Repräsentation einer CPU. Um die Struktur übersichtlicher zu gestalten, wurde die Verwaltung der Register in die Datei *register\_x86.h* ausgelagert.

Um alle Zustände der Maschine zu speichern, weisen die Variable für die Register *regs* und die Variable für den Arbeitsspeicher *ram* eine Listenstruktur auf. Ein Zustand des Arbeitsspeichers wird mit einer Mapstruktur abgebildet. Bei dem Schlüssel handelt es sich um einen vorzeichenlosen Integerwert mit 32 Bits, um den kompletten Speicherbereich abzubilden. Der zugehörige Wert wird ebenfalls durch einen vorzeichenlosen Integerwert dargestellt, dieser benötigt jedoch lediglich 8 Bits, bzw. 1 Byte, um ein Zeichen im Arbeitsspeicher darzustellen. Ein Zustand der Register wird durch ein Objekt der Klasse register\_x86 abgebildet. Diese Klasse enthält neben Verwaltungsfunktionen eine Klassenvariable regs, welche ebenso eine Mapstruktur aufweist. Hier wird ein String, welcher den Namen des Registers enthält, auf einen vorzeichenlosen 32-Bit Wert abgebildet, weil die Größe der Register auf einer 32-Bit Maschine 4 Bytes entspricht und die Variablen in der verwendeten Struktur zum herauslesen der Registerwerte ebenfalls diese Größe aufweisen.

Um einen Prozess zu untersuchen existieren erneut mehrere Möglichkeiten. Der im Programm verwendete Vorgang erfolgt mithilfe des Systemaufrufs „ptrace“, welches auch in Debuggern wie „gdb“ Verwendung findet[[6]](#footnote-6). Eine andere Möglichkeit besteht darin, direkt einen Debugger zu verwenden. Zum Beispiel bietet der „gdb“-Debugger die Option mit einer Maschinenschnittstelle gestartet zu werden. Im Vergleich bietet der Debugger deutlich mehr Funktionalität bei der Untersuchung eines Prozesses, auch weil Systeme über eine externe Debugschnittstelle von anderen Maschinen aus erreicht und untersucht werden können.

Eine weitere, jedoch komplizierte und aufwendige, Möglichkeit die Register und Arbeitsspeicherwerte herauszulesen, kann durch Inlineassembler in der Vorlage der zu untersuchenden, ausführbaren Datei realisiert werden. Die Register werden durch Assembleranweisungen, wie beispielsweise *asm volatile("vmovdqu %%ymm0, %0" : "=g" (out))[[7]](#footnote-7),* auf Variablen abgebildet, welche anschließen abgespeichert werden können. Problematisch wird es jedoch, wenn bei Assembleranweisungen Werte in Registern oder im Arbeitsspeicher ändern. Da bessere Alternativen existieren, die Realisierung dieses Vorgangs äußerst zeitaufwendig ist und der Ablauf möglicherweise unvorhersehbar ist, wird auf weitere Untersuchung der Möglichkeit verzichtet.

### Ptrace

### GDB/MI

## Verhalten analysieren

## Auswertung speichern

Die nächste Frage besteht darin, wie die Datensätze dargestellt und gespeichert werden. Da die Art des Speicherns, bzw. das Speichermedium, (SQL, Textdatei) und das Format (Trennzeichen, JSON) nur schnittstellenentscheidend und deshalb nicht relevant für die Kernfunktion des Programms ist, wird hier nicht weiter darauf eingegangen. In der programmatischen Ausarbeitung wurde die Speicherung in üblichen Textdateien mit bestimmter Syntax mit Trennzeichen gewählt. Das wichtige ist eher die interne Darstellung der extrahierten Informationen. Mehr dazu im Kapitel „Darstellung

1. http://stackoverflow.com/questions/9960721/how-to-get-c-code-to-execute-hex-bytecode [↑](#footnote-ref-1)
2. Deklariert in *sys/mman.h* [↑](#footnote-ref-2)
3. Deklariert in *string.h* [↑](#footnote-ref-3)
4. *cpu\_x86\_32.cpp* [↑](#footnote-ref-4)
5. Deklariert in *fstream* [↑](#footnote-ref-5)
6. <https://en.wikipedia.org/wiki/Ptrace>, 05.06.2016 [↑](#footnote-ref-6)
7. Laden eines „unaligned double-quadword floating-point“ Wertes aus dem YMM0-Register in die Variable *out* [↑](#footnote-ref-7)