# 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 Rekonstruktion 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 zu untersuchenden Software hilfreiche Symboltabellen entfernt wurden, 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, Zusammenhänge erkannt und leicht verständlich für den Benutzer ausgegeben werden müssen. Ebenso können Missverständnisse entstehen, wenn Maschinenbefehle direkt in lesbaren Code umgewandelt oder falsch zusammengefasst werden. Durch kleine Fehler in der Übersetzung kann die Darstellung des Codes verwirrender als der Maschinencode selbst 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 direkten Zugriff auf die 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 Emulators 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 die Implementierung der Funktionen erfolgt.

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 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 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. Aufgrund der begrenzten Zeit wurde in dieser Arbeit die Priorität auf die Kernfunktionen des Programms gelegt. Genauere Beschreibungen der Probleme 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. Erwähnenswert ist außerdem, dass bestimmte Befehle nur ausgeführt und bestimmte Register (wie die Debug-Register) nur geändert werden können, wenn das Programm sich im Kernel Level befindet. Da eine Realisierung im Kernel Level jedoch sehr zeitintensiv ist, wurde in dieser Arbeit darauf verzichtet.

### 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 hat den Wert 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, zum anderen eine neue Datei erstellt werden, 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* durch die Funktion *chmod*[[6]](#footnote-6) 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 oder 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. Das führt dazu, dass der Prozess aufgrund einer „Illegal Instruction“ terminiert.

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[[7]](#footnote-7). 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 mit *ptrace* bietet der Debugger deutlich mehr Funktionalität bei der Untersuchung eines Prozesses, unter anderem 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 Inlineassembleranweisungen in der Vorlage der zu untersuchenden, ausführbaren Datei realisiert werden. Die Register werden durch Assembleranweisungen, wie beispielsweise *asm volatile("vmovdqu %%ymm0, %0" : "=g" (out))[[8]](#footnote-8),* auf Variablen abgebildet, welche anschließen abgespeichert werden können. Problematisch wird es jedoch, wenn sich 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[[9]](#footnote-9)

Ptrace erlaubt einem Prozess, dem Tracer, einen anderen Prozess, den Tracee, zu steuern, zu beobachten und zu ändern. Die Initialisierung der Verbindung zwischen den beiden Prozessen kann sowohl durch den Tracer, als auch den Tracee erfolgen. Dafür ist ein Aufruf der Funktion *ptrace* [[10]](#footnote-10) mit dem entsprechenden Parameter erforderlich. Die Signatur der Funktion sieht wie folgt aus:

**long ptrace(enum \_\_ptrace\_request** *request***, pid\_t** *pid***, void \****addr***, void \****data***);**

Der erste Parameter *request* erwartet ein enum, durch welches die auszuführende Aktion bestimmt wird. Durch den Parameter *pid* wird der Prozess gewählt, auf den die Aktion ausgeführt werden soll. Die Aufgabe der anderen Parameter, sowie des Rückgabewerts, richtet sich nach dem übergebenen enum.

Da das Programm zum Sammeln von Daten gezielt einen neuen Kindprozess mit dem *fork* Systemaufruf startet, kann durch einen Funktionsaufruf von *ptrace* mit dem Requestwert *PTRACE\_TRACEME* im Kindprozess die Beobachtung eingeleitet werden. Bei einem anschließenden Aufruf einer Funktion der *exec*-Familie stoppt der Tracee automatisch und wartet auf ein Signal des Tracers um fortzufahren. Eine mögliche Implementierung des Ablaufs ist in Listing 5 zu sehen.

int wstatus**;**

pid **=** fork**();**

**if(**pid **==** 0**)** **{**

ptrace**(**PTRACE\_TRACEME**,** 0**,** **nullptr,** **nullptr);**

**if** **(**execlp**(**"./execute"**,** "./execute"**)** **<** 0**)** **{**

cerr **<<** "ERROR: Execution of file \"execute\" failed!" **<<** endl**;**

**return** **false;**

**}**

**}**

waitpid**(**pid**,** **&**wstatus**,** 0**);**

**if(**wstatus **>>** 8 **!=** SIGTRAP**)** **{**

cerr **<<** "ERROR: Failed to catch exec!" **<<** endl**;**

**return** **false;**

**}**

ptrace**(**PTRACE\_SETOPTIONS**,** pid**,** **nullptr,** PTRACE\_O\_EXITKILL**);**

**Listing 5: Initialisierung der Ptrace-Verbindung**

In Zeile 2 spaltet sich ein Kindprozess vom aktuellen Prozess ab. Der Kindprozess initialisiert sich im darauffolgenden *if*-Block selbst für die *ptrace*-Verbindung. Die Parameter der *ptrace*-Funktion, ausgenommen des Requestparameters, werden ignoriert, weshalb sie den Wert null erhalten. In Zeile 5 wird die aus der Vorlage erstellte, ausführbare Datei in den Prozess geladen. Vor dem ausführen der Datei stoppt der Prozess und wartet auf den Elternprozess. Dieser erkennt in den Zeilen 10-14 durch die Funktion *waitpid*[[11]](#footnote-11) und durch Überprüfen des Statusflags *wstatus*, dass der Tracee pausiert und bereit ist. In Zeile 15 wird noch zur Absicherung die Option *EXITKILL* gesetzt, sodass beim Beenden des Tracers auch der Tracee beendet wird.

Nach der Initialisierung besteht die nächste Aufgabe darin, die zu untersuchende Stelle in der Main-Funktion zu finden. Dies ist durch die aus der Vorlage bestimmten Speicheradressen möglich. Die Speicheradressen sind in dem Fall durch die „defines“ *BRACE\_OPEN* und *BRACE\_CLOSE* hartkodiert. Weil bei beiden Adressen die gleichen Funktionen ausgeführt werden müssen, bietet es sich an die beiden Fälle zusammenzufassen.

**for(**int i **=** 0**;** i **<** 2**;** i**++)** **{**

uint32\_t brace **=** 0**;**

**switch(**i**)** **{**

**case** 0**:** brace **=** BRACE\_OPEN**;** **break;**

**case** 1**:** brace **=** BRACE\_CLOSE**;** **break;**

**}**

**Listing 6: Zusammenfassung der Funktionen durch Schleife mit zwei Fällen**

In Listing 6 ist zu sehen, dass die Schleife zwei Durchgänge erzeugt. Der erste Durchgang wird mit der Startadresse und der zweite mit der Endadresse durchlaufen. Die folgende *while*-Schleife in Listing 7 lässt den Tracee bis zu der gewünschten Adresse mit der Ausführung fortfahren. Zusätzlich müssen mögliche Fehlerquellen durch *if*-Anweisungen abgefangen werden.

uint32\_t eip\_tmp **=** 0**;**

**while(true)** **{**

uint32\_t eip **=** user\_reg**(**PTRACE\_PEEKUSER**,** EIP**,** 0**);**

**if(**eip **==** brace**)** **{**

**break;**

**}** **else** **if(**brace **==** BRACE\_CLOSE **&&** **(**eip **>** BRACE\_CLOSE **||** eip **<** eip\_tmp**))** **{**

**break;**

**}** **else** **if(**eip **==** 0**)** **{**

**return** **false;**

**}** **else** **if(**wstatus **>>** 8 **!=** SIGTRAP**)** **{**

cout **<<** hex **<<** "Terminated with Signal Code 0x" **<<** **(**wstatus **>>** 8**)**

**<<** " in address 0x" **<<** eip **<<** "!" **<<** endl**;**

len **=** eip **-** BRACE\_OPEN**;**

**if(**len **<=** 0**)** **{**

cerr **<<** "ERROR: Length of instruction is 0!" **<<** endl**;**

**return** **false;**

**}**

**if(!**user\_reg**(**PTRACE\_POKEUSER**,**

EFLAGS**,**

user\_reg**(**PTRACE\_PEEKUSER**,** EFLAGS**,** 0**)**

**&** **~(**1 **<<** RESUME\_FLAG**)))** **{**

cerr **<<** "Failed to change EFLAGS register!" **<<** endl**;**

**return** **false;**

**}**

**break;**

**}**

ptrace**(**PTRACE\_SINGLESTEP**,** pid**,** **nullptr,** **nullptr);**

waitpid**(**pid**,** **&**wstatus**,** 0**);**

eip\_tmp **=** eip**;**

**}**

**Listing 7: *While*-Schleife zum Erreichen der Zieladressen**

Durch die geschützte Klassenmethode *user\_reg* in Zeile 3 wird die aktuelle Adresse des Instruction Pointers (im folgenden IP) mithilfe des *ptrace*-Systemaufrufs bestimmt. Falls keine der *if*-Anweisungen greift, also die Adresse nicht erreicht und kein Fehler gefunden wurde, so wird in den Zeilen 27-29 der nächste Maschinenbefehl durch den *ptrace*-Request *PTRACE\_SINGLESTEP* vom Tracee ausgeführt. Zeile 29 sichert den IP zur Fehlererkennung im nächsten Durchlauf der *while*-Schleife. Die erste *if*-Anweisung in Zeile 4 prüft, ob der IP die gewünschte Adresse erreicht hat. In Zeile 6 wird der Fall abgefangen, dass gerade ein Maschinenbefehl untersucht wir, der den IP ändert, wie beispielsweise bei einem *jmp*-Befehl. Zuerst wird geprüft, ob die Startadresse bereits passiert wurde, anschließend ob die aktuelle Adresse im IP einen Wert außerhalb des zu untersuchenden Adressbereichs angenommen hat. In diesen beiden Fällen wird die *while*-Schleife mit einem break verlassen, um die Registerwerte und den Arbeitsspeicher auszulesen. Zeile 8-9 brechen den Vorgang durch ein *return* mit dem bool‘schen Rückgabewert *false* ab, falls die Funktion *user\_reg* den Wert null für den IP zurückgibt. Dies deutet auf einen Fehler hin. Eine entsprechende Fehlermeldung wird in der Funktion *user\_reg* ausgegeben. Der letzte Fall in Zeile 10 der eintreten kann, behandelt eine Terminierung des Prozesses mit einem Signal Code, der nicht einen gewöhnlichen Stopp des Prozesses indiziert. Hier wird in den Zeilen 11 und 12 eine Meldung auf der Konsole ausgegeben. Anschließend wird in Zeile 13 die Länge des ausgeführten Bytecodes ermittelt, indem die Differenz zwischen der Startadresse und dem aktuellen Wert des IP berechnet wird. Ist die Länge null, so wurde kein Befehl erfolgreich ausgeführt. In diesem Fall wird derzeit auf eine weitere Analyse des Bytecodes verzichtet. Es kann sich jedoch um einen gültigen Befehl handeln, falls sich der Signal Code vom „define“ *SIGILL* (illegal instruction) unterscheidet. Dieser Befehl muss zur Untersuchung gegebenenfalls durch andere Befehle vorbereitet werden. Ein Beispiel ist am Ende des vorherigen Teilkapitels „Befehl ausführen“ aufgeführt. Wurde trotzdem festgestellt, dass ein Teil des Bytecodes ausgeführt wurde, so wird in den Zeilen 18-21 das *RESUME\_FLAG* im EFLAGS-Register zurückgesetzt, welches eine fehlerhafte Terminierung eines Prozesses anzeigt. Die Änderung erfolgt durch den *ptrace*-Request *POKE\_USER*. Anschließend wird wie bei den ersten beiden eindeutigen Fällen durch ein *break* mit dem Lesen der Register und des Arbeitsspeichers fortgefahren.

Das Lesen und Speichern der Registerwerte und des Arbeitsspeichers erfolgt durch den Aufruf der beiden geschützten Klassenmethoden *read\_regs* und *read\_ram*. Nach erfolgreichem Ausführen der beiden Methoden wird der Kindprozess mit dem *ptrace*-Request *PTRACE\_KILL* explizit beendet, um Speicherbereich freizugeben.

Die Funktion *read\_regs* erhält die Requests *PTRACE\_GETREGS*, *PTRACE\_GETFPREGS* und *PTRACE\_GETFPXREGS*. Als Rückgabeparameter erwartet die *ptrace*-Funktion eine Variable der Struktur *user\_regs\_struct*, *user\_fpregs\_struct*, bzw. *user\_fpxregs\_struct*[[12]](#footnote-12). Beim ersten Request handelt es sich um die „User“-Register, beim zweiten um die „Floating Point“-Register und beim dritten um „Floating Point“-Register mit Streaming SIMD Extensions (also den XMM-Registern). Weil die dritte Struktur alle Werte der zweiten Struktur enthält, kann auf diese im Allgemeinen verzichtet werden. Nachdem die Strukturen durch den *ptrace*-Funktionsaufruf gefüllt wurden, wird ein neues Objekt vom Typ *register\_x86* erstellt, welches die Werte in einer geeigneten Mapstruktur abspeichert. Anschließend wird das Objekt an die Liste *reg* (Klassenvariable der Klasse cpu\_x86\_32) angehängt. Ein Nachteil von *ptrace* gegenüber „gdb“ ist die unvollständige Struktur *user\_fpxregs\_struct*. Diese umfasst zwar die XMM-Register, es fehlen jedoch die oberen 128 Bits der YMM-Register[[13]](#footnote-13).

Das Auslesen des Arbeitsspeichers gestaltet sich etwas schwieriger, weshalb die Funktion *read\_ram* anhand des Quellcodes im Listing 8 erklärt wird.

bool cpu\_x86\_32**::**read\_ram**()** **{**

ifstream maps\_f**;**

maps\_f**.**open**(**"/proc/" **+** to\_string**(**pid**)** **+** "/maps"**,** ios**::**in**);**

**if(!**maps\_f**.**is\_open**())** **{**

cerr **<<** "ERROR: Failed to open \"maps\"-File \" of process "

**<<** pid **<<** "!" **<<** endl**;**

**return** **false;**

**}**

vector**<**pair**<**uint32\_t**,** uint32\_t**>>** limits\_l**;**

**for(**string line**;** getline**(**maps\_f**,** line**);** **)** **{**

limits\_l**.**push\_back**(**make\_pair**(**stoul**(**line**.**substr**(**0**,** 8**),** **nullptr,** 16**),**

stoul**(**line**.**substr**(**9**,** 8**),** **nullptr,** 16**)));**

**}**

maps\_f**.**close**();**

map**<**uint32\_t**,** uint8\_t**>** map\_m**;**

**for(**pair**<**uint32\_t**,** uint32\_t**>** limits **:** limits\_l**)** **{**

uint32\_t addr **=** limits**.**first**;**

**while(**addr **<** limits**.**second**)** **{**

long val **=** ptrace**(**PTRACE\_PEEKDATA**,** pid**,** addr**,** **nullptr);**

**if(**val **!=** **-**1 **&&** **!**errno**)** **{**

map\_m**.**insert**(**make\_pair**(**addr**,** val**));**

**}** **else** **if(**log**)** **{**

cerr **<<** hex **<<** "Failed to fetch address " **<<** addr **<<** endl**;**

**}**

addr**++;**

**}**

**}**

ram**.**push\_back**(**map\_m**);**

**return** **true;**

**}**

**Listing 8: Funktion *read\_ram* zum Auslesen des Arbeitsspeichers**

Um ein Auslesen des gesamten 4 Gigabyte-Adressraums einer 32-Bit Maschine zu umgehen, müssen die benutzten virtuellen Speicheradressen bekannt sein. Das Mapping der virtuellen Speicheradressen lässt sich aus der Datei „maps“ im *proc*-Dateisystem eines jeden Prozesses herauslesen. Besagte Datei wird in Zeile 3 als Eingangsdateistrom geöffnet. In Zeile 9 wird eine Liste angelegt, welche Paare von zwei Zahlen vom Typ *uint32\_t* enthält. Jedes dieser Paare stellt einen verwendeten Speicherbereich mit einer vorzeichenlosen 32-Bit Start- und Endadresse dar. In den Zeilen 10-13 wird die Datei Zeile für Zeile gelesen. Aus der gelesenen Zeile *line* werden mit der Methode *substr* die geforderten Zahlen extrahiert und durch die Funktion *stoul* in eine vorzeichenlose 32-Bit Zahl umgewandelt. Da die Adresse eine hexadezimale Schreibweise aufweist, muss die Basis der zu konvertierenden Zahl in der Funktion als letzter Parameter angegeben werden. Nachdem das Mapping vollständig ausgelesen wurde, wird der Dateistrom in Zeile 14 geschlossen.

Mit den verwendeten Adressbereichen kann das Auslesen des Arbeitsspeichers erfolgen. In Zeile 16 wird dafür der Behälter, die Map mit der Adresse als Schlüssel und dem zugeordneten Zeichen als dazugehörigen Wert, initialisiert. In der darauffolgenden Zeile wird mithilfe einer *for-each*-Schleife über die Liste mit den Adresse-Zeichen-Paaren, welche die Adressbereiche beinhalten, iteriert. Mit einer *while*-Schleife kann durch einen *ptrace*-Funktionsaufruf mit dem Request *PTRACE\_PEEKDATA* der Wert an jeder Adresse des Adressbereichs gelesen werden. Nach erfolgreichem Lesen einer Adresse wird die Adresse mit dem Wert in die Map eingefügt. Zum Schluss wird die vollständige Map *map\_m* an die Liste *ram* (Klassenvariable von cpu\_x86\_32) angehängt.

Das Auslesen und die Organisation lässt sich effizienter gestalten. Weil die *ptrace*-Funktion laut Signatur ein 32-Bit Wert zurückgibt, beinhaltet die Variable *val* in Zeile 20 auch die Zeichen der nächsten drei Adressen. Diese werden aufgrund der impliziten Konvertierung in Zeile 22 abgeschnitten. Mit diesem Wissen kann in Zeile 26 die Variable *addr* um den Wert 4 erhöht werden. Das Speichern in der Map kann durch bitweise Verschiebung und Konvertierung erfolgen. Eine weitere mögliche Optimierung der Organisation ließe sich erreichen, wenn der Wert der Map ebenfalls ein 32-Bit Datentyp aufweisen würde. Auf diese Änderung wurde jedoch im Rahmen der Bachelorarbeit verzichtet, da dies Änderungen in weiteren Teilen des Programms erfordern würde.

### GDB/MI[[14]](#footnote-14)

Wie am Anfang des Kapitels beschrieben, bietet der „gdb“-Debugger, welcher auch *ptrace* verwendet, deutlich mehr Funktionalität. Um den Debugger zu starten und mit diesem zu kommunizieren, kann ein Pseudoterminal benutzt werden. Durch das Pseudoterminal werden Dateideskriptoren für beide Prozesse, den Prozess für das Sammeln der Daten und den Debugger-Prozess, erstellt, mit welchen eine Kommunikation über *stdout* und *stdin* des Kindprozesses stattfinden kann. Eine mögliche Implementierung eines Pseudoterminals ist „openpty“[[15]](#footnote-15). Mit der Funktion *forkpty[[16]](#footnote-16)* und den richtigen Parametern kann der „gdb“-Prozess gestartet und mit ihm kommuniziert werden. Bei den Parametern von „gdb“ muss über das Flag „-i“ die Schnittstelle „mi“ (für „Machine Interface“) angegeben werden. Die Schnittstelle sorgt für eine für Maschinen bessere Ausgabe und Eingabe. Zum Beispiel wird bei der Abfrage der Registerwerte ein Query im JSON-Format ausgegeben, welches sich leicht parsen und weiterverarbeiten lässt. Im Gegensatz zu *ptrace* werden auch die vollständigen Werte der YMM-Register auslesbar.

Der Ablauf zum Untersuchen des Maschinenbefehls unterscheidet sich nicht von der Variante mit *ptrace*, er gestaltet sich lediglich etwas einfacher. Zuerst muss der Prozess an der Startadresse des Maschinenbefehls angehalten werden. Anschließend müssen die Register und der Arbeitsspeicher ausgelesen werden. Danach wird der Maschinenbefehl so weit wie möglich ausgeführt. Zum Schluss werden erneut die Register und der Arbeitsspeicher ausgelesen, um die Änderungen feststellen zu können. Im Listing 9 ist eine beispielhafte Kommunikation zwischen den beiden Prozessen zu sehen.

(gdb)

**-break-insert \*0x80483ee**

^done,bkpt={number="1",type="breakpoint",disp="keep",enabled="y",addr="0x080483ee",func="main(int, char const\*\*)",file="template.c",fullname="/home/bachelor/Bachelor/emulator/template.c",line="2",thread-groups=["i1"],times="0",original-location="\*0x80483ee"}

(gdb)

**-exec-run**

=thread-group-started,id="i1",pid="21391"

=thread-created,id="1",group-id="i1"

^running

\*running,thread-id="all"

…

(gdb)

**-data-list-register-names**

^done,register-names=["eax","ecx","edx","ebx","esp","ebp","esi","edi","eip","eflags","cs","ss","ds","es","fs","gs","st0","st1","st2","st3","st4","st5","st6","st7","fctrl","fstat","ftag","fiseg","fioff","foseg","fooff","fop","xmm0","xmm1","xmm2","xmm3","xmm4","xmm5","xmm6","xmm7","mxcsr","ymm0h","ymm1h","ymm2h","ymm3h","ymm4h","ymm5h","ymm6h","ymm7h","","","","","","","orig\_eax","al","cl","dl","bl","ah","ch","dh","bh","ax","cx","dx","bx","","bp","si","di","ymm0","ymm1","ymm2","ymm3","ymm4","ymm5","ymm6","ymm7","mm0","mm1","mm2","mm3","mm4","mm5","mm6","mm7"]

(gdb)

**-data-list-register-values x 8**

^done,register-values=[{number="8",value="0x80483ee"}]

(gdb)

**-data-list-register-values x**

^done,register-values=[{number="0",value="0x1"},{number="1",value="0xcf16efe9"}, ... {number="86",value="{uint64 = 0x0, v2\_int32 = {0x0, 0x0}, v4\_int16 = {0x0, 0x0, 0x0, 0x0}, v8\_int8 = {0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}}"},{number="87",value="{uint64 = 0x0, v2\_int32 = {0x0, 0x0}, v4\_int16 = {0x0, 0x0, 0x0, 0x0}, v8\_int8 = {0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}}"}]

(gdb)

**-data-read-memory-bytes 0x08048000 4**

^done,memory=[{begin="0x08048000",offset="0x00000000",end="0x08048004",contents="7f454c46"}]

(gdb)

**-exec-step-instruction**

^running

\*running,thread-id="1"

(gdb)

\*stopped,reason="end-stepping-range",frame={addr="0x080483ef",func="main",args=[{name="argc",value="1"},{name="argv",value="0xffffd164"}],file="template.c",fullname="/home/bachelor/Bachelor/emulator/template.c",line="3"},thread-id="1",stopped-threads="all",core="0"

(gdb)

**-data-list-register-values x 8**

^done,register-values=[{number="8",value="0x80483ef"}]

(gdb)

**-exec-step-instruction**

^running

\*running,thread-id="all"

(gdb)

\*stopped,reason="signal-received",signal-name="SIGSEGV",signal-meaning="Segmentation fault",frame={addr="0x080483ef",func="main",args=[{name="argc",value="1"},{name="argv",value="0xffffd164"}],file="template.c",fullname="/home/bachelor/Bachelor/emulator/template.c",line="3"},thread-id="1",stopped-threads="all",core="0"

(gdb)

…

**Listing 9: Eine Debug-Sitzung mit dem Machine Interface**

Aufgrund vieler Informationen in der Ausgabe ist es schwer die Übersicht zu behalten, weshalb das Protokoll an einigen Stellen gekürzt wurde. Die entsprechenden Stellen sind mit drei aufeinanderfolgenden Punkten markiert. Anders als bei *ptrace* kann sofort, wie in Zeile 2, ein Breakpoint für die gewünschte Adresse angelegt werden. Durch den Befehl *-exec-run* wird der Prozess bis zu der Adresse ausgeführt. Mit dem Befehl in Zeile 15 *-data-list-register-names* können die Namen der Register ausgegeben werden. Die Reihenfolge der Registername in der Ausgabe entspricht der Nummerierung der Register. So wird in Zeile 24 durch *-data-list-register-values x 8* der Wert des an der Stelle 8 befindlichen EIP-Registers ausgegeben, welcher dem Wert der Startadresse entspricht. Wird der gleiche Befehl, wie in Zeile 27, ohne zweiten Parameter verwendet, werden alle verfügbaren Register formatiert ausgegeben. Durch den Befehl *-data-read-memory-bytes* kann der Arbeitsspeicher ausgelesen werde. Der erste Parameter gibt in dem Fall die Adresse an, der zweite Parameter die Länge der zu lesenden Zeichen. Dadurch kann auf einen Schlag der gesamte Adressbereich ausgelesen werden. Der Befehl in Zeile 37 *-exec-step-instruction* entspricht dem *ptrace*-Request *PTRACE\_SINGLESTEP*. Das Auslesen des EIP-Registers in Zeile 46 indiziert, dass ein Byte ausgeführt wurde. Auch in den Zeile 42 und 43 kann sowohl der Grund des Haltens des Prozesses sowie der Wert des aktuellen Instruction Pointers entnommen werden. Nach erneuter Einzelschrittausführung eines Maschinenbefehls in Zeile 49 terminiert der Prozess aufgrund eines „Segmentation Faults“, wie der Zeile 53 zu entnehmen ist. Anschließend können mit den üblichen Mitteln die Register und der Arbeitsspeicher ausgelesen werden.

Genauso wie bei *ptrace* muss hier im Falle eines *jmp*-Befehl das EIP-Register nach Erreichen der Startadresse in jedem Einzelschritt überprüft werden. Neben dem Auslesen aller Register kann auch der Befehl *-data-list-changed-registers* in Betracht gezogen werden, um nur die Registerwerte aufzulisten, die sich verändert haben. Für den Anfang und um Bytecodes, die in einem „Segmentation Fault“ landen, zu untersuchen, ist es jedoch sinnvoll, alle Registerwerte aufzunehmen.

## Verhalten analysieren

Es gibt verschiedene Befehlstypen, die zuverlässig erkannt werden müssen. Jeder Befehl müsste mehrfach mit unterschiedlichen vorangestellten Befehlen untersucht werden, um ihn kategorisieren zu können, weil die Abhängigkeit von beispielsweise Registerwerten nicht bei erstem Mal erkannt werden kann. Die Vorbereitung des Befehls muss deshalb gründlich überdachte werden und möglichst alle Fälle abdecken. Grundlegende Maschinenbefehle können in folgende Kategorien unterteilt werden: arithmetische Operationen, logische Operationen, bit-orientierte Operationen, Speicheroperationen, Vergleichsoperationen, Steueroperationen, Datenkonvertierungsoperationen. Einige Befehle treten auch als eine Kombination mehrerer Kategorien auf[[17]](#footnote-17). Auch die vom jeweiligen Befehl verwendete Adressierungsform muss zuverlässig erkannt werden. Die automatische Kategorisierung erweist sich als äußerst schwer und ist wahrscheinlich am besten mit Vorgehensweisen aus der künstlichen Intelligenz, wie zum Beispiel Entscheidungsbäumen, lösbar.

In dieser Arbeit kommt die tatsächliche Analyse eines Maschinenbefehls aufgrund der Schwierigkeit dieser Aufgabe und der begrenzten Zeit nicht ausführlich vor. Es wird im Weiteren davon ausgegangen, dass es sich bei jedem Befehl um eine arithmetische Operation als Addition oder Subtraktion handelt.

Eine Addition oder Subtraktion zu erkennen ist mit einfachen Mitteln möglich. Ändert sich ein Register oder ein Speicherbereich kann durch die Bildung der Differenz des geänderten Wertes und des vorherigen Wertes der Summand, bzw. der Subtrahend, bestimmt werden. Zusätzlich muss erkannt werden, ob die Operation Register oder den Arbeitsspeicher als Ziel hat und ob sie mit einem hartkodierten Wert oder einem Wert im Speicherbereich des Prozesses erfolgt. Erfolgt eine Addition im Arbeitsspeicher, so muss der Wert des Stack Pointers berücksichtigt werden, da sich dieser bei jedem neu erzeugten Prozess eines Programmes unterscheiden kann. Falls kein hartkodierter Wert benutzt wird, ist die Erkennung der verwendeten Speicheradresse, bzw. des Registers, erforderlich. Wird der Wert dem Arbeitsspeicher entnommen, muss darauf geachtet werden, ob die verwendete Speicheradresse absolut oder relativ zu einem festen oder variablen Wert ist.

Alle diese Informationen müssen für einen effizienten Emulator erkannt, aufbereitet und geeignet abgespeichert werden. Das Erkennen weiterer benötigter Informationen und das Erstellen einer geeigneten Datenbank ist in diesem Kontext ein kontinuierlicher Prozess, da bei komplexen Prozessoren mit vielen verschiedenen Befehle Verhaltensweisen existieren, die nicht immer eindeutig sind. Deshalb wird in dem für diese Arbeit erstellten Programm lediglich der Unterschied der Zustände vor und nach dem untersuchten Maschinenbefehl abgespeichert. Der Versuch, einen kompletten Dump der CPU-nachbildenden Objekte mit allen Informationen und Zuständen zu sichern, führte schnell zu einem vollen Festplattenspeicher, weshalb diese Option verworfen wurde.

## Auswertung speichern und laden

### Auswertung speichern

Die nächste Frage besteht darin, wie die Datensätze dargestellt und gespeichert werden. Da die Art des Speicherns, bzw. des Speichermediums, (SQL, Textdatei) und des Formats (Trennzeichen, JSON) nur schnittstellenentscheidend und deshalb nicht relevant für die Kernfunktion des Programms ist, wird hier nicht weiter darauf eingegangen. In der programmatischen Ausarbeitung werden, wie im vorigen Teilkapitel „Verhalten analysieren“ beschrieben, nur die Unterschiede zwischen den Zuständen gesichert. Hierbei gehen entscheidende Informationen, wie zum Beispiels ein verwendeter Registerwert, verloren. Um die Informationen zu erkennen und zu behalten, müsste vor dem Speichern das CPU-Objekt intensiver untersucht werden. Für das Ziel dieser Bachelorarbeit, unter der Prämisse, dass nur arithmetische Befehle untersucht werden, reicht es, die Änderung der Werte passend zu speichern. Die Speicherung erfolgt dabei in üblichen Textdateien mit bestimmter Syntax.

**$ cat ff75f4**

EIP: 0x080483f0 1-> 0x080483f3

ESP: 0xffae65e8 1-> 0xffae65e4

0xffae65e5: 0x00 1-> 0x70

0xffae65e6: 0x00 1-> 0x71

0xffae65e7: 0x00 1-> 0xf7

**Listing 10: Informationen des Befehls *pushl -0xc(%ebp)* mit dem Bytecode 0xff75f4**

Im Listing 10 ist der Inhalt der Datei „ff75f4“ zu sehen. Im verwendeten Datenmodell erhalten die Dateien den Namen des Befehls, über welchen sie die Informationen enthalten. In der Datei sind im ersten Block alle Register und im zweiten alle Speicheradressen gelistet, die sich nach Ausführung des Prozesses geändert haben. Die Blöcke sind dabei durch eine freie Zeile getrennt. Der Anfang jeder Zeile enthält sonst den Namen des Registers, bzw. den Wert der Speicheradresse, gefolgt von einem Doppelpunkt. Anschließend folgt der Anfangswert des betrachteten Speicherelements. Darauf folgt jeweils ein Pfeil „->“ mit dem neuen Wert. Die Notierung der Werte und Adressen erfolgt in hexadezimaler Schreibweise mit dem Präfix „0x“ und zwischen allen Syntaxelementen befindet sich ein Tabulatorzeichen. Die Zahl vor dem Pfeil gibt an, in welchem Schritt die Änderung erfolgt. Da im vorgestellten Teil des Programms bisher immer nur ein Maschinenbefehl untersucht wurde, beträgt der Wert dieser Angabe immer 1 und ist somit vorerst irrelevant. Sollen jedoch wie beabsichtig mehrere Maschinenbefehle kombiniert untersucht werden, so ist diese Angabe äußerst wichtig. Außerdem wurde diese Darstellung für Kompatibilität zwischen Datensammler und Emulator gewählt, da bei der Emulation der Benutzer dadurch nachvollziehen kann, nach welchem Maschinenbefehl die Änderung stattgefunden hat.

Betrachtet man die gespeicherten Daten des Bytecodes und vergleicht diese mit den Informationen, die für den entsprechenden Assemblerbefehl *pushl -0xc(%ebp)* benötigt werden, so wird das zuvor erwähnte Problem klar. Mit dem Befehl wird ein 32-Bit Wert aus dem Arbeitsspeicher auf den Stack gelegt. Die Speicheradresse des Werts im Arbeitsspeicher wird in diesem Fall mithilfe des EBP-Registers (Base Pointer) bestimmt. Ohne die Information scheint der Befehl lediglich den festen Wert 0xf7717000 auf dem Stack abzulegen. Interpretiert man diesen Befehl arithmetisch, so wird der Instruction Pointer mit der Länge des Befehls 3 summiert und der Stack Pointer um die Größe eines Wertes vom Typ *long* 4 erniedrigt. Bei den geänderten Adressen relativ zum Stack Pointer wird jeweils der zweite Wert aufsummiert.

Die entsprechende Funktion zum Speichern der Änderungen befindet sich in der Datei „cpu\_x86\_32.cpp“ und lautet *dump\_change*. In dieser wird, falls noch keine Datei für den untersuchten Maschinenbefehl existiert, eine neue Datei angelegt und durch die Hilfsklassenmethoden *diff\_ram* der Klasse *cpu\_x86\_32* und *diff\_change* der Klasse *register\_x86* mit den Informationen in der angegebenen Syntax gefüllt.

### Auswertung laden

Für die Emulation ist es erforderlich, die gespeicherten Daten entsprechend wieder laden und verwenden zu können.

# Funktionen

In diesem Kapitel soll die entwickelte Software beispielhaft an einer Testdatei vorgeführt werden. Dabei werden explizit die Informationen über die in der kompilierten Testdatei verwendeten Maschinenbefehle von der Maschine gesammelt. Anschließend wird versucht das Verhalten des Testprogramms durch eine Emulation nachzuvollziehen. Einfachheitshalber werden in der Testdatei hauptsächlich arithmetische Operationen durchgeführt.

Listing test…

Listing Makefile…

## Daten sammeln

## Ausführbare Datei emulieren

# Fazit & Ausblick

Architektur der Datenstruktur (list/map)

Gdb/mi

Vererbung

Effizienz

Künstliche intelligenz

# Anhang

## collect\_data.cpp

## cpu\_x86\_32.cpp

## cpu\_x86\_32.h

## emulate\_file.cpp

## Makefile

## register\_x86.cpp

## register\_x86.h

## template.c

## test/test.c

# Quellenverzeichnis

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. Deklariert in *sys/stat.h* [↑](#footnote-ref-6)
7. <https://en.wikipedia.org/wiki/Ptrace>, 05.06.2016 [↑](#footnote-ref-7)
8. Laden eines „unaligned double-quadword floating-point“ Wertes aus dem YMM0-Register in die Variable *out* [↑](#footnote-ref-8)
9. <http://man7.org/linux/man-pages/man2/ptrace.2.html>, 06.06.2016 [↑](#footnote-ref-9)
10. Deklariert in *sys/ptrace.h* [↑](#footnote-ref-10)
11. Deklariert in *sys/wait.h* [↑](#footnote-ref-11)
12. Definiert in *sys/user.h* [↑](#footnote-ref-12)
13. <https://de.wikipedia.org/wiki/Advanced_Vector_Extensions>, 07.06.2016 [↑](#footnote-ref-13)
14. <https://sourceware.org/gdb/onlinedocs/gdb/GDB_002fMI.html>, 07.06.2016 [↑](#footnote-ref-14)
15. <http://man7.org/linux/man-pages/man3/openpty.3.html>, 07.06.2016 [↑](#footnote-ref-15)
16. Deklariert in *pty.h* [↑](#footnote-ref-16)
17. <https://de.wikipedia.org/wiki/Maschinensprache>, 07.06.2016 [↑](#footnote-ref-17)