# Binäranalyse von Executables

Hagen Fritsch
<fritsch+bincode@in.tum.de>

Hauptseminar: Codeanalyse & Codemanipulation
Betreuerin: Andrea Flexeder

26. Juni 2008

# Inhaltsverzeichnis

# 1 Einführung

Da Software in der Regel nicht in Form des Quelltextes sondern lediglich als ausführbare Datei vorliegt, ist die Analyse von Executables nicht nur für Cracker interessant, sondern wird auf breiter Ebene praktiziert. Einige der Anwendungsbereiche sind folgende:

**Anwendungssicherheit.** In vielen Bereichen (besonders in sicherheitskritischen) möchte man sichergehen, dass eine Anwendung zum einen nur das tut, wofür sie bestimmt ist, also keine Hintertüren oder ähnliche versteckte Features enthält, und zum anderen frei von Fehlern ist.

**Realverhalten.** In genügend Grenzfällen definiert sich Anwendungsverhalten erst durch die Umsetzung des Quelltextes durch den Compiler in Binärcode. Eine tatsächlich genaue Analyse wäre in solchen Fällen auf der Quelltextebene nicht möglich. In Kapitel 1.1 wird dies an einem Beispiel demonstriert.

**Verhaltensanalyse.** Insbesondere in der Antivirus-Industrie ist die Analyse von Viren, Rootkits und anderer Malware unerlässlich, um deren Funktionsweise zu bestimmen und Erkennungs- und Gegenmaßnahmen zu entwickeln.

**Reverse Engineering.** Durch Binäranalyse, lassen sich Anwendungsprotokolle, Datenformate und Algorithmen bestimmen, die vorher nicht öffentlich bekannt waren. Sie lassen sich dadurch z.B. in Open-Source Projekten implementieren, um platformunabhängiges Arbeiten zu ermöglichen.

**GPL-Violations.** Die Analyse von Anwendungen kann zu Tage führen, dass in jener Anwendung GPL-lizenzierter Quelltext ohne Einhaltung der Lizenzbestimmungen verwendet wurde.

**Patchanalyse.** Erstellt eine Softwarefirma einen Sicherheitspatch, so werden oft keine Details über die Sicherheitslücke genannt. Durch eine Differenzanalyse lassen sich allerdings die Änderungen verfolgen und geübte Sicherheitsexperten sind sehr schnell in der Lage, herauszufinden, worin die Sicherheitslücke besteht. Mit diesen Informationen können nun Hersteller von Intrusion Detection Systemen Signaturen erstellen, um der Ausnutzung der Sicherheitslücke in noch nicht gepatchten Produkten zuvorzukommen. Auf dieses Thema wird in Kapitel 4.1 noch detaillierter eingegangen.

**Optimierungen.** Um Optimierungen von Anwendungen, deren Quelltext nicht mehr vorliegt, durchführen zu können, ist die Analyse des Binärcodes unerlässlich.

## 1.1 Unvorhersehbares Compilerverhalten

Quelltext ist nicht immer eindeutig und in vielen Fällen bestimmt erst der Compiler, wie Code umgesetzt wird, und definiert damit erst das tatsächliche Anwendungsverhalten.

Listing 1: Beispiel für unklares Anwendungsverhalten C

```
1  int p = 0;
2  p = p++ + ++p;
```

In Listing 1 ist eine zulässige C-Anweisung dargestellt, deren Verhalten dem ungeübten Leser nicht sofort klar wird. Nimmt man an, die Anweisung `p++` wird zuerst ausgeführt, so ist der erste Operand null auf den `++p` also mit pre-increment addiert wird, was eins wäre. Passiert nun das post-inkrement von `p++` vor der Zuweisung `p = ...` so ist `p = 1`, passiert es danach ist `p = 2`. In einer anderen Ausführungsreihenloge der Operationen kann auch noch `p = 3` oder `p = 4` herauskommen. Erst ein Blick in Listing 2 offenbart, dass `p = 3` sein muss.

Listing 2: Beispiel für klares Anwendungsverhalten in Assembler

```
1  movl    $0, -8(%ebp)      ; p = 0
2  addl    $1, -8(%ebp)      ; ++p          (=1)
3  movl    -8(%ebp), %eax
4  addl    %eax, -8(%ebp)    ; p = p+p      (=2)
5  addl    $1, -8(%ebp)      ; p++          (=3)
```

Ein weiteres schönes Beispiel sind Typumwandlungen, die regelmäßig Menschen in die Verzweiflungen treiben:

Listing 3: Beispiel 2 für unklares Anwendungsverhalten C

```
1  char c = 62*4;                  // 248
2  int x[63];                      // size:= 4*63 = 252
3  printf(c > sizeof(x) ? "yay" : "no");
```

Der naive Leser denkt sich an dieser Stelle, dass 248 ja kleiner als 252 ist und daher „no” ausgegeben wird. Der etwas erfahrenere erkennt schon, dass `c` vom Typ `signed char` ist und damit der größte fassbare Wert 127 ist. In `c` steht also nicht 248, sondern −8. Aber das ist ja immernoch kleiner

als 252. Wer nun wirklich mit C vertraut ist oder (DMS06) gelesen hat, weiß: `sizeof(x)` gibt einen Wert vom Typ `size_t` also einen `unsigned int` zurück. Der Vergleich mit einem `signed char` ergibt dann einen Typcast auf `signed int` (mit *sign extension*) bei dem `c` dann immernoch $-8$ ist und anschließend auf `unsigned int`, so dass `c` plötzlich 0xFFFFFFF8 ist und damit weit größer als 252 und am Ende wirklich „yay“ ausgegeben wird. Ein Blick in den Assemblerquelltext gibt hier auch ohne detailliertes Hintergrundwissen schnell Auskunft über das tatsächliche Anwendungsverhalten wie in Listing 4 dargestellt. In diesem Fall hat der Compiler sich den ganzen Typumwandlungsaufwand sogar gespart und man erkennt, dass jeder negative Wert in `c` zur Ausgabe von „yay“ führt.

Listing 4: Beispiel 2 für klares Anwendungsverhalten in Assembler

```
1         movb    $-8, -5(%ebp)     ; c = (char) 248 = -8
2         cmpb    $0, -5(%ebp)
3         jns     .L2               ; c < 0 ?
4         push    "yay"
5         jmp     .L4
6 .L2:
7         push    "no"
8 .L4:
9         call    printf
```

Während in diesen Beispielen das Verhalten, wenn man denn den entsprechenden C-Standard gut genug kennt, genau definiert ist, lassen sich auch Beispiele finden für die der C-Standard nicht explizit festlegt, in welcher Reihenfolge z.B. Funktionsparameter ausgewertet werden, was auf unterschiedlichen Plattformen und unterschiedlichen Compiler zu verschiedenen Resultaten führt.

# 2 IDA Pro

IDA Pro ist ein Werkzeug, das die händische Analyse von Binärdateien erleichtert. Es kennt alle gängigen Formate ausführbarer Dateien (auch für verschiedene Plattformen) und ist in der Lage grundlegende Informationen wiederherzustellen. IDA Pro ist in erster Linie ein Disassembler (vgl. FKL08).

In der automatischen Analysephase werden Funktionsgrenzen (d.h. Anfang und Ende von Funktionen) bestimmt und Referenzierungen gebildet. Für jede Funktion ist somit bekannt, an welchen Stellen sie aufgerufen wird und welche Funktionen sie aufruft. Des weiteren werden lokale Variablen

(d.h. Stack-Variablen, die z.B. relativ über das `ebp`-Register adressiert werden) analysiert. DLL-Aufrufe werden aufgelöst und Stack-Parameter den Funktionsargumenten zugeordnet.

## 2.1 Demonstration an einem Beispiel

An einem Beispiel (Listing 5) werden nun die wesentlichsten Konzepte erklärt und anschließend die Defizite von IDA Pro aufgezeigt.

Listing 5: Das laufende Beispiel (angelehnt an RBL06)

```
1  #include <stdio.h>
2
3  void array_example(int a, int b) {
4      int array[10], i;
5      for(i=0;i<10;i++)
6          array[i] = i;
7  }
8
9  int main() {
10     int a=4;
11     int b=5;
12     array_example(a, b);
13     return 0;
14 }
```

In IDA Pro wird, wie in Abbildung 1 am Beispiel der Funktion `array_example` dargestellt, die ansonsten lineare Struktur des Assemblercodes aufgebrochen und die durch die `for`-Schleife entstandenen Sprünge werden visualisiert. IDA Pro erkennt richtig, dass die Funktion zwei Variablen besitzt und wandelt die relative Adressierung `[ebp-4]` in das Pseudonym `[ebp+var_4]` um. Es ist dem Benutzer dann möglich diese Variable umzubenennen. Jeder Zugriff innerhalb dieser Funktion auf die Variable wird von IDA Pro mit dem Pseudonym versehen, was die Lesbarkeit erheblich verbessert. Die Deklaration am Anfang der Funktion gibt Auskunft über die Größe der Variablen. `var_4` ist 4 Byte groß und `var_2C` ist 0x2C $-4$ = 0x28 = 40 Byte groß, entspricht also genau der Größe des definierten Arrays.

## 2.2 Defizite von IDA Pro

Allerdings ist IDA Pro an dieser Stelle lediglich bekannt, dass `var_2C` 40 Byte groß ist. Es hat aber keine Informationen über die interne Struktur der

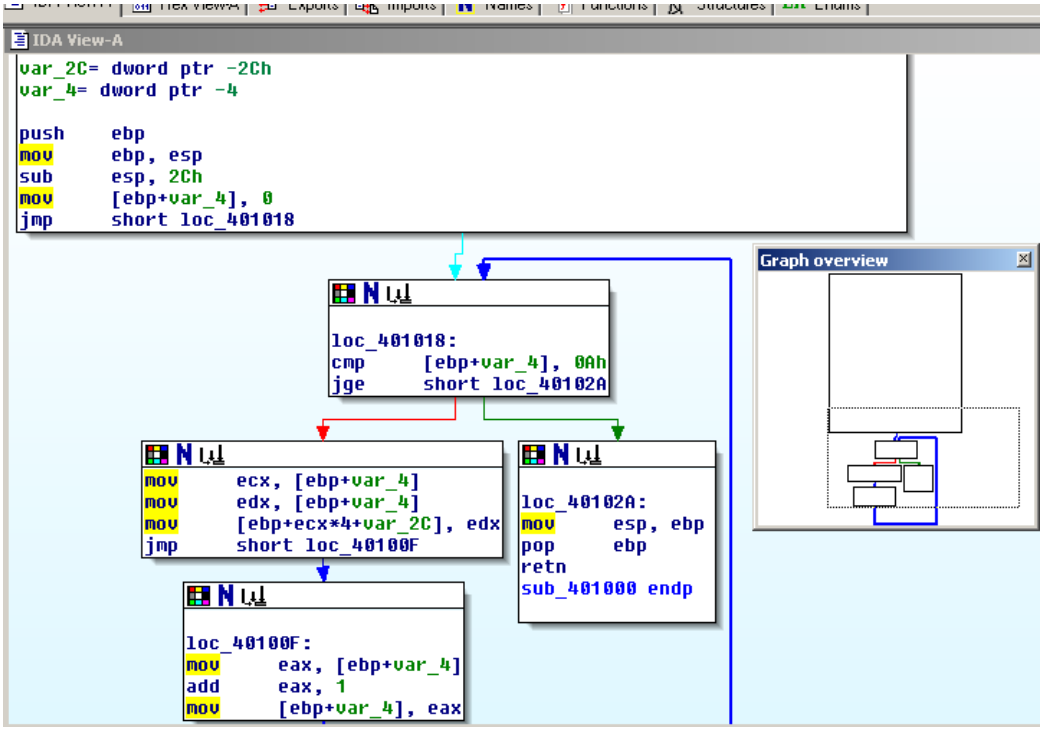


Abbildung 1: `array_example` in IDA Pro

Variablen. Konkret hört die Analysefunktion von IDA Pro dort auf, wo die Komplexität des Programms über die Verwendung von einfachen Datentypen auf globaler oder lokaler Ebene hinausgeht.

Listing 6: Beispiel für die Benutzung einer Strukturvariablen

```
1        struct _pos {
2                int x;
3                int y;
4        } *pos;
5        pos->x = 4;
6        pos->y = 5;
```

Listing 7: Beispiel für die Benutzung einer Strukturvariablen in Assembler

```
1        mov eax, [ebp+pos]
2        mov [eax], 4
3        mov [eax+4], 5
```

In Listing 6 wird die Zuweisung auf eine über einen Pointer referenzierte Strukturvariablen demonstriert. In Listing 7 sieht man die entsprechende

Umsetzung in Assemblercode. Die Typinformationen sind hier verschwunden und IDA Pro weiß in Zeile 2 bereits nicht mehr, dass `eax` der Ort ist, auf den die lokale Variable, die hier das Pseudonym `pos` trägt, verweist. Mit dem Auge ist dieses Beispiel noch sehr einfach nachzuvollziehen. Hat man allerdings mit komplizierteren Strukturen und mehrstufiger indirekter Adressierung zu tun, wie das in der Realität häufig der Fall ist und bei objektorientierter Programmierung noch viel mehr, so ist diese fehlende Analysefunktion von IDA Pro jedem Benutzer schnell ein Dorn im Auge. IDA Pro bietet auch keine Lösungen für viele Funktionen, die für die programmatische und automatisierte Binäranalyse von Interesse sind, für die händische Analyse aber ggf. vernachlässigbar sind. Dynamische Funktionsaufrufe des Prinzips `call *eax` können im allgemeinen nicht aufgelöst werden[1] und fehlen im Call-Graph und bei den von IDA Pro erzeugten Referenzen, erschweren also dem Benutzer die Analyse insbesondere in Fällen, in denen von Herstellerseite (z.B. bei Malware oder Kopierschutzsoftware) Anstrengungen unternommen werden, um die Analyse des Programms zu erschweren. Auch lassen sich Verifizierungsalgorithmen auf einfachem Binärcode nur sehr bedingt ausführen, da zu viele Informationen aus der Hochsprache beim Kompilieren verloren gegangen sind.

## 2.3 Alternativen zu IDA Pro

Wer auf die perfekt integrierte bunten Flussgraphen verzichten kann, findet zahlreiche auch kostenfreie Alternativen zu IDA Pro. Der in früheren Zeiten sehr populäre Kernelmode Disassembler und Debugger SoftICE hat, da die Entwicklung eingestellt wurde, stark an Popularität verloren, ist aber immer noch eine sehr nützliche Software. Zur statischen Analyse eignet sich W32Dasm hervorragend. Im Bereich des Debuggings ist z.B. OllyDbg ein exzellentes Programm, das IDA Pro in einigen Bereichen deutlich überlegen ist, dessen Debugger eher rudimentär ausgeprägt ist und z.B. nicht in der Lage ist Arbeitsspeicher zu verändern (FKL08).

# 3 Codesurfer/x86

Wo IDA Pro aufhört, fangen andere Projekte an. Codesurfer/x86 gebietet sich nun, ein Framework für die Analyse von x86-Executables zu sein und

[1] In gewissen Ausnahmefällen, wenn im selben Programmblock dem Register `eax` ein konstanter Wert zugewiesen und nicht verändert wird, erkennt IDA Pro dies und löst die Referenz korrekt auf.

implementiert u.a. zwei sehr interessante Konzepte, die im Folgenden vorgestellt werden sollen.

## 3.1 Value Set Analysis

Für verschiedenste Analysevorgänge ist es hilfreich, wenn nicht sogar notwendig, zu wissen, welche Werte ein Register oder eine Stelle im Speicher zu welcher Zeit annehmen kann. Value Set Analysis (VSA, RBL06) analysiert nun die Assemblerinstruktionen um genau diese Informationen herauszufinden.

Listing 8: Beispiel für einen indirekten Funktionsaufruf

```
1  mov eax, 80FF2Ch
2  mov eax, [eax+20h]
3  call *eax
```

Im Beispiel aus Listing 8 findet VSA heraus, dass `eax` in Zeile 2 nur 0x80FF2C sein kann und sich die indirekte Adressierung also auf die Adresse 0x80FF4C bezieht. Wenn VSA nun durch Analyse des Rests des Programms bekannt ist, dass an jener Adresse beispielsweise nur zwei verschiedene Werte stehen können, weiß es damit auch, dass an dieser Stelle des Programms nur zwei verschiedene Funktionen aufgerufen werden können. Diese Information ist in händischer Arbeit allein bereits nicht mehr ohne weiteres zu erbringen.

### 3.1.1 Memory Regions

Das Konzept von VSA stößt recht schnell an seine Grenzen, wenn es um lokale Variablen, die auf dem Stack liegen, oder Strukturen, die über `malloc` alloziert wurden, geht, denn sowohl der lokale Stackbereich als auch dynamisch reservierter Speicher haben variable Adressen, für die VSA dann lediglich $[0 \dots \infty]$ annehmen kann, was die Analyse nicht wesentlich voranbringen würde. Wegen dieses Defizits führt VSA das Konzept der Memory Regions ein. Dabei bekommt jeder Stack- und jeder `malloc`-allozierte Bereich seine eigene Memory-Region. Somit beziehen sich die Werte, die eine Variable annehmen kann, nicht mehr nur auf den globalen Adressraum, sondern können auch relativ zu einer dieser Memory Regions bestehen, was die Mächtigkeit von VSA deutlich erhöht.

## 3.2 Aggregate Structure Identification

Wie in Kapitel 2.2 dargelegt, kann IDA Pro nichts über die Struktur von Variablen aussagen. An dieser Stelle setzt die *Aggregate Structure Identification*

(ASI, ebenfalls RBL06) ein, um dem Abhilfe zu schaffen. Basierend auf den Ergebnisse von VSA (siehe Kapitel 3.1) findet ASI heraus, wie auf welche Variable zugegriffen wird. Im `array_example`-Beispiel aus Abbildung 1 wird auf das Array mit der Anweisung mov [ebp+ecx∗4+var_2C], edx zugegriffen, wobei `ecx` und `edx` aus dem Zähler stammen. VSA weiß jetzt bereits, dass dieser Codepfad nur eintreten kann, wenn der Zähler aus dem Wertebereich $[0 \dots 9]$ stammt, folglich gilt selbiges für den Wertebereich der beiden Register. Der Zugriff auf das Array erfolgt damit nur an den Stellen $[0, 4, 8, \dots, 36]$. Basierend auf den Informationen kann jetzt zumindest die Aussage getroffen werden, dass das Array aus 10 `int`s besteht.

Eine Anwendung, die einem bei diesem Beispiel sofort ins Auge sticht, ist herauszufinden, ob es zu *out-of-bounds* Zugriffen auf die Variable kommen kann. Falls ja stellt der dorthin führende Codepfad aller Wahrscheinlichkeit nach eine Sicherheitslücke dar.

In dem durch VSA und ASI gewonnen Modell der Anwendung sind viele Informationen, die sonst frühestens zur Laufzeit in einem Debugger erkennbar sind, schon zur statischen Analyse vorhanden und würden integriert in ein Analyse-Tool wie IDA Pro die Analyse von Anwendungen enorm vereinfachen.

In der Tat basiert das Codesurfer/x86-Framework nach Beschreibungen von (RBLT05) u.a. auf den voranalysierten Daten von IDA Pro und ist für deren Zwecke geeignet, eine Intermediate Repräsentation der Binärdateien zu konstruieren, die dann auch mit den soeben vorgestellten weitergehenden Analysen als Basis für weitere Arbeiten auf dem Binärcode dient, wie z.B. Model-Checking-Algorithmen oder Codeoptimierer.

# 4 VinE

Codesurfer/x86 ist eine schöne Idee mit seinen Analysen und die in den dazugehörigen Papers (vgl. u.a. RBLT05 oder RBL06) beschriebenen Anwendungsmöglichkeiten sind mindestens spannend, allerdings ist Codesurfer/x86 nicht für die Allgemeinheit verfügbar und über die interne Funktionsweise ist nicht viel bekannt. Ein vielversprechenderer Ansatz kommt mit dem Projekt VinE (Uni07), der praktisch die Basis für Codesurfer/x86 und andere Projekte sein könnte.

Die Implementierung von VSA und ASI (wie in den Kapiteln 3.1 und 3.2 beschrieben) wird, falls sie nicht an der ungeheuren Komplexität des x86-Befehlsatzes scheitert, zumindest zu einer Lebensaufgabe. Der x86-Befehlsatz besteht aus hunderten Instruktionen mit verschiedensten Adressierungsmodi und vielen kleinen Seiteneffekten. Vine stellt nun eine Bibliothek zur

Verfügung, die diesen Befehlssatz mit all seinen Details abstrahiert und in eine RISC-artige Intermediate Repräsentation überführt, die mit nur wenigen klar definierten Instruktionen auskommt. Basierend darauf lassen sich mit plötzlich geringem Aufwand Algorithmen für z.B. VSA und ASI implementieren. Andere Anwendungsmöglichkeiten dieses Prinzips sind die Realisierung von Optimierungen oder wie im von (CGP+06) beschrieben die automatische Generierung von Exploits basierend auf durch Fuzzing verursachten Programmabstürzen.

Eine weitere wirklich auf Vine fußende Anwendung wird im folgenden Unterkapitel beschrieben.

## 4.1 Automatic Patch-Based Exploit Generation

Wie bereits in der Einleitung erwähnt wurde, ist einer der Gründe Binäranalyse zu betreiben die Analyse von Patches, um nähere Informationen über die gepatchte Sicherheitslücken zu erhalten. (BPSZ08) beschreiben ein auf der Analyse durch Vine basierendes Verfahren, um aus gegebenen Patches automatisch Exploits für die im Patch behobene Schwachstelle zu generieren. Dies verdeutlicht am besten ein auch aus dem Paper stammendes Beispiel.

Listing 9: Original und Patch

```
1 if(input%2 == 0)                    if(input%2 == 0)
2   goto 3 else goto 5;                 goto 3 else goto 5;
3 s := input + 2;                     s := input + 2;
4 goto 6;                             goto 6;
5 s := input + 3;                     s := input + 3;
6 <nop>                               if(s > input)
7                                       goto 8 else goto ERROR;
8 ptr := realloc(ptr, s);             ptr := realloc(ptr, s);
9 /* use of ptr */                    /* use of ptr */
```

In Listing 9 wird eine Verarbeitungsroutine für einen durch den Benutzer gelieferten Wert gezeigt. Im rechten Teil ist die gepatchte Version, die um eine zusätzliche Prüfung ergänzt wurde. Da mit den Variablen in Computerarithmetik umgegangen wird, kann es für gewisse Werte von `input` in Zeile 3 oder 5 zu einem Integer-Overflow kommen, der durch die neue Prüfung abgefangen wird. Der Ansatz von APEG ist nun, solch zusätzlich eingefügte Prüfungen zu finden, in der Hoffnung, dass Code, der dabei den Fehlerfall auslöst, in der ungepatchten Version exploitable ist. Ein auf der Intermediate Repräsentation von Vine arbeitender Algorithmus findet nun alle Codepfade, die zum Fehlerfall der neuen Prüfung führen und erstellt eine Constraint-Formel basierend auf der schwächsten Vorbedingung die zum Fehlerfall führt.

Die Formel wird dann durch einen *Simple Theorem Prover* (Gan07) gelöst und gibt eine Liste der möglichen Eingaben aus, die zum Fehlerfall führen. Im obigen Beispiel würden für die zwei verschiedenen Codepfad zwei Constraint-Formeln aufgestellt werden:

**(1)** $(input \mod 2 == 0) \wedge (s = input + 2 \mod 2^{32}) \wedge \neg(s > input)$

**(2)** $\neg(input \mod 2 == 0) \wedge (s = input + 3 \mod 2^{32}) \wedge \neg(s > input)$

Für (1) ergibt sich lediglich die Lösung 0xFFFFFFFE ($-2$), wärend (2) die beiden Lösungen [0xFFFFFFFD ($-3$), 0xFFFFFFFF ($-1$)] zulässt. Jeder der drei Eingabewerte führt also dazu, dass in der gepatchten Version der Fehlerfall eintritt bzw. in der Originalversion ein zu kleiner Speicherbereich reserviert wird. Im weiteren Verlauf des verwundbaren Programms würde dann außerhalb dieses Bereichs geschrieben werden und damit ist die Wahrscheinlichkeit sehr hoch, dass sich die Sicherheitslücke zum Ausführen von Schadcode nutzen lässt.

# 5 Zusammenfassungen

IDA Pro und verwandte Programm stellen nur rudimentäre Werkzeuge zur Verfügung um Binärdateien zu analysieren. Automatisiert funktioniert nur das Wenigste, jedoch lassen sich Programme durch Kombination aus der statischen Analyse und einer dynamischen Laufzeitanalyse im Debugger in meist aufwendiger Handarbeit noch recht gut analysieren. Die hier vorgestellten größtenteils theoretischen Techniken könnten bestehende Werkzeuge sehr gut ergänzen und Binäranalysen in vielen Bereichen stark vereinfachen. Auf der anderen Seite erschwert die fortschreitende Abstraktion der Hochsprachen die Analyse oft sehr, da man mit viel *boilerplate code* konfrontiert wird, der für das Programmverhalten unwesentlich für den Analysierenden jedoch undurchschaubar ist. Binäranalyse kann sehr viel effizienter werden, wird aber nie den Dinge rekonstruieren können, die einem das Lesen von Quelltext so erleichtern, wie z.B. Kommentare oder Variablennamen.

# Literatur

**BPSZ08** David Brumley, Pongsin Poosankam, Dawn Song, and Jiang Zheng. Automatic patch-based exploit generation, 2008. [Online; Zugriff Mai 2008].

**CGP+06** Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. Exe: automatically generating inputs of death. In Ari Juels, Rebecca N. Wright, and Sabrina De Capitani di Vimercati, editors, *ACM Conference on Computer and Communications Security*, pages 322–335. ACM, 2006.

**DMS06** Mark Dowd, John McDonald, and Justin Schuh. *The Art of Software Security Assessment: Identifying and Preventing Software Vulnerabilities*. Addison-Wesley Professional, 2006.

**FKL08** Justin Ferguson, Dan Kaminsky, and Jason Larsen. *Reverse Engineering Code with IDA Pro*. Syngress Publishing, 2008.

**Gan07** Vijay Ganesh. A decision procedure for bitvectors and arrays, 2007. [Online; Zugriff Mai 2008].

**RBL06** Thomas Reps, Gogul Balakrishnan, and Junghee Lim. Intermediate-representation recovery from low-level code, 2006. [Online; Zugriff Mai 2008].

**RBLT05** Thomas Reps, Gogul Balakrishnan, Junghee Lim, and T. Teitelbaum. A next-generation platform for analyzing executables, 2005. [Online; Zugriff Mai 2008].

**Uni07** Berkeley University of California. Vine: The bitblaze static analysis component, 2007. [Online; Zugriff Mai 2008].