# Malware-Analyse und Reverse Engineering

8: Statische und dynamische Analysen zur Malware-Erkennung

11.5.2017

Prof. Dr. Michael Engel

# Überblick

**Themen:**

- Statische Analysen zur Malware-Erkennung
- Dynamische Analysen zur Malware-Erkennung

# Malware-Erkennung

**Statische Analysen**

- Signaturen
- Abstract Interpretation

**Dynamische Analysen**

- Anomalieerkennung
- Kontrollflussverfolgung

# Statische Analysen

**Vorige Vorlesung**

- Statische Analyse von Programm-Quellcode zum Finden möglicher Sicherheitslücken

**Jetzt**

- Statische Analyse von Binärprogrammen, um festzustellen, ob ein Programm Malware enthält
- Statische Analyse:
  - Informationen über das Verhalten eines Programms ermitteln, ohne das Programm selbst auszuführen

# Einfache Statische Analyse: Signaturen

HOCHSCHULE COBURG

**Idee**

- Malwarecode besteht aus bestimmter (Maschinen-)Befehlsfolge
- Wenn diese in Programmcode vorkommt $\Rightarrow$ Malware entdeckt
- Grundlage vieler einfacher Virenscanner

**Signatur**

- Folge von Bytes im Programmcode
- Nicht notwendigerweise aufeinanderfolgend
  - Auch komplexere Muster möglich
- *Syntaktische* Methode – kein Verständnis des *Verhaltens* der Malware

# Signaturen: Beispiel

**„Stoned“-Virus von 1987**

- Angeblich von einem Studenten in Neuseeland entwickelt
  - 1989: in Neuseeland und Australien verbreitet
  - 1990: Varianten auf der ganzen Welt verbreitet
- Bootsektor-Virus: befällt Bootsektoren von Disketten und Festplatten (unter MS-DOS!)
  - Kopiert sich beim Booten resident in den Hauptspeicher
  - Infiziert neu eingelegte Disketten (und neu angeschlossene Festplatten – die waren aber damals nicht im Betrieb wechselbar...)
- Malware-Funktionalität:
  - Bei jedem 8. Bootvorgang des PC (von befallener Diskette oder Festplatte) wird der Text *„Your PC is now Stoned!”* ausgegeben

# Stoned “in the wild”

**Ein Virus von 1987 macht 20 Jahre später noch Ärger…**

virus BULLETIN Covering the global threat landscape

Blog Bu

## Boot virus shipped on German laptops

Posted by **Virus Bulletin** on *Sep 14, 2007*

*Aged malware installed on batch of Vista systems.*

A consignment of laptops from German manufacturer *Medion*, sold through German and Danish branches of giant retail chain *Aldi*, have been found to be infected with the boot sector virus 'Stoned.Angelina', first seen as long ago as 1994 and last included on the official WildList in 2001.

According to German sources, anywhere between 10,000 and 100,000 systems may be infected, but as the machines apparently ship without floppy drives the virus is unlikely to spread. The systems come pre-installed with *Windows Vista* and *Bullguard* anti-virus, which will warn of the 'harmless' infection on booting the machines, but is unable to remove it.

HOCHSCHULE COBURG

# Dump des “Stoned”-Virus

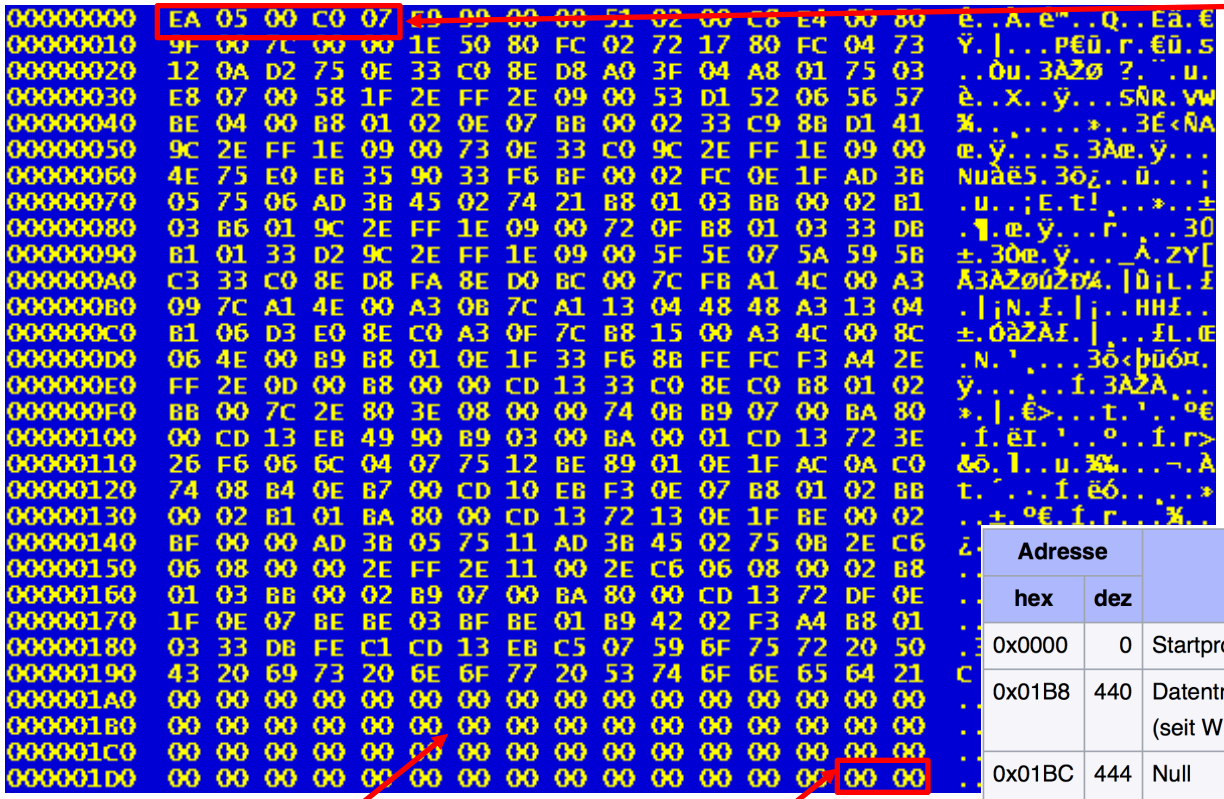
# Analyse des “Stoned”-Virus

**Bootsektor = „Master Boot Record“ (MBR)**

- Erster Sektor einer Diskette/Festplatte bei PCs, die mit klassischem BIOS (nicht UEFI) arbeiten: **512 Bytes**
- Wird von BIOS beim Start des PC von Diskette/ Platte geladen
- Enthält Code, der mit Hilfe von BIOS-Funktionen (SW-Interrupts) Rest des OS lädt

| Adresse | | Funktion / Inhalt | | Größe (Bytes) |
|---|---|---|---|---|
| hex | dez | | | |
| 0x0000 | 0 | Startprogramm (englisch *Bootloader*) (Programmcode) | | 440 |
| 0x01B8 | 440 | Datenträgersignatur (seit Windows 2000) | | 4 |
| 0x01BC | 444 | Null (0x0000) | | 2 |
| 0x01BE | 446 | Partitionstabelle | | 64 |
| 0x01FE | 510 | $55_{hex}$ | Bootsektor-Signatur (wird vom BIOS für den ersten Bootloader geprüft) | 2 |
| 0x01FF | 511 | $AA_{hex}$ | | |
| **Gesamt:** | | | | **512** |

# Der “Stoned”-Bootsektor



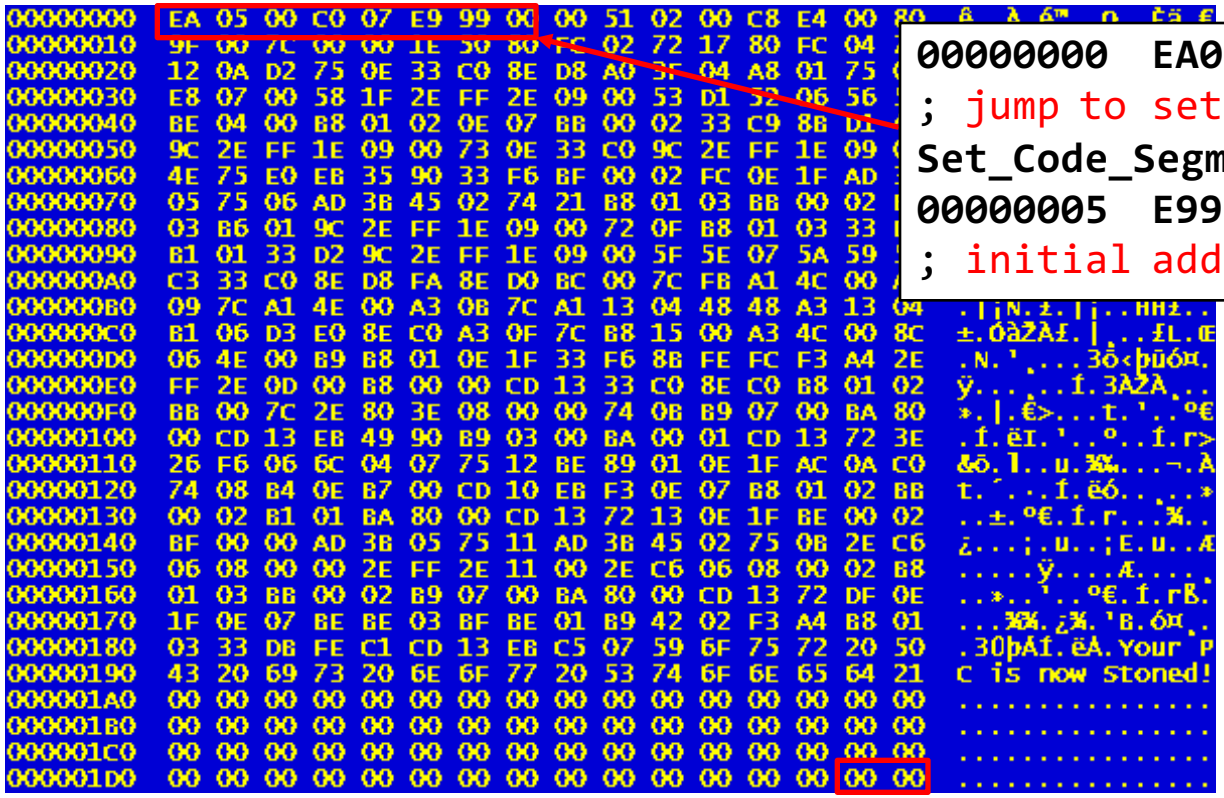

**0x000-004: Sprungbefehl “JMP 07C0:0005”**
(wird oft zur Erkennung eines gültigen Bootsektors verwendet)

**0x1BE-0x1FD: Partitionstabelle (wird freigelassen und später ergänzt)**

**0x1FE/1FF: Signatur des Bootsektors (wird später eingefügt)**

| Adresse hex | Adresse dez | Funktion / Inhalt | | Größe (Bytes) |
|---|---|---|---|---|
| 0x0000 | 0 | Startprogramm (englisch *Bootloader*) (Programmcode) | | 440 |
| 0x01B8 | 440 | Datenträgersignatur (seit Windows 2000) | | 4 |
| 0x01BC | 444 | Null (0x0000) | | 2 |
| 0x01BE | 446 | Partitionstabelle | | 64 |
| 0x01FE | 510 | $55_{hex}$ | Bootsektor-Signatur (wird vom BIOS für den ersten Bootloader geprüft) | 2 |
| 0x01FF | 511 | $AA_{hex}$ | | |
| **Gesamt:** | | | | **512** |

# Analyse: “Stoned”-Bootsektor

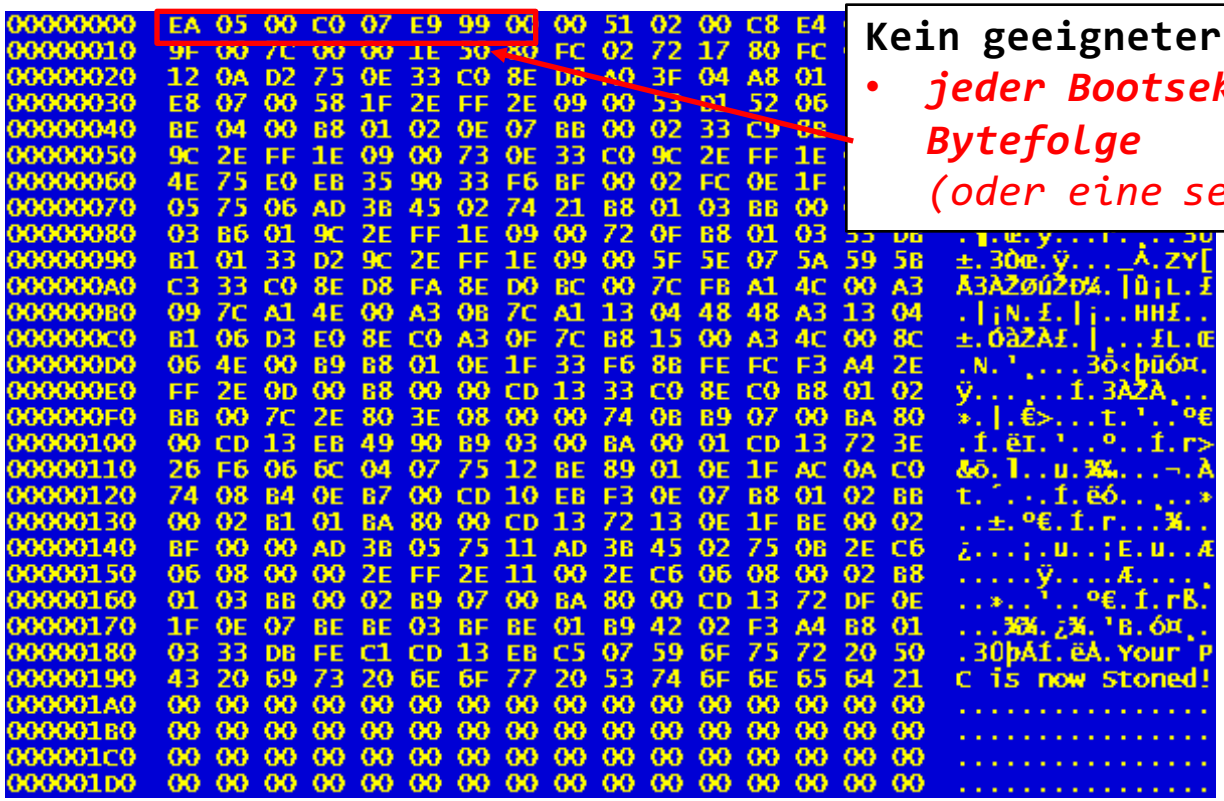



```
00000000  EA0500C007  jmp 07C0:0005
; jump to set correct Code Segment
Set_Code_Segment:
00000005  E99900      jmp Stoned_Start
; initial address of stoned boot virus
```

BIOS lädt Bootsektor immer an Adresse 0x07C0:0000
Virus nutzt absolute Adresse aus, um Daten usw. zu referenzieren

# Erkennung von “Stoned” (1)

## Erkennen von Bytefolge, die eindeutig Virus identifiziert



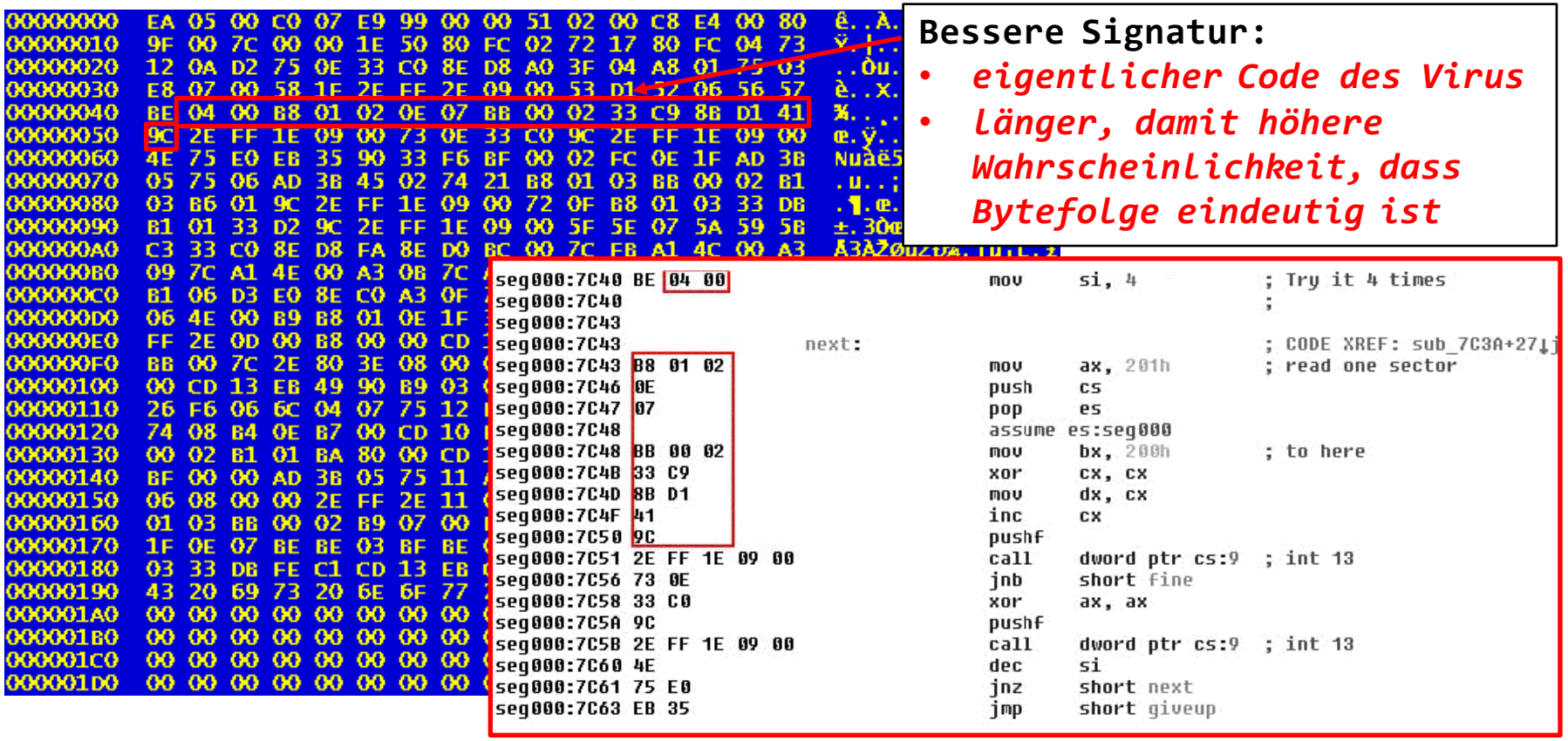

**Kein geeigneter Kandidat:**

- *jeder Bootsektor enthält diese Bytefolge*
  *(oder eine sehr ähnliche)*

http://www.stoned-vienna.com/analysis-of-stoned.html

# Erkennung von “Stoned” (2)

## Erkennen von Bytefolge, die eindeutig Virus identifiziert



# Signaturen: Probleme

**False Positives *(falscher Alarm)***

- Folge von Bytes „legaler“ Bestandteil eines Programms
  - Wie zwischen gut und böse unterscheiden?

**False Negatives**

- Virus ist vorhanden, wird aber nicht erkannt
- Ursache: kleine Änderung des Virus-Codes
  - Andere Adressen, Verwendung anderer Register, ...
- *Polymorphe Malware*
  - Viren, die sich bei jeder Infektion selbst verändern, um Signaturprüfung zu entgehen

# Signaturen:
# Beispiel für false positives

**„Stoned“-Virus und Bitcoins**

- Virus-Signatur von „Stoned“ wurde im Mai 2014 in die Blockchain von Bitcoin eingeführt
  - Blockchain = Daten, *kein ausführbares Programm*
  - Virenscanner scannt Dateien auf Festplatte (+evtl. Inhalt des Hauptspeichers), kann nicht zwischen Code und Daten unterscheiden
- Folge: Microsoft Security Essentials erkannte Blockchain als Virus und wollte Datei löschen
  - Bitcoin-SW lädt Blockchain neu
  - Virenscanner erkennt „Virus“ erneut...

http://thehackernews.com/2014/05/microsoft-security-essential-found.html

# Weitere Probleme von Signaturen

**Mit jedem neuen Virus steigt Größe der Signatur-Datenbank**

- Aufwendigerer Scanvorgang – immer mehr Signaturen müssen überprüft werden

**Signaturdatenbank muss aktuell gehalten werden**

- Neu auftretende Malware wird nicht erkannt, da Hersteller von Virenscannern diese erst analysieren müssen
- Danach wird Signaturdatenbank aktualisiert
- ...und dann (irgendwann) verteilt...

# Polymorphe Malware

**Ziel: Umgehen der Erkennung durch Signaturen**

- Änderung von Bytes in Code des Programms

- Absichtlich: *Mutation* oder *Obfuskation*
  - Einfügen von „**NOP**“: **0x90** = XCHG EAX, EAX oder anderen Instruktionen, die die Semantik nicht ändern
  - Ändern der Codereihenfolge durch Sprungbefehle
  - Ersetzen von (Folgen von) Maschineninstruktionen durch semantisch identische Instruktionen

- ...oder auch eher unabsichtlich durch Adressänderungen im Code, Compileroptimierungen bei „Update“ eines Virus usw.

# Malware-Obfuskation: Beispiel für Einfügen von NOPs (1)

HOCHSCHULE COBURG

## Einfügen von „NOP“-Operationen

- Beibehaltung der Semantik
- aber: andere Signatur

**Code aus Chernobyl/CIH-Virus:**
Mihai Christodorescu and Somesh Jha, *“Static analysis of executables to detect malicious patterns”*, USENIX Security Symposium - Volume 12, 2003

*Original code*

```
E8 00000000     call 0h
5B              pop ebx
8D 4B 42        lea ecx, [ebx + 42h]
51              push ecx
50              push eax
50              push eax
0F01 4C 24 FE   sidt [esp - 02h]
5B              pop ebx
83 C3 1C        add ebx, 1Ch
FA              cli
8B 2B           mov ebp, [ebx]
```

*Obfuscated code*

```
E8 00000000     call 0h
5B              pop ebx
8D 4B 42        lea ecx, [ebx + 45h]
90              nop
51              push ecx
50              push eax
50              push eax
90              nop
0F01 4C 24 FE   sidt [esp - 02h]
5B              pop ebx
83 C3 1C        add ebx, 1Ch
90              nop
FA              cli
8B 2B           mov ebp, [ebx]
```

# Malware-Obfuskation: Beispiel für Einfügen von NOPs (2)

HOCHSCHULE COBURG

**Andere Signatur benötigt**

- Einfügen von NOPs ist aber an beliebigen Stellen möglich...

*Original code*

```
E8 00000000     call 0h
5B              pop ebx
8D 4B 42        lea ecx, [ebx + 42h]
51              push ecx
50              push eax
50              push eax
0F01 4C 24 FE   sidt [esp - 02h]
5B              pop ebx
83 C3 1C        add ebx, 1Ch
FA              cli
8B 2B           mov ebp, [ebx]
```

*Obfuscated code*

```
E8 00000000     call 0h
5B              pop ebx
8D 4B 42        lea ecx, [ebx + 45h]
90              nop
51              push ecx
50              push eax
50              push eax
90              nop
0F01 4C 24 FE   sidt [esp - 02h]
5B              pop ebx
83 C3 1C        add ebx, 1Ch
90              nop
FA              cli
8B 2B           mov ebp, [ebx]
```

*Signature*

```
E800 0000 005B 8D4B 4251 5050
0F01 4C24 FE5B 83C3 1CFA 8B2B
```

*New signature*

```
E800 0000 005B 8D4B 4290 5150
5090 0F01 4C24 FE5B 83C3 1C90
FA8B 2B
```

# Flexible Erkennung von durch „NOP“ mutierten Varianten

**Signatur wird zu *regulärem Ausdruck***

- Möglichkeit der Beschreibung optionaler und wiederholter Bytes und Bytefolgen notwendig
- Aufwendiger zu analysieren

*Signature*

```
E800 0000 005B 8D4B 4251 5050
0F01 4C24 FE5B 83C3 1CFA 8B2B
```

*New signature*

```
E800 0000 005B 8D4B 4290 5150
5090 0F01 4C24 FE5B 83C3 1C90
FA8B 2B
```

An den mit (90)* markierten Stellen könnten möglicherweise eine oder mehrere NOP-Instruktionen zusätzlich auftauchen

***Flexible Signatur***

| | | | |
|---|---|---|---|
| E800 | 0000 | 00(90)* | 5B(90)* |
| 8D4B | 42(90)* | 51(90)* | 50(90)* |
| 50(90)* | 0F01 | 4C24 | FE(90)* |
| 5B(90)* | 83C3 | 1C(90)* | FA(90)* |
| 8B2B | | | |

# Malware-Obfuskation: Beispiel für Code-Umsortierung

HOCHSCHULE COBURG

## Code wird im Speicher umgeordnet

- Verwenden von Sprungbefehlen, um Kontrollfluss zu verändern

*Original code*

```
call 0h
pop ebx
lea ecx, [ebx+42h]
push ecx
push eax
push eax
sidt [esp - 02h]
pop ebx
add ebx, 1Ch
cli
mov ebp, [ebx]
```

*Code obfuscated through code transposition*

```
      call 0h
      pop ebx
      jmp S2
S3:   push eax
      push eax
      sidt [esp - 02h]
      jmp S4
      add ebx, 1Ch
      jmp S6
S2:   lea ecx, [ebx+42h ]
      push ecx
      jmp S3
S4:   pop ebx
      cli
      jmp S5
S5:   mov ebp, [ebx]
```

# Malware-Obfuskation: Verwendung semantisch identischer Instr.

**Instruktionen werden ersetzt**

- Semantik wird beibehalten
- z.B. push durch explizite SP-Operation + Store ersetzen

*Original code*

```
call 0h
pop ebx
lea ecx, [ebx+42h]
push ecx
push eax
push eax
sidt [esp - 02h]
pop ebx
add ebx, 1Ch
cli
mov ebp, [ebx]
```

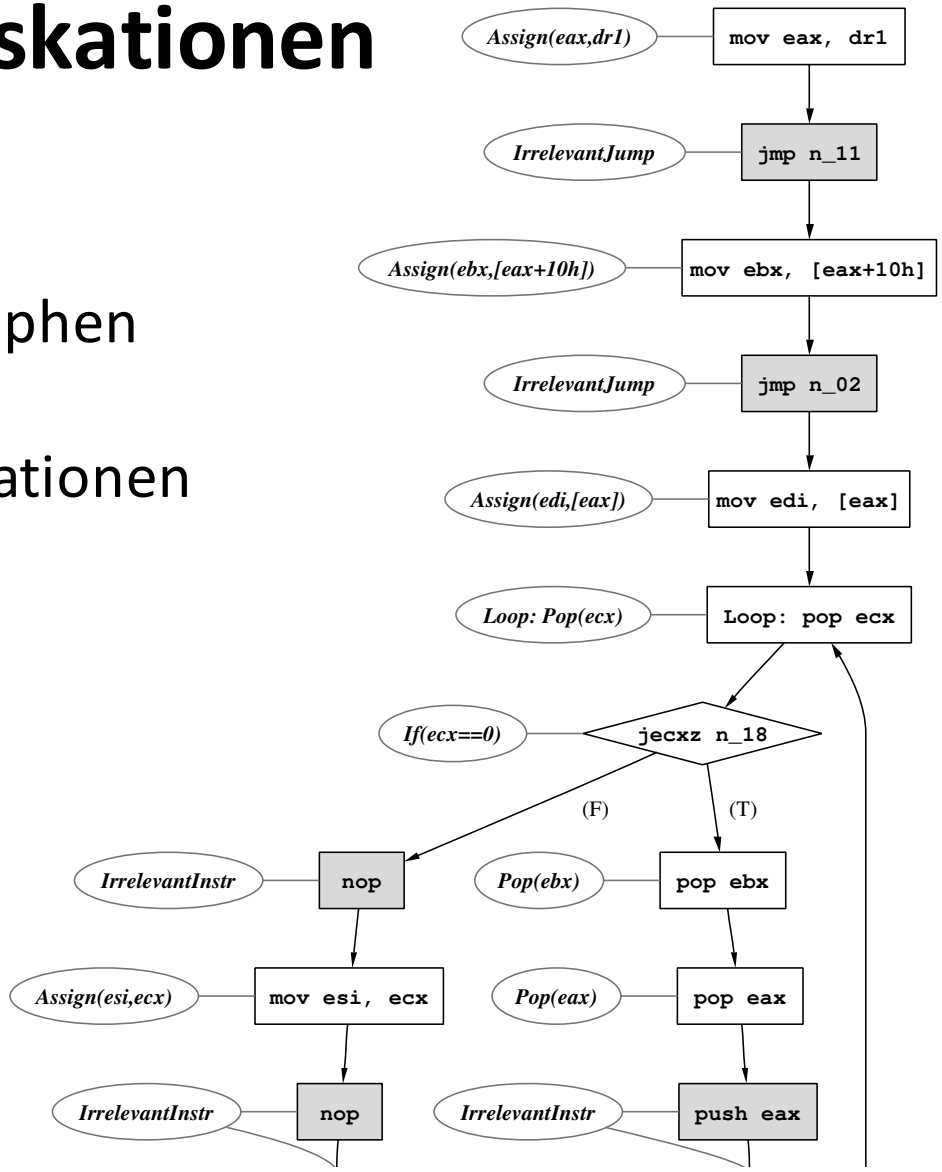
*Code obfuscated through instruction substitution*

```
call 0h
pop ebx
lea ecx, [ebx+42h]
sub esp, 03h
sidt [esp - 02h]
add [esp], 1Ch
mov ebx, [esp]
inc esp
cli
mov ebp, [ebx]
```

# Erkennung von Obfuskationen

**Vorgehensweise:**

- Erstellen des Kontrollflussgraphen des Maschinencodes
- Markieren „unnötiger“ Operationen (NOP, JMP, etc.)
- Vergleich mit bekanntem Kontrollflussgraphen (*Graph-Isomorphismus*)



# Überblick Obfuskationstechniken

## Kombination der verschiedenen Techniken möglich

- Extrem hoher Erkennungsaufwand

*Original code*

```
call 0h
pop ebx
lea ecx, [ebx+42h]
push ecx
push eax
push eax
sidt [esp - 02h]
pop ebx
add ebx, 1Ch
cli
mov ebp, [ebx]
```

*Code obfuscated through dead-code insertion*

```
call 0h
pop ebx
lea ecx, [ebx+42h]
nop                    (*)
nop                    (*)
push ecx
push eax
inc eax                (**)
push eax
dec [esp - 0h]         (**)
dec eax                (**)
sidt [esp - 02h]
pop ebx
add ebx, 1Ch
cli
mov ebp, [ebx]
```

*Code obfuscated through code transposition*

```
        call 0h
        pop ebx
        jmp S2
S3:     push eax
        push eax
        sidt [esp - 02h]
        jmp S4
        add ebx, 1Ch
        jmp S6
S2:     lea ecx, [ebx+42h ]
        push ecx
        jmp S3
S4:     pop ebx
        cli
        jmp S5
S5:     mov ebp, [ebx]
```

*Code obfuscated through instruction substitution*

```
call 0h
pop ebx
lea ecx, [ebx+42h]
sub esp, 03h
sidt [esp - 02h]
add [esp], 1Ch
mov ebx, [esp]
inc esp
cli
mov ebp, [ebx]
```

| | | Norton® Antivirus 7.0 | McAfee® VirusScan 6.01 | Command® Antivirus 4.61.2 |
|---|---|---|---|---|
| Chernobyl | original | ✓ | ✓ | ✓ |
| | obfuscated | ✗[1] | ✗[1,2] | ✗[1,2] |
| z0mbie-6.b | original | ✓ | ✓ | ✓ |
| | obfuscated | ✗[1,2] | ✗[1,2] | ✗[1,2] |
| f0sf0r0 | original | ✓ | ✓ | ✓ |
| | obfuscated | ✗[1,2] | ✗[1,2] | ✗[1,2] |
| Hare | original | ✓ | ✓ | ✓ |
| | obfuscated | ✗[1,2] | ✗[1,2] | ✗[1,2] |

**Kommerzielle Virenscanner erkannten mutierte Varianten nicht!**

# Polymorphe Malware: Beispiel für Adreßänderungen

HOCHSCHULE COBURG

**Verwendung einer statisch gelinkten Library**

- Identischer Assembler-Quelltext
- Unterschiedliche absolute Adressen durch anderes Speicherlayout
- Komplexere Signaturerkennung notwendig

```
6A 58                   push  58h
68 70 E4 40 00          push  offset unk_40E470
E8 9A 04 00 00          call  __SEH_prolog4
33 DB                   xor   ebx, ebx
89 5D E4                mov   [ebp+var_1C], ebx
89 5D FC                mov   [ebp+ms_exc.disabled], ebx
8D 45 98                lea   eax, [ebp+StartupInfo]
50                      push  eax
FF 15 C0 B0 40 00       call  ds:GetStartupInfoA
```

```
6A 58                   push  58h
68 60 0A 55 00          push  offset unk_550A60
E8 BB 05 00 00          call  __SEH_prolog4
33 DB                   xor   ebx, ebx
89 5D E4                mov   [ebp+var_1C], ebx
89 5D FC                mov   [ebp+ms_exc.disabled], ebx
8D 45 98                lea   eax, [ebp+StartupInfo]
50                      push  eax
FF 15 6C 11 51 00       call  ds:GetStartupInfoA
```

# Polymorphe Malware: Beispiel für Compileroptimierungen

**Compiler eliminiert nicht benötigte Maschineninstruktionen**

- Oft identischer (z.B. C-)Quelltext
- Optimierungs-phase des Compilers erkennt, dass einige erzeugte Maschinen-instruktionen in Kontext nicht benötigt werden
- Entfernt diese aus Programm

```
55                      push   ebp
8B EC                   mov    ebp, esp
51                      push   ecx
8B 45 08                mov    eax, [ebp+arg_0]
89 45 FC                mov    [ebp+var_4], eax
83 7D FC 09             cmp    [ebp+var_4], 9
0F 87 B0 00 00 00       ja     loc_4010C4
8B 4D FC                mov    ecx, [ebp+var_4]
FF 24 8D D8 10 40+      jmp    ds:off_4010D8[ecx*4]
```

```
55                      push   ebp
8B EC                   mov    ebp, esp
8B 45 08                mov    eax, [ebp+arg_0]
83 F8 09                cmp    eax, 9
0F 87 A7 00 00 00       ja     loc_4010B6
FF 24 85 C8 10 40+      jmp    ds:off_4010C8[eax*4]
```

# Komplexere Statische Analyse: Abstrakte Interpretation

HOCHSCHULE COBURG

**Idee**

- Ermitteln der Semantik eines Programms, ohne es auszuführen
- Erkennung von Anomalien

**Methode: Abstrakte Interpretation**

- Informationen über das Verhalten von Programmen (Analyse der Semantik) erhalten, indem man von Teilen des Programms abstrahiert und die Anweisungen Schritt für Schritt nachvollzieht (Interpretation)

Patrick Cousot, Radhia Cousot: "*Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints*", Sixth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 1977

# Abstrakte Interpretation

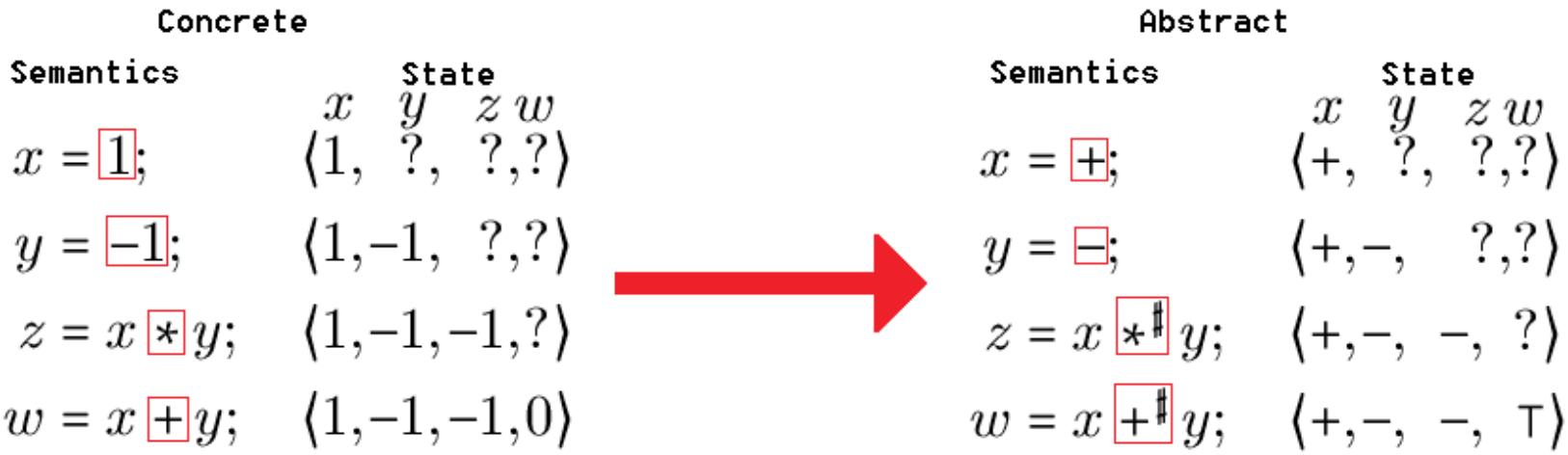
**Vorgehensweise**

- Konzentration auf Teilaspekte der Ausführung der Anweisungen
- Gezieltes Weglassen von Information (Abstraktion)
- Ergebnis: Näherung an die Programmsemantik
  - Kann für gewünschten Zweck vollkommen ausreichend sein
- Viele Eigenschaften von Programmen sind nicht berechenbar
  - Man kann kein Programm angeben, welches sie in endlicher Zeit mit endlichen Speicherresourcen für beliebige Programme berechnet (→ Halteproblem)
- Approximation (Weglassen einiger Informationen) ermöglicht einige weitere Analysen

# Abstrakte Interpretation: Beispiel

**Abstrakte Interpretation ist komplexer mathematischer Ansatz**

- Die Grundprinzipien sind aber einfach
- Beispiel: Bestimmen des Vorzeichens von Variablen




Ersetzen von **konkretem Zustand** durch **abstrakten Zustand** und **konkreter Semantik** durch **abstrakte Semantik**

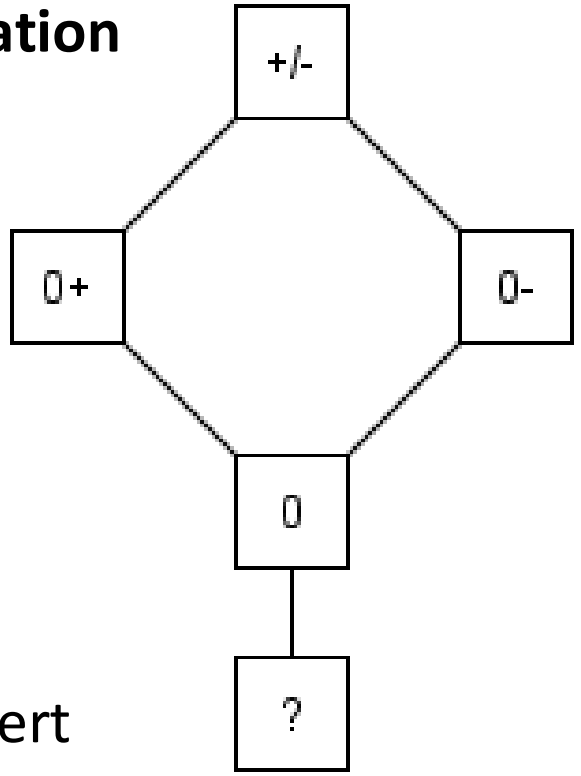
# Abstrakte Interpretation: Zustandsabstraktion

HOCHSCHULE COBURG

**Verschiedene Methoden der abstr. Interpretation verwenden unterschiedliche Abstraktionen**

- Für Vorzeichenanalyse ist jede Variable:
  - unbekannt: positiv oder negativ (+/-)
  - positiv: x >= 0 (0+)
  - negativ: x <= 0 (0-)
  - null (0)
  - uninitialisiert (?)

- Alle weiteren Informationen werden ignoriert
  - Hier z.B. der konkrete Wert der Variablen



# Abstrakte Interpretation: Semantikabstraktion (1)

HOCHSCHULE COBURG

**Abstrakte Interpretation befolgt Vorzeichenregeln für die Multiplikation:**

- „+“ * „+“ = „+“
- „–“ * „–“ = „+“
- „–“ * „+“ = „+“ * „–“ = „–“

| * | ? | 0 | 0+ | 0- | +/- |
|---|---|---|---|---|---|
| ? | +/- | 0 | +/- | +/- | +/- |
| 0 | 0 | 0 | 0 | 0 | 0 |
| 0+ | +/- | 0 | 0+ | 0- | +/- |
| 0- | +/- | 0 | 0- | 0+ | +/- |
| +/- | +/- | 0 | +/- | +/- | +/- |

- Hinweis: diese Regeln beziehen sich auf mathematische ganze Zahlen, im Computer repräsentierte Zahlen können überlaufen und unabsichtlich ihr Vorzeichen ändern!

# Abstrakte Interpretation: Semantikabstraktion (2)

HOCHSCHULE COBURG

**Abstrakte Interpretation benötigt Vorzeichenregeln für die Addition:**

- „+“ + „+“ = „+“
- „–“ + „–“ = „–“
- „–“ + „+“ = „+“ + „–“ = „**?**“

| + | ? | 0 | 0+ | 0- | +/- |
|---|---|---|---|---|---|
| ? | +/- | +/- | +/- | +/- | +/- |
| 0 | +/- | 0 | 0+ | 0- | +/- |
| 0+ | +/- | 0+ | 0+ | +/- | +/- |
| 0- | +/- | 0 | 0- | 0+ | +/- |
| +/- | +/- | 0 | +/- | +/- | +/- |

- Beispiel:
  - –5 + 5 ⇒ konkretes Ergebnis 0
  - –6 + 5 ⇒ konkretes Ergebnis –1
  - –5 + 6 ⇒ konkretes Ergebnis 1

# Konkrete Anwendung der abstrakten Interpretation: Sprunganalyse

**Herausfinden der Struktur einer compilierten „switch“-Anweisung**

- Compiler verwenden Sprungtabellen für effizienten Code

```
switch(x)
{
  case 0:  /* ... */ break;
  case 1:  /* ... */ break;
  /* ... */
  case 9:  /* ... */ break;
  default: /* ... */ break;
}
```

```
cmp     eax, 9          ; switch 10 cases
ja      loc_4010B6      ; default
jmp     ds:off_4010C8[eax*4] ; switch jump

off_4010C8 dd offset loc_401016
dd offset loc_401026
dd offset loc_401036
dd offset loc_401046
dd offset loc_401056
dd offset loc_401066
```

```
switch(x)
{
  case 0: case 2: case 4: case 6:
  case 8:  printf("even\n"); break;

  case 1: case 3: case 5: case 7:
  case 9:  printf("odd\n");  break;

  default: printf("other\n"); break;
}
```

```
cmp     eax, 9          ; switch 10 cases
ja      short loc_401129 ; default
movzx   eax, ds:index_table[eax]
jmp     ds:off_40113C[eax*4] ; switch jump


off_40113C      dd offset loc_401109    ; DATA
                dd offset loc_401119    ; jump
index_table     db      0,      1,      0,      1
                db      0,      1,      0,      1
                db      0,      1
```

# Sprunganalyse (2)

**Switch mit weit verteilten Werten**

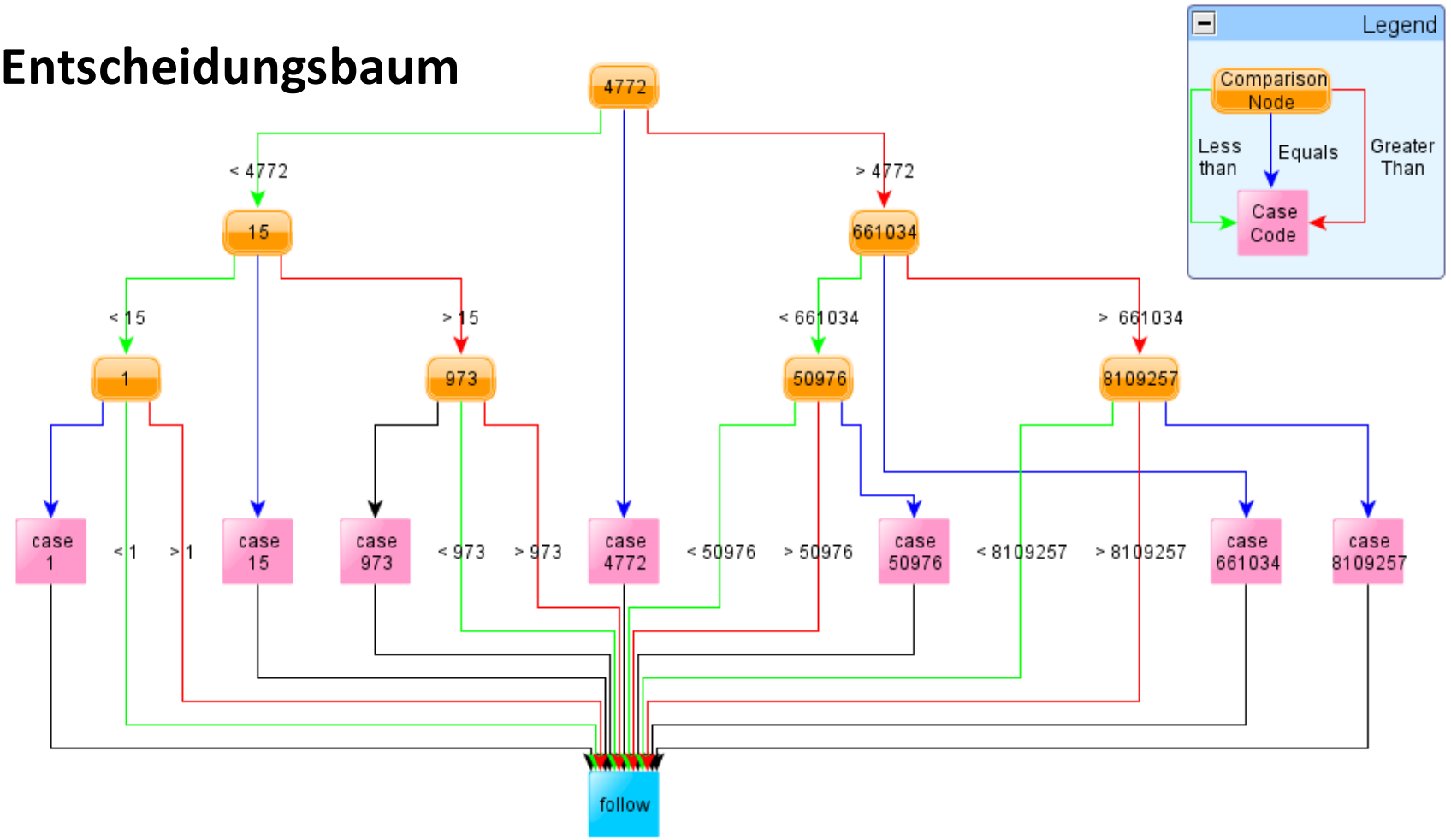
- Sprungtabelle wäre nur dünn besetzt (viele Nullen)
- Ersetzung durch Folgen von „if“-Abfagen ist ineffizient
  - Viele Abfragen und bedingte Sprünge notwendig: O(N)
  - stattdessen verwenden Compiler Entscheidungsbäume, Aufwand: O(log(n))

```
switch(x)
{
  case       1: /*1*/ break;
  case      15: /*2*/ break;
  case     973: /*3*/ break;
  case    4772: /*4*/ break;
  case   50976: /*5*/ break;
  case  661034: /*6*/ break;
  case 8109257: /*7*/ break;
}
```

```
if(x ==       1) /*1*/ else
if(x ==      15) /*2*/ else
if(x ==     973) /*3*/ else
if(x ==    4772) /*4*/ else
if(x ==   50976) /*5*/ else
if(x ==  661034) /*6*/ else
if(x == 8109257) /*7*/;
```

# Sprunganalyse (3)

## Entscheidungsbaum



# Sprunganalyse: Assemblercode (1)

## Implementierung des Entscheidungsbaums

```
mov     eax, [ebp+arg_0]
cmp     eax, 11270h
jg      short loc_40167B
jz      short loc_40166B
cmp     eax, 3C3h
jg      short loc_401654
jz      short loc_401644
dec     eax
jz      short loc_401634
sub     eax, 11
jnz     loc_4016BE
push    offset a00000012
call    ds:__imp__printf
```

Vergleich mit Entscheidungswert

Größer? Springen!

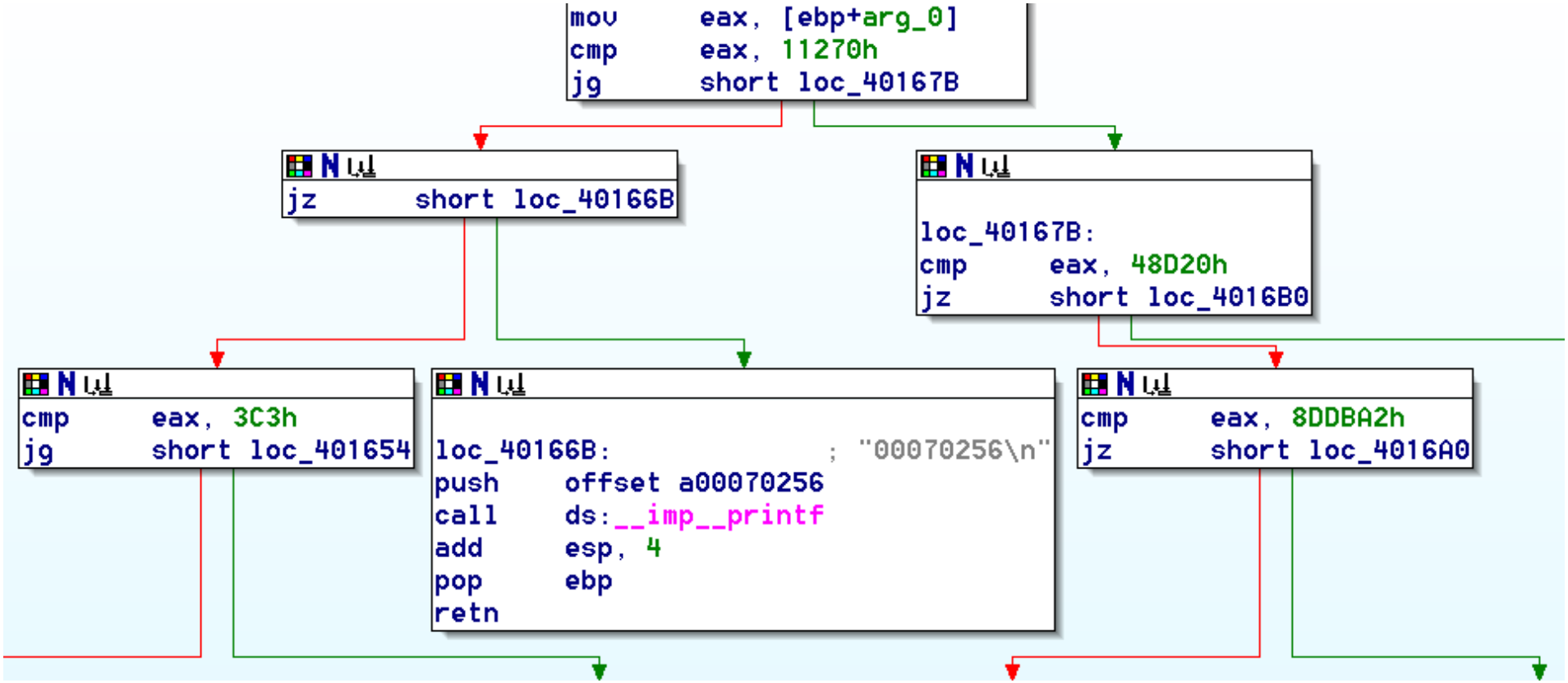
Gleich? Springen!

Sonst: weiter im Code, nächsten Vergleich ausführen

Kleines Problem:
eax wird von markierten Instruktionen verändert

# Sprunganalyse: Assemblercode (2)

## Kontrollflussdarstellung:



# Abstraktion der Entscheidungen im switch-Statement

- **Erkenntnis:**
- wichtig: welche *Wertebereiche* führen zur Ausführung welches case-Falls?

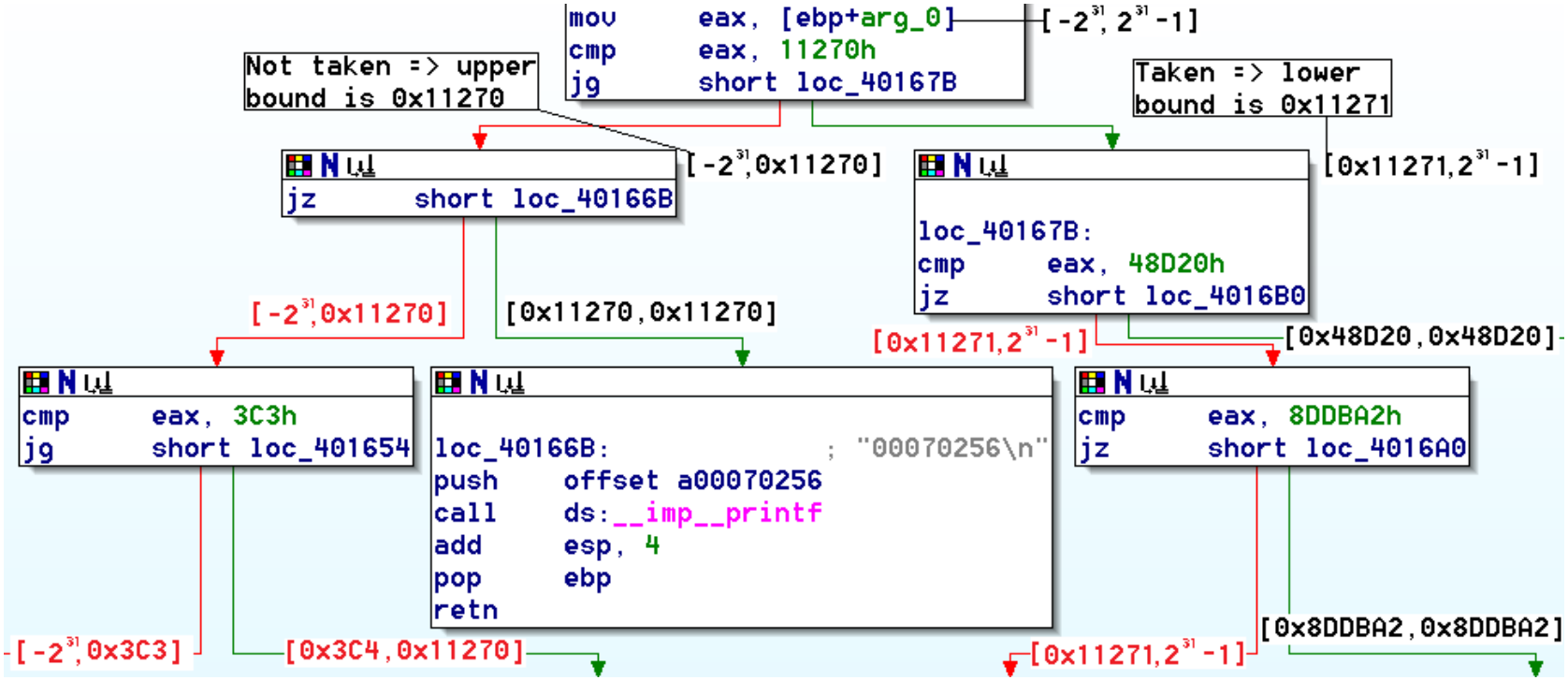
- **Datenabstraktion**:
  - Intervalle [l, u] mit l <= u
- switch implementiert mit **sub, dec** und **cmp**-Instruktionen – alles Subtraktionen – und bedingten Sprüngen
- **Semantikabstraktion:**
  - Beibehalten der Subtraktionen
  - Verzweigung bei branch-Befehlen

```
switch(x)
{
  case       1: /*1*/ break;
  case      15: /*2*/ break;
  case     973: /*3*/ break;
  case    4772: /*4*/ break;
  case   50976: /*5*/ break;
  case  661034: /*6*/ break;
  case 8109257: /*7*/ break;
}
```

```
mov     eax, [ebp+arg_0]
cmp     eax, 11270h
jg      short loc_40167B
jz      short loc_40166B
cmp     eax, 3C3h
jg      short loc_401654
jz      short loc_401644
dec     eax
jz      short loc_401634
sub     eax, 11
jnz     loc_4016BE
push    offset a00000012
call    ds:__imp__printf
```

# Analyseergebnisse switch-Statement

- Keine initiale Information über arg_0 vorhanden
- Jeder Pfad durch Code schränkt mögliche Werte für arg_0 ein
- Wert an Blättern (case-Labels): Wert/einfacher Wertebereich



# Dynamische Analysen

**Überwachung des Programmverhaltens zur Laufzeit**

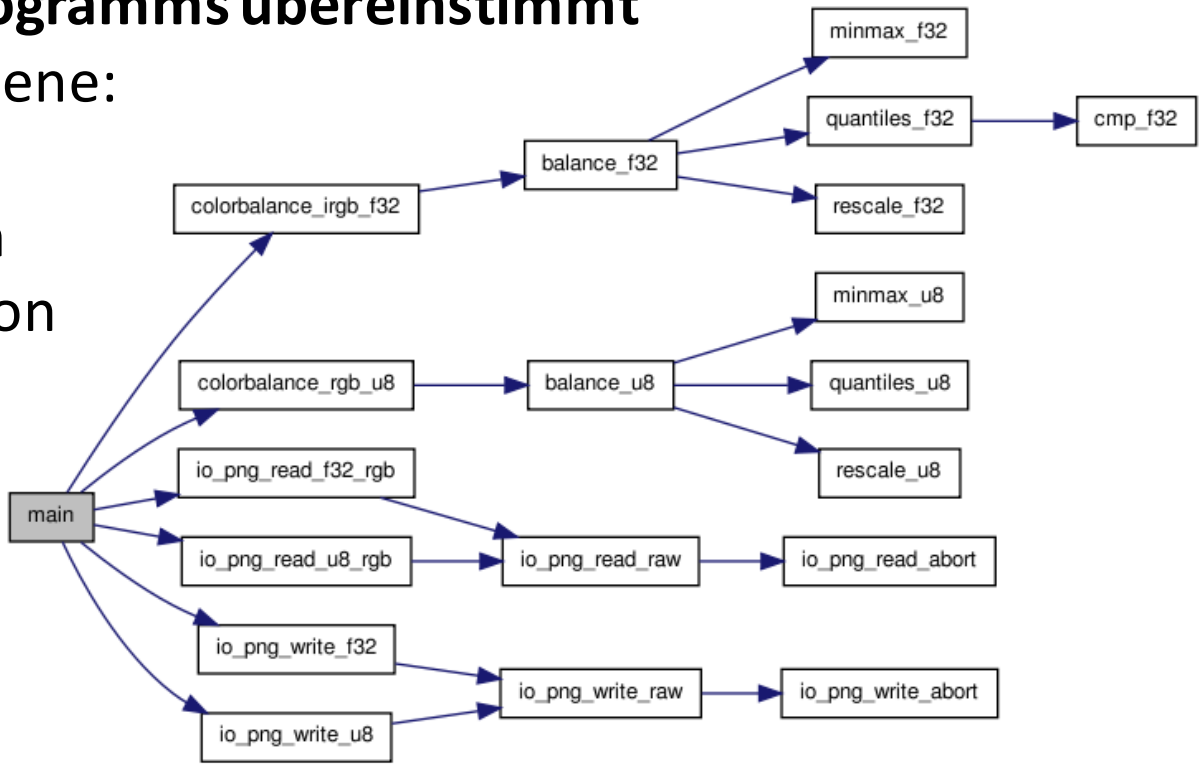
- Statische Analysen benötigen Signaturen
  - Können neue und evtl. mutierte Malware nicht erkennen
- Idee der dynamischen Analyse:
  - Betrachten des Verhaltens eines Programms zur Laufzeit

- Semantische Methode – Analyse des *Verhaltens* eines Programms

# Dynamische Analyse: Kontrollflussverfolgung

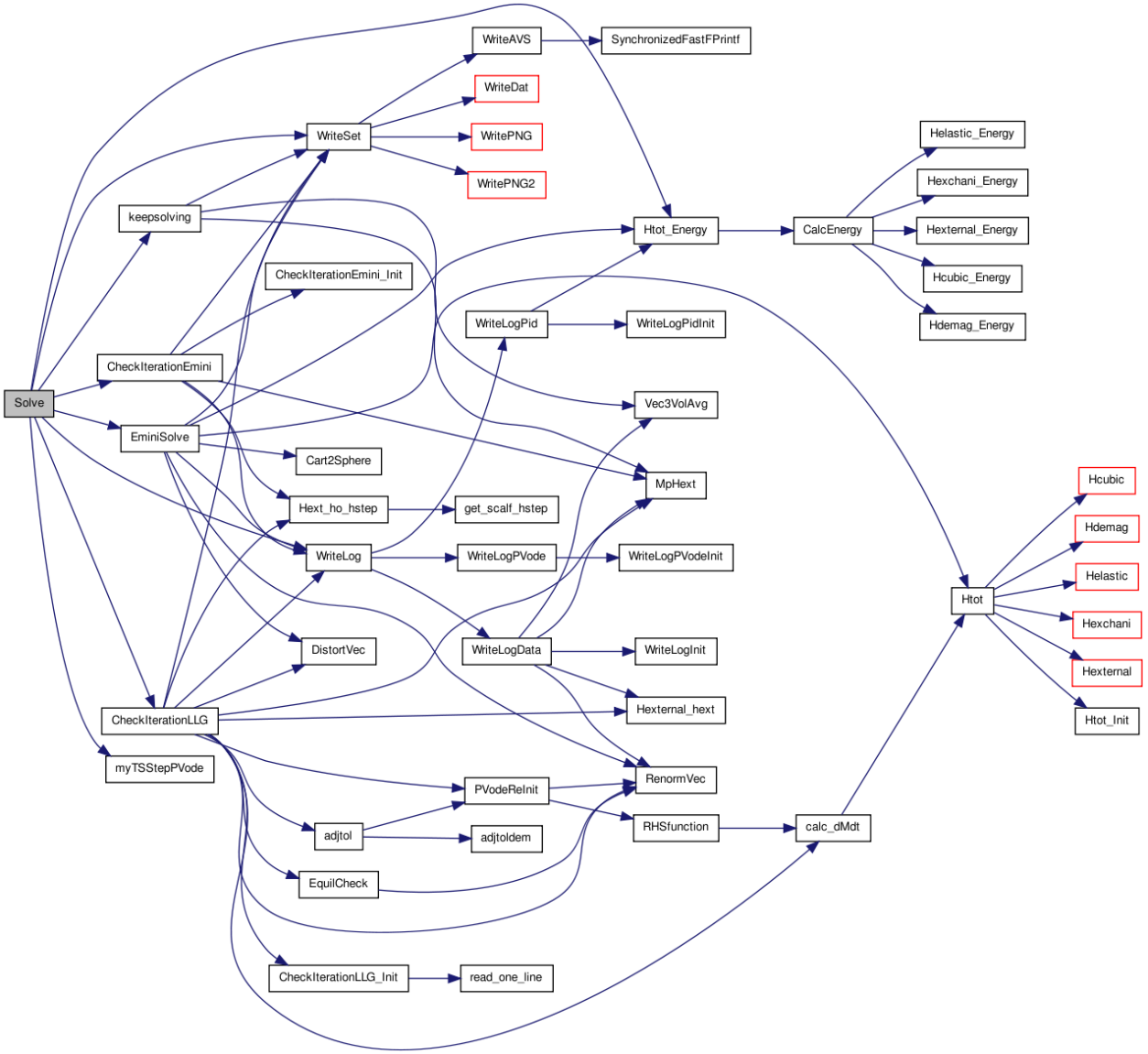
**Idee: Überprüfen, ob tatsächlich stattgefundener Kontrollfluss mit Semantik eines Programms übereinstimmt**

- z.B. auf Funktionsebene: Überprüfen von Rücksprungadressen bei Eintritt in Funktion
- Vergleich mit Funktionsaufruf-graph
- Entdeckt z.B. manipulierte Rück-sprungadressen



# Komplexer Funktions-aufrufgraph

**Komplexere Programme machen Analyse aufwendig…**



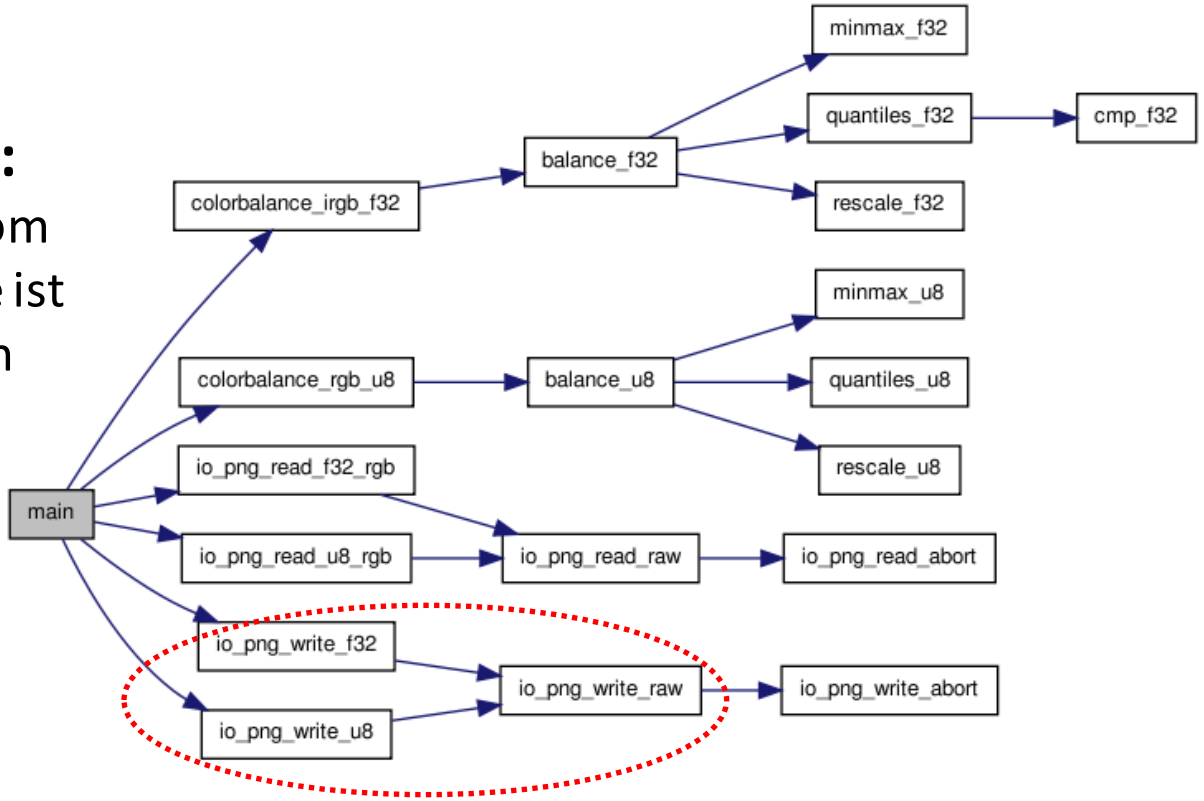
# Implementierung der Kontrollflussverfolgung

**Idee: Erstellen einer Repräsentation des Aufrufgraphen**

- Automatisch durch Compiler erstellbar

**Bei Eintritt in Funktion:**

- Überprüfen, ob Kante vom Aufrufer (Returnadresse ist auf dem Stack!) in Graph enthalten ist
- z.B. darf (kann?) „io_png_write_raw“ **nur** von „io_png_write_f32“ und „io_png_write_u8“ aufgerufen werden



# Probleme der Kontrollflussverfolgung

**Dynamischer Code**

- Verwendung von Funktionszeigern/Sprungtabellen: Compiler kann zur Übersetzungszeit keine vollständige Liste der Funktionen erstellen, die zum Aufruf einer Funktion *f* führen
  - Also ist der Aufrufgraph unvollständig

- Lösung:
  - Zeigeranalyse wäre erforderlich... ✘

# Fazit

**Statische Methoden zur Erkennung von Malware**

- Signaturchecks sind problematisch
  - Mutierende Malware, mangelnde Aktualität
- Aufwendigere statische Methoden existieren
  - Abstrakte Interpretation

**Dynamische Analysen zur Erkennung von Malware**

- Kontrollflussanalyse durch Überwachung von Funktionsaufrufen
- mehr: nächste Woche...