Malware-Analyse und Reverse Engineering

9: Polymorphe Viren und selbstmodifizierender Code

18.5.2017

Prof. Dr. Michael Engel

Überblick

Themen:

- Erschweren des Disassemblierens von Code
- Ver- und Entschlüsselung von Code
- Selbstmodifizierender Code

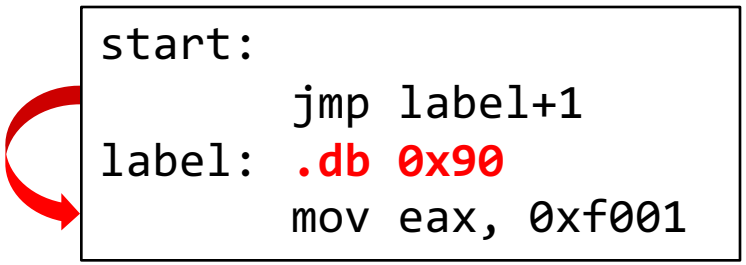
Erschweren des Disassemblierens von Code (1)

Wir kennen Kontrollflussverfolgung

- ...nicht nur auf Funktionsebene möglich, sondern auch bei bedingten und unbedingten Sprüngen

⇒ Ablauf des Codes kann nachverfolgt werden, Virusverhalten damit reverse engineerbar

Codebeispiel:



Erschweren des Disassemblierens von Code (2)

Codebeispiel:

Disassembler (objdump) decodiert 0x90 korrekt als „nop“

```
start:
        jmp label+1
label:  .db 0x90
        mov eax, 0xf001
```

```
08048080 <start>:
 8048080: e9 01 00 00 00      jmp   8048086 <label+0x1>
08048085 <label>:
 8048085: 90                  nop
 8048086: b8 01 f0 00 00      mov   eax,0xf001
```

Erschweren des Disassemblierens von Code (2)

HOCHSCHULE COBURG

Codebeispiel: Was erzeugt Disassembler bei Opcode != 0x90?

```
start:
        jmp label+1
label:  .db 0xe9
        mov eax, 0xf001
```

```
08048080 <start>:
 8048080: e9 01 00 00 00     jmp  8048086 <label+0x1>
08048085 <label>:
 8048085: e9 b8 01 f0 00     jmp  8f484242 <__bss_start+0xeff1b6>
```

Byte **0xe9** ist Start einer gültigen Instruktion, die > 1 Byte lang ist: “jmp”

Arten von Disassemblern (1)

Linear sweep
(z.B. objdump)

- Beginnt beim 1. Byte des Textsegments
- Liest linear weitere folgende Instruktionen
- Probleme, wenn Textsegment eingebettete Datenbytes enthält

```
global startAddr, endAddr;
proc DisasmLinear(addr)
begin
  while(startAddr <= addr < endAddr) do
    I:= decode instruction at address addr;
    addr += length(I);
  done
end

proc main()
begin
  startAddr := address of the first executable byte;
  endAddr := startAddr + text section size;
  DisasmLinear(ep);
end
```

Arten von Disassemblern (2)

Recursive traversal (z.B. IDA Pro)

- Berücksichtigt Kontrollfluss
- Ermittelt bei jedem Sprung mögliche Zieladressen
- Setzt von dort Disassem-blieren fort

```
proc main()
begin
  startAddr := program entry point;
  endAddr := startAddr + text section size;
  DisasmRec(startAddr);
end
```

```
global startAddr, endAddr;
proc DisasmRec(addr)
begin
  while(startAddr <= addr < endAddr) do
    if (addr has been visited already) return;
    I := decode instruction at address addr;
    mark addr as visited;
    if (I is a branch or function call)
      for each possible target t of I do
        DisasmRec(t);

      done
      return;
    else addr += length(I);
  done
end
```

Basisblöcke

Code in einem Basisblock hat…

- Einen Einsprungpunkt:
  Ziel einer Sprunganweisung
- Einen Aussprungpunkt:
  nur letzte Instruktion darf dazu führen, dass Code eines anderen Blocks ausgeführt wird
- Wenn 1. Instruktion eines Basisblocks ausgeführt wird, werden die restlichen Instr. genau einmal in der angegebenen Reihenfolge ausgeführt

```
w = 0;
x = x + y;
y = 0;
if( x > z)
  {
   y = x;
   x++;
  }
else
  {
   y = z;
   z++;
  }
w = x + z;
```

Source Code

```
w = 0;
x = x + y;
y = 0;
if( x > z)
```

```
y = x;
x++;
```

```
y = z;
z++;
```

```
w = x + z;
```

Basic Blocks

Evolution von Virusobfuskation

Entwicklung der Obfuskationstechniken

- Ver- und Entschlüsselung, Virus in zwei Teilen:
  - Verschlüsselter Code (eigentlicher Virus)
  - Entschlüsselungsfunktion in „Klartext“
- Oligomorphe Viren
  - Veränderung der Entschlüsselung nach wenigen festen Mustern
- Polymorphe Viren
  - Einsatz von Mutationsverfahren zur Erzeugung neuer Dekryptoren für neue Generationen von Viren
- Metamorphe Viren
  - Verändern auch den eigentlichen Code des Virus, nicht nur die Entschlüsselungsfunktion

Selbstentschlüsselung von Code (1)

HOCHSCHULE COBURG

Virus besteht aus zwei Teilen:

- Verschlüsselter „Nutzcode“
  - Eigentliche Funktionalität des Virus
  - Durch unterschiedliche Schlüssel ist binäre Darstellung des verschlüsselten Teils leicht zu ändern
  - **Ziel:** Funktion des Virus verschleiern *und* Entdeckung durch Signaturen erschweren
- Unverschlüsselte Entschlüsselungsfunktion („Dekryptor“)
  - Kurze Funktion, die verschlüsselten Teil in gültige Opcodes entschlüsselt

Selbstentschlüsselung von Code (2)

HOCHSCHULE COBURG

Grundlegende Verschlüsselungsmethode: XOR

- Zur Ver- und Entschlüsselung verwendet
- Die XOR-Operation ist reversibel:
  - 0xf247 XOR 0x0682 = 0x0f4c5
  - 0xf4c5 XOR 0x0682 = 0x0f247
- XOR ist ein schnelles und reversibles Ver-/Entschlüsselungs-verfahren => hier gut geeignet
  - Code zur Entschlüsselung wird mit Virus mitgeliefert, daher macht aufwendigere Verschlüsselung wenig Sinn...

Selbstentschlüsselung von Code (3)

HOCHSCHULE COBURG

Vereinfachte Version mit 4 Bytes „Nutzlast“:

```
    push %eax
    mov  %esp, %eax
    lea  Virus,%esi
    mov  $4, %esp
Decrypt:
    xor  %esp,(%esi)
    xor  %esi,(%esi)

    mov  %eax, %esp
    pop  %eax

Virus:
    0x1e 0x1f 0xc1 0xcb
```

Disassemblieren der Opcodes an Adresse „Virus“ ergibt:

```
0x1e        push  %ds
0x1f        pop   %ds
0x1c 0xcb   sbb   $0xcb,%al
```

- Code ergibt wenig Sinn...
  - und verwirrt den Virusforscher!

Selbstentschlüsselung von Code (4)

HOCHSCHULE COBURG

- **Entschlüsselungsalgorithmus verwendet XOR**

```
Berechne:
Entschlüsselter Code = Verschlüsselter Code XOR Adresse XOR Länge_des_Codes
```

- Hier:
  - Verschlüsselter Code (little endian): 0xcbc11f1e
  - Adresse = 0x08084044, Länge = 4 (Bytes):

0xcbc11f1e XOR 0x08084044 XOR 0x00000004 => **0xc3c95f5e**

```
Virus:
    0x1e 0x1f 0xc1 0xcb
```

```
0x5e   pop %esi
0x5f   pop %edi
0xc9   leave
0xc3   ret
```

Selbstentschlüsselung von Code (5)

HOCHSCHULE COBURG

Ziel: Verschlüsselung der Opcodes: 0x5e 0x5f 0xc9 0xc3

- Little endian: Wort = **0xc3c95f5e**

- Annahme: Code liegt an Adresse 0x08084044
- **Verschlüsselungsalgorithmus: invers zur Entschlüsselung**

```
Berechne:
Verschlüsselter Code = Code XOR Adresse XOR Länge_des_Codes
```

- Hier: Adresse = 0x08084044, Länge = 4 (Bytes):
  0xc3c95f5e XOR 0x08084044 XOR 0x00000004 => **0xcbc11f1e**

```
0x5e    pop %esi
0x5f    pop %edi
0xc9    leave
0xc3    ret
```

```
Virus:
    0x1e 0x1f 0xc1 0xcb
```

Selbstentschlüsselung von Code (6)

HOCHSCHULE COBURG

Realer Code aus „Cascade“ DOS-Virus: #Bytes > 4 => Schleife

```
    push %eax          ; save current EAX
    mov  %esp, %eax    ; save ESP into EAX
    lea  Virus,%esi    ; start of encrypted code
                       ; (computed by virus)
    mov  $0x684, %esp  ; length of encrypted code (4 bytes)
Decrypt:
    xor  %esp,(%esi)   ; xor code with its address
    xor  %esi,(%esi)   ; xor code with its inverse index
    add  $4, %esi      ; increase esi to read next 4 bytes
    sub  $4, %esp      ; decrease esp = code length by 4 bytes
    jnz  Decrypt       ; until all bytes decrypted

    mov  %eax, %esp    ; restore ESP
    pop  %eax          ; restore EAX

Virus:
 1e 1f c1 cb .. .. .. ; encrypted virus code body @ 0x08084044
```

Problem: Bestimmen der aktuellen PC-Adresse

HOCHSCHULE COBURG

Code wird evtl. an dem Angreifer nicht bekannte Adresse geladen

- Wie erhält man Adresse der aktuell ausgeführten Instruktion?

  `mov %eip, %eax ?!?`

  error: invalid register name

- Der PC kann bei x86 (32 bit) nicht direkt ausgelesen werden!
  - Bei anderen Architekturen ist PC normales Register
  - Bei x86-64 kann PC direkt gelesen werden: **`lea rax, [rip]`**
- Trick bei x86 32 bit: mal wieder der Stack...

```
0: e8 00 00 00 00   call   0x5
5: 58               pop    %eax
```

Unterprogrammaufruf mit *relativem Offset!*

call hat eip auf Stack geschrieben: in eax holen!

Oligomorphe Viren

Vorgehensweise bei Weiterverbreitung des Virus:

- Im Gegensatz zu einfachen verschlüsselten Viren ändern oligomorphe Viren ihren Dekryptor in neuen Generationen
- Beispiel: Der Virus Win95/Memorial konnte 96 unterschiedliche Dekryptor-Muster erzeugen
  - Erkennung des Virus basierend auf Dekryptorcode damit nicht (einfach) möglich
- Abhilfe: dynamische Entschlüsselung des Virus-“Nutzcodes“ mit Hilfe des mitgelieferten Dekryptors
  - Erkennung basiert damit auf unverändertem Code des entschlüsselten Virus

Polymorphe und metamorphe Viren

Polymorphe Viren – bei Weiterverbreitung:

- Polymorphe Viren erzeugen eine unendlich große Anzahl möglicher neuer Dekryptoren, die unterschiedliche Verschlüsselungsmethoden für den Viruscode verwenden

Metamorphe Viren

- Verändern auch den eigentlichen Code des Virus für die nächste Generation der Weiterverbreitung , nicht nur die Entschlüsselungsfunktion

Gemeinsamkeiten

- Die folgenden Obfuskationstechniken können in allen drei Varianten zum Einsatz kommen

Obfuskation durch Instruktions-ersetzung (1)

HOCHSCHULE COBURG

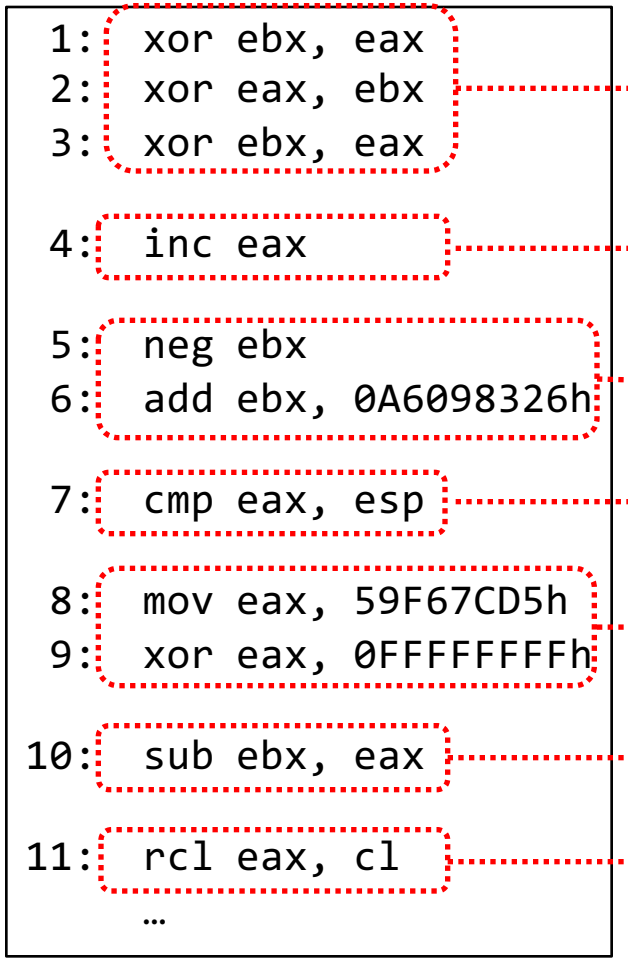
Ziel: Erschweren des reverse engineering

```
add eax, 5
```

```
xor ebx, eax
xor eax, ebx
xor ebx, eax
inc eax
neg ebx
add ebx, 0A6098326h
cmp eax, esp
mov eax, 59F67CD5h
xor eax, 0FFFFFFFFh
sub ebx, eax
rcl eax, cl
push 0F9CBE47Ah
add dword ptr [esp], 6341B86h
sbb eax, ebp
sub dword [esp], ebx
pushf
pushad
pop eax
add esp, 20h
test ebx, eax
pop eax
```

Obfuskation durch Instruktions-ersetzung (2)

HOCHSCHULE COBURG



Register eax und ebx vertauschen:
ebx = eax_1

Bedeutet hier "Wert von eax in Zeile 1"

Überflüssiger "junk"-Code, eax in Z. 8 überschrieben

ebx wird negiert und zu 0x0a6098326 addiert:
ebx = $-eax_1$ + 0x0a6098326

eax mit esp verglichen, ändert nur Flags => junk

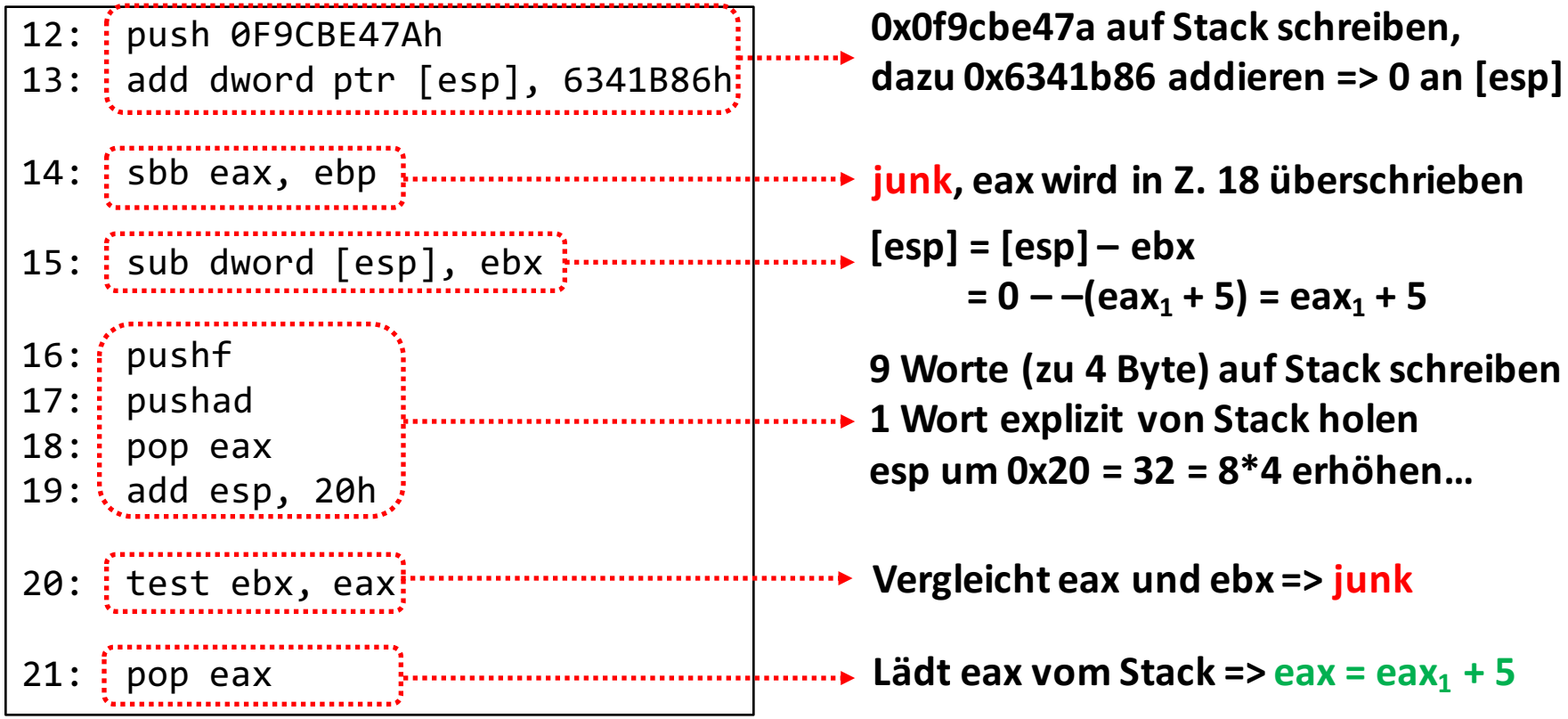
eax mit 0x59f67cd5 laden und invertieren (xor)
=> eax = 0x0a609832a

ebx = ebx – eax = $-eax_1$ + 0x0a6098326 - 0x0a609832a
ebx = $-eax_1$ – 5 = –(eax_1 + 5)

junk, eax wird in Z. 18 überschrieben

Obfuskation durch Instruktions-ersetzung (3)

HOCHSCHULE COBURG



Obfuskation durch Instruktions-ersetzung (4)

HOCHSCHULE COBURG

Verwendete Techniken zur Obfuskation:

- Pattern-basierte Obfuskation
- Constant unfolding
- Einfügen von „junk code“
- Stack-basierte Obfuskation
- Verwendung von unüblichen Instruktionen
  - z.B. RCL, SBB, PUSHF oder PUSHAD

Obfuskation durch Instruktionsersetzung: Techniken (1)

HOCHSCHULE COBURG

Verwendete Technik: *Constant* ***unfolding***

- *Constant folding* ist grundlegende Compileroptimierung
  - Ziel: Berechnungen mit zur Übersetzungszeit bekanntem Ergebnis durch Ergebnis der Berechnung ersetzen
- C-Statement x = 4 * 5; => Ausdruck „4 * 5“ besteht aus Operator (*) und zwei dem Compiler bekannten Konstanten (4 und 5)
  - Compiler kann Zuweisung ersetzen durch x = 20;
- *Constant* ***unfolding****:* Obfuskation durch dazu inverse Operation
  - Ersetze Konstante durch Berechnung, die Konstante ergibt:

```
01: push 0F9CBE47Ah
02: add dword ptr [esp], 6341B86h
```

} push 0

Obfuskation durch Instruktionsersetzung: Techniken (2)

HOCHSCHULE COBURG

Verwendete Technik: *Dead Code Insertion*

- Die u.a. Funktion gibt die Zahl 3 zurück
- Vorher werden „nutzlose“ Berechnungen ausgeführt, die die Semantik der Funktion nicht ändern
- Erste Zuweisungen an x und y sind „tot“ (dead code), da sie keine Auswirkung auf folgende Berechnungen haben

```
int f() {
    int x, y;
    x = 1;    // this assignment to x is dead
    y = 2;    // y is not used again, so it is dead
    x = 3;    // x above here is not live
    return x; // x is live
}
```

Obfuskation durch Instruktionsersetzung: Techniken (3)

HOCHSCHULE COBURG

Verwendete Technik: *Arithmetic Substitution via Identities*

- Ausnutzen mathematischer Identitäten, z.B.:
  - -x = ~x + 1 (nach Definition des Zweierkomplements)
  - rotate left(x,y) = (x << y) | (x >> (bits(x)-y))
  - rotate right(x,y) = (x >> y) | (x << (bits(x)-y))
  - x-1 = ~-x
  - x+1 = - x

Viel mehr "Spaß" in der Art gibt's im Hackers' Delight: http://www.hackersdelight.org

Obfuskation durch Instruktionsersetzung: Techniken (3)

Verwendete Technik: *Pattern-Based Obfuscation*

- Manuelle Erstellung von Transformationen, die eine oder mehrere (aufeinanderfolgende) Instruktionen in eine kompliziertere Folge von Instruktionen mit gleicher Semantik umschreiben

Beispiele:

```
01: push reg32
```

```
01: push imm32
02: mov dword ptr [esp], reg32
```

```
01: lea esp, [esp-4]
02: mov dword ptr [esp], reg32
```

```
01: sub esp, 4
02: mov dword ptr [esp], reg32
```

Obfuskation durch Instruktionsersetzung: Techniken (4)

HOCHSCHULE COBURG

Verwendete Technik: *Pattern-Based Obfuscation*

- Ersetzungen können beliebig komplex sein

Beispiel:

```
01: sub esp, 4
```

```
01: push reg32
02: mov reg32, esp
03: xchg [esp], reg32
04: pop esp
```

- **Mehrfaches Hintereinanderausführen von Ersetzungen möglich**

```
01: push ecx
```

```
01: sub esp, 4
02: mov dword ptr [esp], ecx
```

```
01: push ebx
02: mov ebx, esp
03: xchg [esp], ebx
04: pop esp
05: mov dword ptr [esp], ecx
```

Obige Ersetzung angewendet

Selbstmodifizierender Code (1)

Einfache Beispielarchitektur (angelehnt an x86):

- 1-Byte Opcodes
- 1-Byte-Adressen und *immediate*-Werte
- Befehle und Codierung:

Assembly	Binary	Semantics
`movb` *value to*	`0xc6` *value to*	set byte at address *to* to value *value*
`inc` *reg*	`0x40` *reg*	increment register *reg*
`dec` *reg*	`0x48` *reg*	decrement register *reg*
`push` *reg*	`0xff` *reg*	push register *reg* on the stack
`jmp` *to*	`0x0c` *to*	jump to absolute address *to*

B. Anckaert, M. Madou, K. De Bosschere, *"A Model for Self-Modifying Code"*, Springer Lecture Notes in Computer Science, vol 4437

Selbstmodifizierender Code (2)

Assembly	Binary	Semantics
`movb` *value to*	`0xc6` *value to*	set byte at address *to* to value *value*
`inc` *reg*	`0x40` *reg*	increment register *reg*
`dec` *reg*	`0x48` *reg*	decrement register *reg*
`push` *reg*	`0xff` *reg*	push register *reg* on the stack
`jmp` *to*	`0x0c` *to*	jump to absolute address *to*

- Beispielcode:

Address	Assembly	Binary
0x0	movb 0xc 0x8	c6 0c 08
0x3	inc %ebx	40 01
0x5	movb 0xc 0x5	c6 0c 05
0x8	inc %edx	40 03
0xa	push %ecx	ff 02
0xc	dec %ebx	48 01

1

Naive Kontrollflussanalyse ergibt:

A) movb 0xc 0x8
B) inc %ebx
C) movb 0xc 0x5
D) inc %edx
E) push %ecx
F) dec %ebx

Selbstmodifizierender Code (3)

Address	Assembly	Binary
0x0	movb 0xc 0x8	c6 0c 08
0x3	inc %ebx	40 01
0x5	movb 0xc 0x5	c6 0c 05
0x8	inc %edx	40 03
0xa	push %ecx	ff 02
0xc	dec %ebx	48 01

Assembly	Binary	Semantics
`movb` *value to*	`0xc6` *value to*	set byte at address *to* to value *value*
`inc` *reg*	`0x40` *reg*	increment register *reg*
`dec` *reg*	`0x48` *reg*	decrement register *reg*
`push` *reg*	`0xff` *reg*	push register *reg* on the stack
`jmp` *to*	`0x0c` *to*	jump to absolute address *to*

= 0xc nach Ausführen von A)

Opcode: inc => jmp, Parameter bleibt gleich

```
1  A) movb 0xc  0x8
   B) inc  %ebx
   C) movb 0xc  0x5
   D) inc  %edx
   E) push %ecx
   F) dec  %ebx
```

A) verändert Speicher an Adresse 8!

```
   A) movb 0xc  0x8

2  B) inc  %ebx
3  C) movb 0xc  0x5
   G) jmp  0x3

   E) push %ecx
   F) dec  %ebx
```

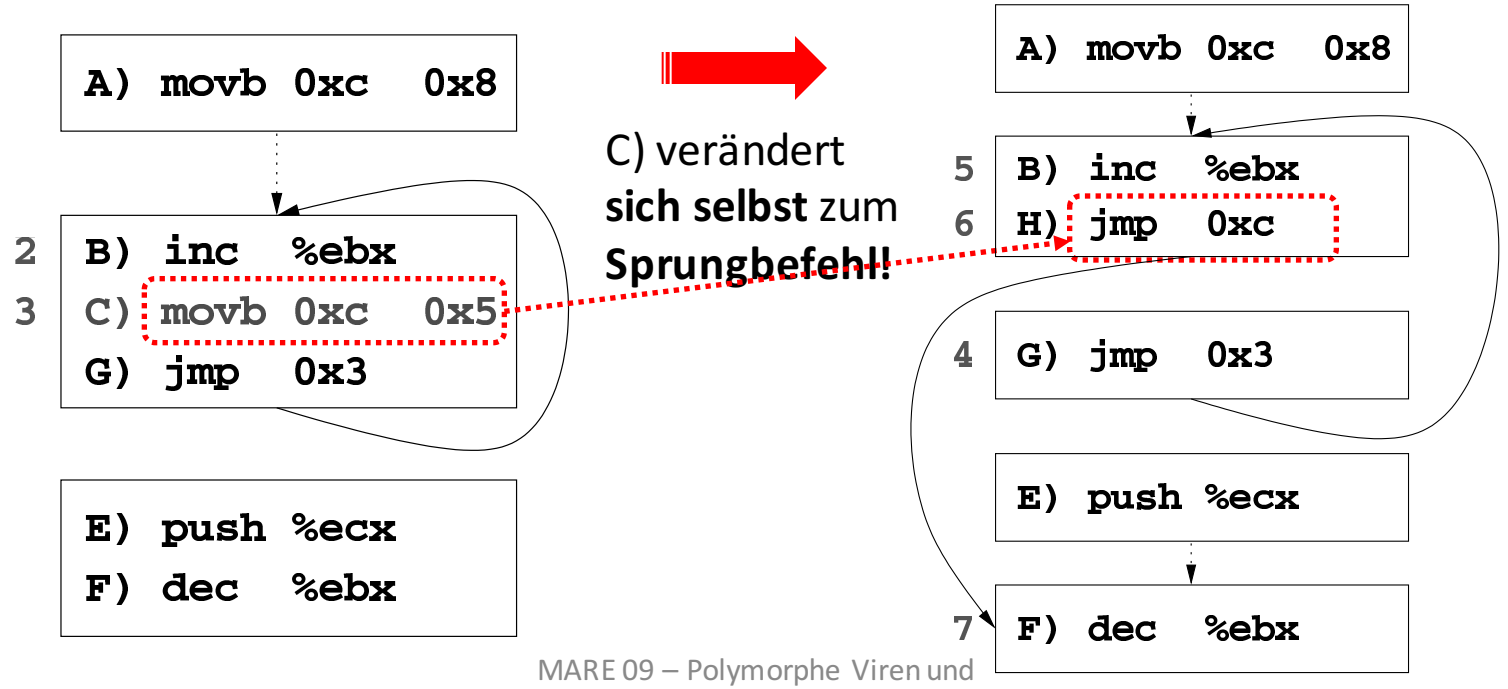
Selbstmodifizierender Code (4)

Address	Assembly	Binary
0x0	movb 0xc 0x8	c6 0c 08
0x3	inc %ebx	40 01
0x5	movb 0xc 0x5	c6 0c 05
0x8	inc %edx	40 03
0xa	push %ecx	ff 02
0xc	dec %ebx	48 01

Assembly	Binary	Semantics
`movb` *value to*	`0xc6` *value to*	set byte at address *to* to value *value*
`inc` *reg*	`0x40` *reg*	increment register *reg*
`dec` *reg*	`0x48` *reg*	decrement register *reg*
`push` *reg*	`0xff` *reg*	push register *reg* on the stack
`jmp` *to*	`0x0c` *to*	jump to absolute address *to*

= 0xc *nach* Ausführen von C)

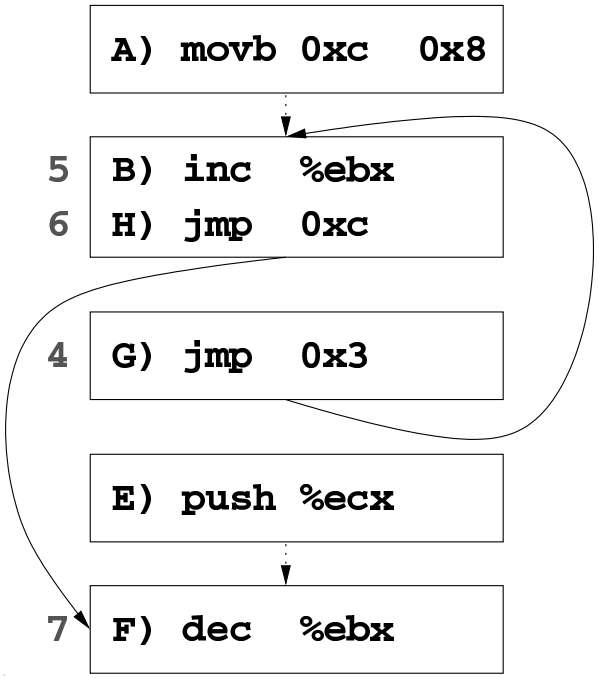
movb => jmp, Parameter jetzt nur 0x0c!



Selbstmodifizierender Code (5)

- Welche Semantik hat der Code?

Assembly	Binary	Semantics
`movb` *value to*	`0xc6` *value to*	set byte at address *to* to value *value*
`inc` *reg*	`0x40` *reg*	increment register *reg*
`dec` *reg*	`0x48` *reg*	decrement register *reg*
`push` *reg*	`0xff` *reg*	push register *reg* on the stack
`jmp` *to*	`0x0c` *to*	jump to absolute address *to*



B) wird **zweimal** ausgeführt: ebx = ebx + 2

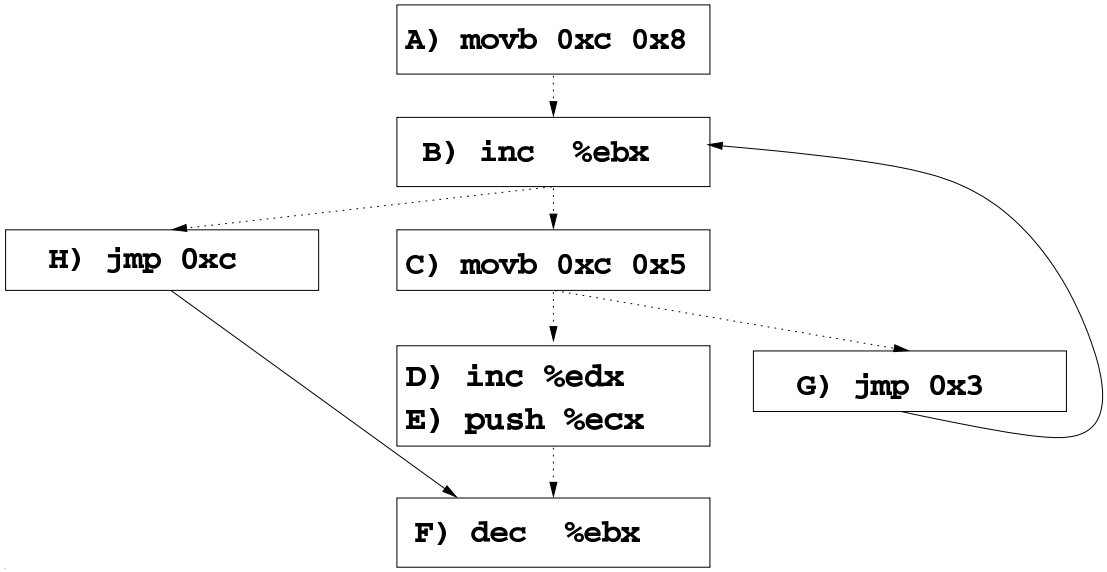
F) wird **einmal** ausgeführt: ebx = ebx - 1

Effektiv ausgeführt: inc ebx

Erweiterter Kontrollflussgraph für selbstmodifizierenden Code

Idee: alle möglichen Codeflüsse inkl. veränderten Instruktionen erfassen

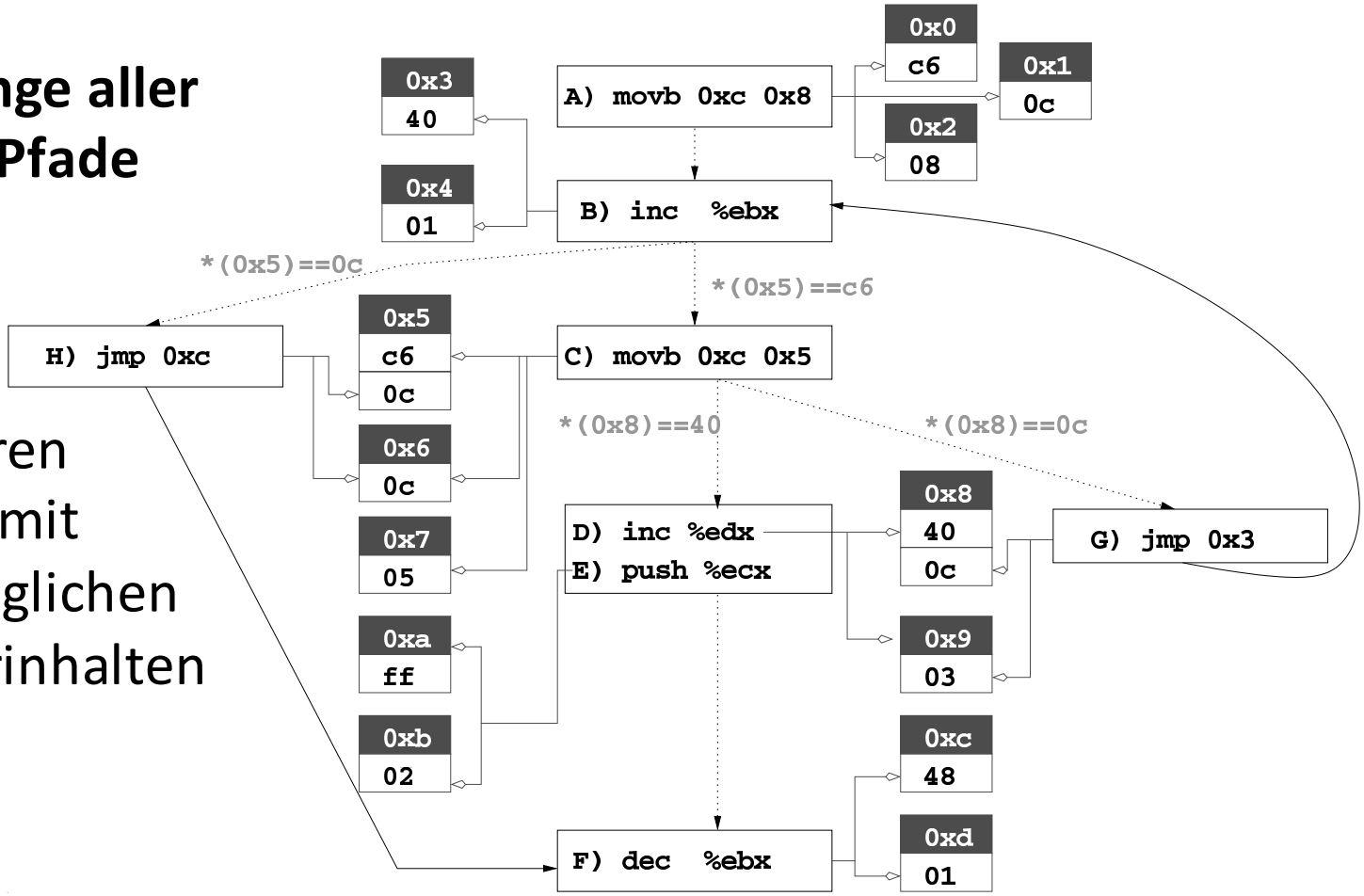
- Überlagerung der CFG aller drei Codevarianten
  - Enthält aber zusätzliche, nie ausgeführte Pfade



Erfassen der Modifikationen des CFG

Exakte Menge aller möglichen Pfade benötigt

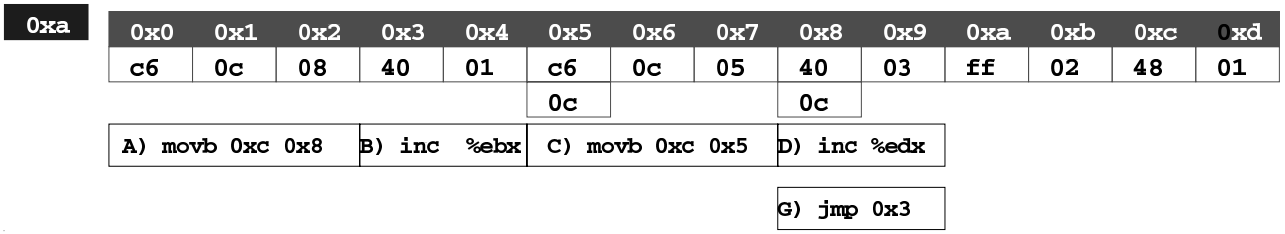
- Annotieren des CFG mit allen möglichen Speicherinhalten

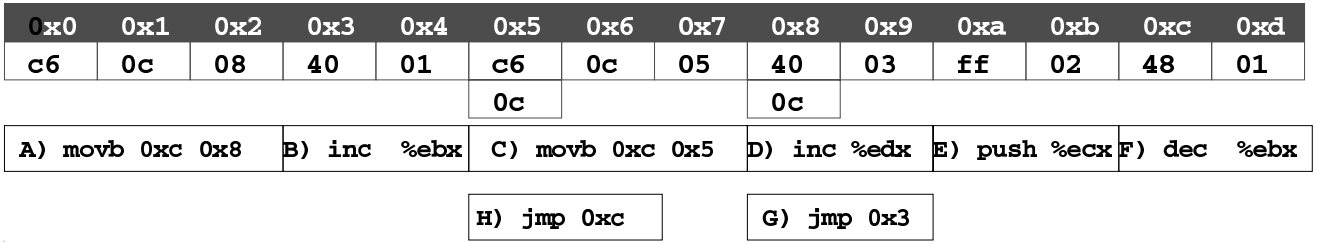


Linearisierung des CFG

Ermitteln von möglichen Ausführungs-reihenfolgen

- Analyse der möglichen Werte, die eine Speicher-zelle haben kann





Fazit

The struggle continues ☺

- Immer bessere Obfuskationsmechanismen...
- und immer bessere Erkennungsmaßnahmen

Beweis, dass perfekte Obfuskation unmöglich ist:

B. Barak et al., *„On the (Im)possibility of Obfuscating Programs“*, Journal of the ACM, Vol. 59, No. 2, Article 6, April 2012