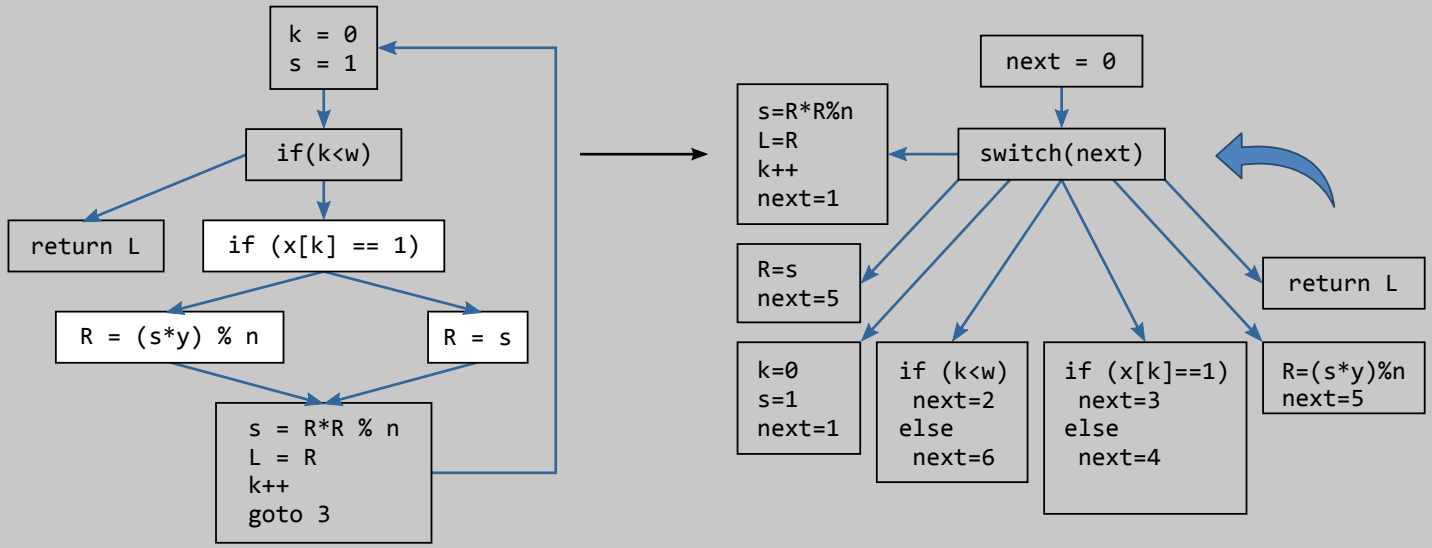
FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

Open C3S
Open Competence Center for Cyber Security



Zertifikatsprogramm

Optimierung und Obfuskierung [MM-107]

Autoren:

Dr. rer. nat. Werner Massonne

Prof. Dr.-Ing. Felix C. Freiling

Friedrich-Alexander-Universität Erlangen-Nürnberg

MM-107
Optimierung und Obfuskierung

Autoren:
Dr. rer. nat. Werner Massonne
Prof. Dr.-Ing. Felix C. Freiling

---

1. Auflage

Friedrich-Alexander-Universität Erlangen-Nürnberg

GEFÖRDERT VOM
Bundesministerium
für Bildung
und Forschung

© 2016 Felix Freiling
Friedrich-Alexander-Universität Erlangen-Nürnberg
Department Informatik
Martensstr. 3
91058 Erlangen

1. Auflage (13. Dezember 2016)

Das Werk einschließlich seiner Teile ist urheberrechtlich geschützt. Jede Verwendung außerhalb der engen Grenzen des Urheberrechtsgesetzes ist ohne Zustimmung der Verfasser unzulässig und strafbar. Das gilt insbesondere für Vervielfältigungen, Übersetzungen, Mikroverfilmungen und die Einspeicherung und Verarbeitung in elektronischen Systemen.

Um die Lesbarkeit zu vereinfachen, wird auf die zusätzliche Formulierung der weiblichen Form bei Personenbezeichnungen verzichtet. Wir weisen deshalb darauf hin, dass die Verwendung der männlichen Form explizit als geschlechtsunabhängig verstanden werden soll.

Das diesem Bericht zugrundeliegende Vorhaben wurde mit Mitteln des Bundesministeriums für Bildung, und Forschung unter dem Förderkennzeichen 16OH12022 gefördert. Die Verantwortung für den Inhalt dieser Veröffentlichung liegt beim Autor.

Inhaltsverzeichnis

Einleitung **4**
I. Abkürzungen der Randsymbole und Farbkodierungen 4
II. Zu den Autoren 5
III. Lehrziele 6

Optimierung und Obfuskierung **7**
1 Lernergebnisse 7
2 Optimierung 7
2.1 Propagierung von Konstanten 7
2.2 Dead Code Elimination 8
2.3 Inlining 9
2.4 Iterative Optimierung 10
2.5 Schleifenoptimierung 11
Unswitching 11
Loop Unrolling 12
Loop Inversion 12
2.6 Kontrollflussoptimierung 13
Branch-to-Branch Elimination 14
Sub-Expression Elimination 15
Branchless Code 15
2.7 Idiome 16
2.8 Frame Pointer Omission 18
2.9 Hot and Cold Parts 18
3 Obfuskierung 19
3.1 String Obfuscation 20
3.2 Junk Code Insertion 21
3.3 Code Permutation 21
3.4 Opaque Predicates 22
Bogus Control Flow / Dead Code 22
Fake Loops 23
Random Predicates 23
3.5 Merging 24
3.6 Splitting 24
3.7 Zahlen „verschleiern“ 24
3.8 Aliasing 26
3.9 Pointer Indirection 26
3.10 Control Flow Flattening (Chenxify) 27
4 Zusammenfassung 28

Stichwörter **29**

Liste der Lösungen zu den Kontrollaufgaben **31**

Verzeichnisse **33**
I. Abbildungen 33
II. Exkurse 33
III. Tabellen 33
IV. Literatur 33

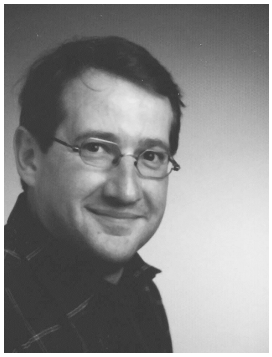
Einleitung

I. Abkürzungen der Randsymbole und Farbkodierungen

Exkurs	E
Quelltext	Q

II. Zu den Autoren

Felix Freiling ist seit Dezember 2010 Inhaber des Lehrstuhls für IT-Sicherheitsinfrastrukturen an der Friedrich-Alexander-Universität Erlangen-Nürnberg. Zuvor war er bereits als Professor für Informatik an der RWTH Aachen (2003-2005) und der Universität Mannheim (2005-2010) tätig. Schwerpunkte seiner Arbeitsgruppe in Forschung und Lehre sind offensive Methoden der IT-Sicherheit, technische Aspekte der Cyberkriminalität sowie digitale Forensik. In den Verfahren zur Online-Durchsuchung und zur Vorratsdatenspeicherung vor dem Bundesverfassungsgericht diente Felix Freiling als sachverständige Auskunftsperson.



Werner Massonne erwarb sein Diplom in Informatik an der Universität des Saarlandes in Saarbrücken. Er promovierte anschließend im Bereich Rechnerarchitektur mit dem Thema „Leistung und Güte von Datenflussrechnern“. Nach einem längeren Abstecher in die freie Wirtschaft arbeitet er inzwischen als Postdoktorand bei Professor Freiling an der Friedrich-Alexander-Universität.

III. Lehrziele

Ein regelmäßiger Untersuchungsgegenstand in der digitalen Forensik ist unbekannte Software, deren Funktionsweise analysiert werden soll. Dabei hat man es standardmäßig mit Software zu tun, deren Quellcode nicht verfügbar ist. Damit wird eine Binärcodeanalyse erforderlich, also die Analyse von Maschinenprogrammen.

Maschinenprogramme werden heute in aller Regel nicht mehr direkt in Assembler programmiert, sondern entstehen durch Kompilierung von Programmen, die in einer höheren Programmiersprache wie bspw. C geschrieben sind. Bei der Kompilierung werden in der Regel Optimierungsverfahren abgewandt mit dem Ziel eine höhere Effizienz des erzeugten Maschinencodes zu erreichen. Diese Optimierungen erschweren allerdings die Binärcodeanalyse in einem nicht unerheblichen Maße. Um solche optimierten Maschinenprogramme analysieren zu können, muss man mit den gängigen Optimierungsverfahren der Compiler vertraut sein. Damit beschäftigt sich der erste Teil dieses Mikromoduls.

Wenn nicht mehr nur die Optimierung, sondern die Erzeugung von möglichst kryptischem Maschinencode im Vordergrund steht, spricht man von Obfuskierung. Dem Analysten soll durch Obfuskierung seine Arbeit so schwer wie möglich gemacht werden. Der Übergang von der Optimierung hin zur Obfuskierung ist dabei fließend, d.h. für den Analysten ist es oftmals nicht erkennbar, ob Programmteile durch Optimierung oder durch Obfuskierung entstanden sind. Mit allgemeinen, d.h. betriebssystemunabhängigen Obfuskierungsmethoden beschäftigt sich der zweite Teil dieses Mikromoduls.

Die Inhalte dieses Mikromoduls sind in großen Teilen von einer konkreten Architektur und von einem konkreten Betriebssystem unabhängig. Wenn für das Verständnis eines Verfahrens die Angabe eines konkreten Maschinencodes in Assembler unabdingbar ist, dann wird auf IA-32, den 32-Bit-Assemblercode der Intel x86-Prozessorfamilie zurückgegriffen. Bei der Formulierung von Hochsprachensequenzen wird die Programmiersprache C benutzt.

Optimierung und Obfuskierung

1 Lernergebnisse

Sie können begründen, warum Optimierungsverfahren bei der Erzeugung von Maschinenprogrammen angewandt werden. Einige dieser Verfahren können Sie benennen und erklären.

Sie können den Begriff Obfuskierung definieren und Gründe darlegen, warum überhaupt Obfuskierungsmaßnahmen ergriffen werden. Gleichzeitig können Sie die Begriffe Optimierung und Obfuskierung voneinander abgrenzen. Ebenso können Sie begründen, warum aus der Sicht eines Programmanalysten der Übergang von Optimierung zur Obfuskierung oftmals fließend ist.

Sie können einige allgemeine Obfuskierungsmethoden benennen und ihre Funktionsweise erklären.

2 Optimierung

Ein wichtiges Einsatzgebiet der Binärcodeanalyse ist die Untersuchung der Funktionsweise von Malware. Die Autoren von Malware haben ein großes Interesse daran, ihren Programmcode zu verschleiern, was Obfuskierung (*Obfuscation*) genannt wird. Gängige Methoden dazu sind die Verschlüsselung, oder allgemein Techniken, Programme übertrieben kompliziert zu gestalten, um wesentliche Programmteile in einer Flut irrelevanten Codes untergehen zu lassen. Mit Obfuscation-Verfahren werden wir uns in Abschnitt 3 beschäftigen.

Schwer zu verstehender Code kann aber auch ganz unabhängig von der Absicht, Malware zu verfassen, das Resultat von Optimierungen sein, die entweder der Programmierer selbst oder der Compiler durchführt, um die Effizienz eines Programms zu steigern. Ziele von Optimierungen sind die Geschwindigkeitssteigerung eines Programms und/oder die Reduzierung der Programmgröße. Optimierungen gehen fast immer zu Lasten der menschlichen Lesbarkeit eines Programms und erschweren somit die Arbeit beim Reverse Engineering. Einige typische Optimierungsmethoden werden im Folgenden vorgestellt.

2.1 Propagierung von Konstanten

Bei dieser einfachen Optimierungsmethode wird eine Variable, die einen konstanten Wert hat, durch diese Konstante ersetzt. Dies erfolgt an allen Stellen, an denen sie benutzt wird, und solange sie nicht verändert wird. Im einfachsten Fall kann das wie folgt aussehen:

Quelltext 1

```
1 // Original:
2 void main() {
3  int size = 256;
4  UpdateSize(size * 5);
5 }
6 // Optimiert:
7 void main() {
8  UpdateSize(1280);
9 }
```

Q

2.2 Dead Code Elimination

Ziel der *Dead Code Elimination* ist das Löschen von unerreichbarem Code innerhalb eines Programms. Ein Beispiel von Dead Code zeigt das folgende kleine Programmbeispiel:

Q

Quelltext 2

```
1 void main() {
2  printf("hier \n");
3  return;
4  printf("dort \n");
5 }
```

Der Compiler könnte erkennen, dass die zweite `printf`-Anweisung niemals zur Ausführung kommen kann und wird sie daher möglicherweise erst gar nicht übersetzen. Ein ähnliches Beispiel ist das folgende:

Q

Quelltext 3

```
1 void main() {
2  int a = 42;
3  int Size = 32 * 5;
4  UpdateSize(Size * 5);
5 }
```

Variable a wird niemals benutzt und braucht daher weder angelegt noch initialisiert zu werden.

E

Exkurs 1: **Dead Code Elimination**

Wie das kleine Beispiel zeigt, ist Dead Code Elimination sehr wohl partiell möglich. Im allgemeinen Fall (für beliebig komplexe Programme) sind der automatischen Dead Code Elimination jedoch Grenzen gesetzt. Die allgemeine Erkennung von Dead Code ist ein – im Sprachgebrauch der theoretischen Informatik – unlösbares (nicht entscheidbares) Problem.

Dead Code Elimination unterscheidet sich von den folgenden Optimierungsverfahren dadurch, dass Reverse Engineering dadurch nicht komplizierter wird, sondern leichter.

2.3 Inlining

Das Ziel von *Inlining* ist es, den Code einer Subroutine direkt an die aufrufende Stelle zu kopieren. Das folgende Beispiel zeigt zwei Funktionen, die zusammengefasst werden können.

Q

Quelltext 4

```
1 int Increment(int x) {
2  return x+1;
3 }
4 int main(int argc, char *argv[]) {
5  return Increment(argc);
6 }
```

Die Übersetzung des ursprünglichen Programms mit Unterprogrammaufruf kann wie in Abb. 1 gezeigt aussehen. (Der Assemblercode ist das Erzeugnis eines realen Compilerlaufs. Details sollen hier nicht weiter betrachtet werden.)

```
.text:00401000 sub_401000      proc near               ; CODE XREF: _main+7↓p
.text:00401000
.text:00401000 arg_0           = dword ptr  8
.text:00401000
.text:00401000                 push    ebp
.text:00401001                 mov     ebp, esp
.text:00401003                 mov     eax, [ebp+arg_0]
.text:00401006                 add     eax, 1
.text:00401009                 pop     ebp
.text:0040100A                 retn
.text:0040100A sub_401000      endp
.text:0040100A
.text:0040100A ; ---------------------------------------------------------------------------
.text:0040100B                 align 10h
.text:00401010
.text:00401010 ; =============== S U B R O U T I N E =======================================
.text:00401010
.text:00401010 ; Attributes: bp-based frame
.text:00401010
.text:00401010 ; int __cdecl main(int argc, const char **argv, const char **envp)
.text:00401010 _main           proc near               ; CODE XREF: ___tmainCRTStartup+10A↓p
.text:00401010
.text:00401010 argc            = dword ptr  8
.text:00401010 argv            = dword ptr  0Ch
.text:00401010 envp            = dword ptr  10h
.text:00401010
.text:00401010                 push    ebp
.text:00401011                 mov     ebp, esp
.text:00401013                 mov     eax, [ebp+argc]
.text:00401016                 push    eax
.text:00401017                 call    sub_401000
.text:0040101C                 add     esp, 4
.text:0040101F                 pop     ebp
.text:00401020                 retn
.text:00401020 _main           endp
```

Abb. 1: Übersetzung ohne Inlining

Abb. 2 zeigt das Assemblerprogramm mit Inlining. Der Vorteil ist klar: Es müssen keine Parameter über den Stack transportiert werden, und die `call/ret`-Befehle fallen weg. Damit wird die Programmausführung schneller.

```
.text:00401000 ; int __cdecl main(int argc, const char **argv, const char **envp)
.text:00401000 _main           proc near               ; CODE XREF: ___tmainCRTStartup+10A↓p
.text:00401000
.text:00401000 arg_0           = dword ptr  4
.text:00401000
.text:00401000                 mov     eax, [esp+arg_0]
.text:00401004                 inc     eax
.text:00401005                 retn
.text:00401005 _main           endp
```

Abb. 2: Übersetzung mit Inlining

Der Nachteil ist die (in diesem Beispiel natürlich nicht erkennbare) Aufblähung des Codes, wenn eine Subroutine von vielen unterschiedlichen Stellen innerhalb eines Programms aus aufgerufen wird und komplexer ist als eine einfache Inkrementierung. Für jeden Aufruf muss dann der komplette Code des Unterprogramms eingefügt werden. Aus Sicht von Reverse Engineering erhöht sich der Analyseaufwand, weil auch Analysetools nicht erkennen, dass es sich immer wieder um die ursprünglich selbe Subroutine handelt.

Inlining geschieht teilweise automatisch durch den Compiler, ist aber auch ab dem C99-Standard in C explizit durch das *inline*-Schlüsselwort erzwingbar.

2.4 Iterative Optimierung

Die bisher gezeigten Optimierungsverfahren können natürlich *iterativ* angewendet werden, wie es die folgende Optimierungssequenz an einem einfachen Beispiel zeigt. Die Funktion `strmute` wandelt alle Buchstaben eines String wahlweise in Groß- bzw. Kleinbuchstaben um. Die Programmdetails für die eigentliche Umwandlung des String sind dabei unwichtig. Die Funktion `caller` ruft `strmute` für einen String `p` auf, dessen Buchstaben in Kleinbuchstaben umgewandelt werden sollen.

Q

Quelltext 5

```
1  inline void strmute(char *str, bool bcase) {
2    if (bcase == 1) {
3       ... wandle_in_kleinbuchstaben
4    }
5    else {
6       ... wandle_in_grossbuchstaben
7    }
8  }
9
10 void caller() {
11   ...
12   strmute(p,1);
13   ...
14 }
```

Zunächst wird per Inlining die Funktion `strmute` in `caller` integriert:

Q

Quelltext 6

```
1  void caller() {
2    ...
3    char *str = p;
4    bool bcase = 1;
5    if (bcase == 1) {
6       ... wandle_in_kleinbuchstaben
7    }
8    else {
9       ... wandle_in_grossbuchstaben
10   }
11   ...
12 }
```

`bcase` ändert sich nicht und kann als Konstante propagiert werden:

Quelltext 7

Q

```
1 void caller() {
2   ...
3   char *str = p;
4   if (1 == 1) {
5     ... wandle_in_kleinbuchstaben
6   }
7   else {
8     ... wandle_in_grossbuchstaben
9   }
10  ...
11 }
```

Da die Bedingung der *if*-Anweisung immer erfüllt ist, kann der *else*-Teil eliminiert werden. Dieser ist unerreichbar und damit Dead Code. Das optimierte Endprodukt sieht damit wie folgt aus:

Quelltext 8

Q

```
1 void caller() {
2   ...
3   char *str = p;
4   ... wandle_in_kleinbuchstaben
5   ...
6 }
```

2.5 Schleifenoptimierung

Im Folgenden werden drei übliche Verfahren zur Optimierung von Schleifen vorgestellt. Üblicherweise bestehen große Teile von Programmen aus Schleifen, sodass sich hier eine Optimierung ganz besonders lohnt. Deswegen gehören die Verfahren zur Schleifenoptimierung zum Standardrepertoire von Compilern und sollten dem Programmanalysten vertraut sein.

Unswitching

Unswitching verschiebt eine Bedingung in einer Schleife aus dieser heraus. Die Schleife wird für jeden der Fälle der Bedingung kopiert. Eine Schleife der Gestalt

Quelltext 9

Q

```
1 for (i=0; i<1000; ++i) {
2   if (!a)
3     { Anweisungen_a }
4   else
5     { Anweisungen_b }
6 }
```

kann bspw. in eine solche umgewandelt werden:

Q

Quelltext 10

```
1 if (!a) {
2     for (i=0; i<1000; ++i)
3       { Anweisungen_a }
4   }
5   else {
6     for (i=0; i<1000; ++i)
7       { Anweisungen_b }
8   }
```

Wenn Variable a innerhalb der Schleife nicht verändert wird, dann wird die Bedingung innerhalb der Schleife immer gleich ausgewertet. Durch Unswitching können so 999 Vergleiche eingespart werden.

Loop Unrolling

Als *Loop Unrolling* bezeichnet man das ganze oder teilweise Aufrollen von Schleifen, wenn zur Kompilierungszeit die Anzahl der Schleifendurchläufe feststeht. Das folgende Programmfragment zeigt dies am Beispiel zweier semantisch äquivalenter Zählschleifen, wobei die erste ohne Loop Unrolling 100-mal und die zweite mit teilweisem Loop Unrolling nur noch 25-mal durchlaufen wird. Dies spart Vergleiche und bedingte Sprünge ein, was zu einem besseren Laufzeitverhalten führt. Der Codeumfang wächst andererseits um fast Faktor 4 an.

Q

Quelltext 11

```
1 ohne Loop Unrolling:
2
3 for(int i = 0; i < 100; i++) {
4   function(i);
5 }
6
7 mit Loop Unrolling:
8
9 for(int i = 0; i < 100; i+=4)
10 {
11   function(i);
12   function(i+1);
13   function(i+2);
14   function(i+3);
15 }
```

Loop Inversion

Als *Loop Inversion* bezeichnet man ein Optimierungsverfahren, dass die Struktur von Schleifen verändert. Die beiden folgenden Programmfragmente zeigen die Umwandlung einer einfachen kopfgesteuerten Schleife in eine fußgesteuerte Schleife:

Quelltext 12

```
1 while (x < y) {
2   ...
3 }
```

Q

Quelltext 13

```
1 if (x < y) {
2   do {
3         ...
4   } while (x < y);
5 }
```

Q

Der Assemblercode einer fußgesteuerten Schleife ist in der Regel effizienter als der einer kopfgesteuerten Schleife, weil Sprunganweisungen eingespart werden.

2.6 Kontrollflussoptimierung

Diese Verfahren optimieren die Kontrollflussstruktur in einem Programm. Das Ziel dieser Optimierungen ist es letztendlich immer, Programmverzweigungen einzusparen, da sich diese negativ auf das Laufzeitverhalten auswirken. Einige Beispiele sollen hier aufgezeigt werden.

Exkurs 2: **Pipelining**

E

Moderne Prozessoren verwenden bei der Programmausführung Pipelining. Darunter versteht man ein Verfahren, das im Maschinencode unmittelbar aufeinanderfolgende Befehle nicht nacheinander, sondern überlappend bearbeitet.

Nehmen wir an, dass die Abarbeitung eines Maschinenbefehls in den drei Phasen *Befehl laden / Befehl decodieren / Befehl ausführen* innerhalb einer CPU abläuft. Durch Pipelining wird im ersten Schritt der erste Befehl geladen, im zweiten Schritt wird der erste Befehl decodiert und der zweite (unmittelbar im Programmspeicher folgende) gleichzeitig geladen. Im dritten Schritt wird der erste Befehl ausgeführt, der zweite decodiert und der dritte geladen usw.

Im Fall einer Verzweigung bricht das Pipelining i. Allg. ab, da der Kontrollfluss sich ändert und ein Nachfolgebefehl von einem anderen Speicherplatz als dem folgenden geladen werden muss. Es entstehen dadurch sogenannte *bubbles* in der Befehlspipeline, also ungenutzte Verarbeitungskapazitäten in der CPU. Dadurch wiederum erhöht sich die Laufzeit eines Programms. Bei realen, modernen CPUs ist die Befehlspipeline um einiges tiefer als die genannten drei Phasen.

Branch-to-Branch Elimination

Die *Branch-to-Branch Elimination* kann bei mehrfach verschachtelten Bedingungen insbesondere innerhalb von Schleifen angewendet werden. Am besten lässt sie sich anhand eines Beispiels erklären. Die Schleife

Q

Quelltext 14

```
1  while(cond1) {
2    if (cond2) {
3      if (cond3) {
4        ...Anweisungen1
5      }
6    }
7    else {
8      ...Anweisungen2
9    }
10 }
```

ist äquivalent zu folgendem Programmfragment, das durch Verwendung der *goto*-Anweisungen schon näher an der Übersetzung in Assembler orientiert ist:

Q

Quelltext 15

```
1  while_header:
2  if (!cond1)
3    goto follow_while;
4
5  if (!cond2)
6    goto do_else;
7
8  if (!cond3)
9    goto follow_else;
10
11 ...Anweisungen1
12 goto follow_else;
13
14 do_else:
15 ...Anweisungen2
16
17 follow_else:
18
19 goto while_header;
20
21 follow_while:
```

Es ist leicht zu erkennen, dass die beiden Sprünge zu `follow_else` einen direkt nachfolgenden Sprung (Branch-to-Branch) nach `while_header` verursachen. Sie können durch einen direkten Sprung nach `while_header` wegoptimiert werden:

Quelltext 16

Q

```
1 while_header:
2 if (!cond1)
3   goto follow_while;
4
5 if (!cond2)
6   goto do_else;
7
8 if (!cond3)
9   goto while_header;
10
11 ...Anweisungen1
12  goto while_header;
13
14 do_else:
15 ...Anweisungen2
16
17 goto while_header;
18
19 follow_while:
```

Sub-Expression Elimination

Bei diesem Optimierungsverfahren werden Teilausdrücke, die mehrfach in Anweisungen benutzt werden, nicht mehrfach, sondern nur einmal ausgewertet. Hierbei muss natürlich gewährleistet sein, dass die Teilausdrücke jeweils denselben Wert haben. Dieses Verfahren kann bspw. bei arithmetischen Ausdrücken und bei Adressberechnungen angewandt werden, wie dies im folgenden Beispiel zu sehen ist:

Quelltext 17

Q

```
1 int e = b + c + d ;
2 int f = a + c + d ;
3
4 a = Struct2->Struct1->Member1;
5 b = Struct2->Struct1->Member2;
```

Die beiden ersten Programmzeilen benutzen denselben Teilausdruck `c+d`, die letzten beiden teilidentische Pointer-Strukturen.

Branchless Code

Ziel der Optimierung durch *Branchless Code* ist es, bedingte Sprünge aus den im vorangegangenen Exkurs genannten Gründen durch semantisch äquivalenten, sprungfreien Code zu ersetzten. Ein beliebtes Mittel dazu ist der `sbb`-Befehl, wie dies in der folgenden Assemblersequenz dargestellt ist:

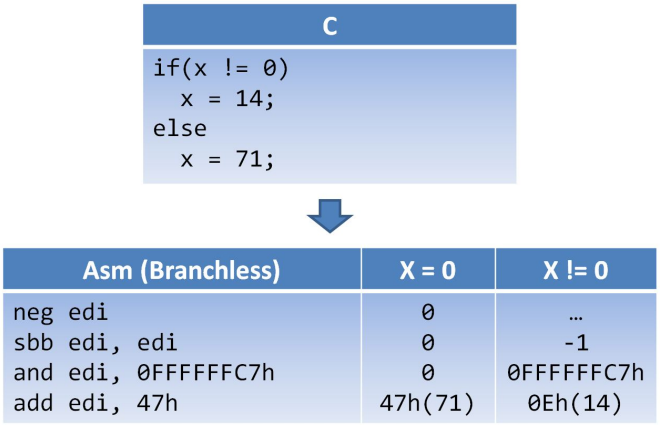
```
cmp  [ecx],2
sbb  eax, eax
inc  eax
```

Der Subtraktionsbefehl `sbb` subtrahiert die Summe aus dem 2. Operanden und dem Carryflag vom 1. Operanden. Zunächst wird der Inhalt der Speicherzelle, auf die `ecx` zeigt, mit der Konstanten `2` verglichen. In Abhängigkeit vom Ergebnis wird das Carryflag gesetzt oder nicht. Der Befehl `sbb eax,eax` liefert daher in Abhängigkeit vom Vergleichsresultat `0` oder `-1`. Dieses Resultat wird schließlich noch inkrementiert und passt damit zum Ergebnis einer C-Bedingung der Form

```
eax = ([ecx] >= 2).
```

Abb. 3 zeigt ein etwas erweitertes Beispiel. Die *if-else*-Anweisung wird hier komplett ohne Programmverzweigung in ein kleines Stück Branchless Code übersetzt.

Abb. 3: Branchless Code



K

Kontrollaufgabe 1

Das Beispiel ist ein wenig knifflig, und ein solcher Code ist wahrscheinlich nicht das Machwerk eines Compilers.

Wie funktioniert das Assemblerprogramm?

Kleine Hilfestellung: Der `neg`-Befehl negiert den Operanden, indem er den Operanden von Null subtrahiert. Ist das Ergebnis Null, so wird das Carryflag gelöscht, andernfalls wird es gesetzt.

2.7 Idiome

Idiome sind Befehle oder Befehlssequenzen, die für einen anderen als den offensichtlichen Zweck verwendet werden. Das wohl am häufigsten anzutreffende Idiom bei IA-32 ist

```
xor eax, eax,
```

das statt dieses Befehls verwendet wird:

```
mov eax, 0.
```

Die Gründe für die Verwendung des `xor`-Befehls sind der kürzere Opcode und die je nach Prozessormodell schnellere Befehlsausführung.

Compiler-Arithmetik

Andere Idiome sind in der Compiler-Arithmetik zu finden, also in der Übersetzung von arithmetischen Operationen und Ausdrücken der Hochsprache. Ziel ist es

Code zu erzeugen, den die Hardware möglichst effizient ausführen kann. Hier sollen nur ein paar Beispiele aufgezeigt werden.

Die allgemeinen Multiplikations-/Divisionsbefehle `mul/div` und `imul/idiv` für vorzeichenlose und vorzeichenbehaftete Zahlen benötigen viele CPU-Zyklen, während Additionen, Subtraktionen und Shift-Befehle von der CPU sehr viel schneller ausgeführt werden können. Das Optimierungsziel ist demnach, Multiplikationen durch Additionen und Shift-Befehle zu ersetzen, bzw. Divisionen durch Subtraktionen und Shift-Befehle, wenn ein Operand zur Kompilierungszeit bekannt ist.

Multiplikation und Division mit Zweierpotenzen sind im Binärsystem sehr schnell mittels der Shift-Befehle `shl` und `shr` zu realisieren, für die die CPU bereits eingebaute Hardware besitzt.

Ein paar Beispiele sehen Sie hier:

```
00000010b * 2^2  = 00001000b 2*4=8         shl 2
00011101b * 2^3  = 11101000b 29*8=232      shl 3
00011100b / 2^1  = 00001110b 28/2=14       shr 1
```

Zwei Beispiele für die Ersetzung von Multiplikationen durch Additionen sind hier zu sehen:

```
lea eax, [eax+eax*2]    ; eax = eax*3

mov ebx, eax
add eax, eax
add eax, eax
add eax, eax
add eax, eax
add eax, ebx            ; eax = 17*eax
```

> Kontrollaufgabe 2
>
> Beschreiben Sie die beiden Assemblersequenzen des Beispiels.

K

Weniger in den Bereich der Optimierung als in den Bereich der Notwendigkeit, dass ein Compiler eventuell Code für verschiedene Varianten der x86-Prozessorfamilie erzeugen muss, fallen folgende Beispiele von Assemblercode, die 64-Bit-Arithmetik in 32-Bit-Arithmetik umwandeln. 64-Bit-Zahlen werden in 2 32-Bit-Zahlen gesplittet, und die Berechnungen werden durch arithmetische Operationen auf dem `Low Dword` und dem `High Dword` der ursprünglichen 64-Bit-Zahl nachgebildet:

Addition:

```
mov eax,  [operand1_lo]
mov edx,  [operand1_high]
add eax,  [operand2_lo]    ; Addition ohne CF
adc edx,  [operand2_high]  ; Addition mit CF
```

Subtraktion:

```
mov eax,  [operand1_lo]
sub eax,  [operand2_lo]    ; Subtraktion ohne CF
mov edx,  [operand1_high]
sbb edx,  [operand2_high]  ; Subtraktion mit CF
```

2.8 Frame Pointer Omission

Der Frame Pointer `ebp` zeigt normalerweise auf den Anfang eines Stack Frame und wird dazu benutzt, Parameter und lokale Variablen einer Funktion zu adressieren. Erfolgt die Adressierung von Parametern und lokalen Variablen ausschließlich über den Stack Pointer `esp`, so kann `ebp` als General-Purpose-Register für andere Zwecke verwendet werden. Wegen der geringen Anzahl an General Purpose Registern bei IA-32 kann dies durchaus zu Performance-Vorteilen führen. Man spricht daher von einer Optimierung durch Wegfall des Frame Pointer (Frame Pointer Omission).

Dieses Verfahren ist allerdings nur anwendbar, wenn zur Kompilierungszeit der Offset von Parametern und lokalen Variablen relativ zum jeweils aktuellen Stack Pointer `esp` bestimmt werden kann. Die Optimierung durch Frame Pointer Omission ist daher nicht generell einsetzbar. Wird bspw. während der Ausführung einer Funktion dynamisch Speicher in zur Kompilierungszeit nicht bestimmbarer Größe auf dem Stack allokiert, so kann der Offset vom Stack Pointer zu den Parametern und lokalen Variablen nicht mehr korrekt bestimmt werden.

Frame Pointer Omission ist ebenfalls ausgeschlossen, wenn innerhalb einer Funktion Structured Exception Handling eingesetzt wird. Der Exception Handler wird in der Regel zur Fehlerregulierung auf die Parameter und Variablen der Funktion zugreifen müssen. Auf der anderen Seite ist der Aufrufzeitpunkt des Exception Handler nicht vorausbestimmbar, also auch nicht der Wert des dann aktuellen Stack Pointer. Demzufolge benötigt der Exception Handler den Frame Pointer zur Adressierung der Parameter und Variablen.

2.9 Hot and Cold Parts

Nehmen wir an, dass ein Kontrollflussgraph einer Funktion wie in Abb. 4 dargestellt aussieht. Sei weiter angenommen, dass zur Kompilierungszeit entschieden werden kann, dass die blau eingefärbten Basic Blocks diejenigen sind, die bevorzugt zur Ausführung kommen.[1] Diese Basic Blocks werden *Hot Parts* genannt, die übrigen *Cold Parts*.

Um die Geschwindigkeitsvorteile durch Caching zu nutzen, macht es Sinn, die Hot Parts von Funktionen zu sammeln und in Speicherseiten zu konzentrieren, die dann aufgrund der Caching-Strategie im Cache verbleiben. Dieses Verfahren kann durchaus funktionsübergreifend angewandt werden, also durch Ansammlung von Hot Parts unterschiedlicher Funktionen. Im Idealfall werden die Basic Blocks nach Ausführungswahrscheinlichkeit sortiert und dann so in Speicherseiten aufgeteilt, dass sich darin jeweils Basic Blocks mit ähnlicher Ausführungswahrscheinlichkeit befinden. Damit lässt sich das „Seitenflattern“[2] bei der Programmausführung minimieren und die Laufzeit optimieren. Der Kontrollfluss wird dadurch für den menschlichen Betrachter verwirrend, Analysetools wie IDA können diese Verschränkung der Basic Blocks allerdings problemlos auflösen.

[1] Es sollte klar sein, dass diese Einteilung nicht wirklich entscheidbar ist, aber durchaus mittels Heuristiken näherungsweise festgelegt werden kann.

[2] Als Seitenflattern bezeichnet man das häufige Ein- und Auslagern von Seiten im Cache.

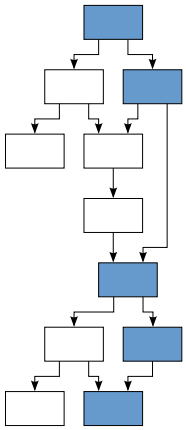


Abb. 4: Hot and Cold Parts

Es gibt viele weitere Beispiele für Codeoptimierung. Eine erweiterte Übersicht findet sich bspw. in [Rolles, 2007].

3 Obfuskierung

Die in Abschnitt 2 gezeigten Optimierungsverfahren erschweren die Dekompilierung und damit die Analyse von Programmen. Die Optimierung hat zum Ziel, die Ausführung eines Programms effizienter zu machen, d. h. in der Regel zu beschleunigen. Wenn die Absicht im Vordergrund steht, ein Programm unverständlicher zu machen, spricht man von Obfuskierung (Verschleierung) oder *Obfuscation*.

Obfuscation als Software-Schutztechniken

Obfuscation gehört zu den Software-Schutztechniken und dient der Verschleierung des ursprünglichen Quellcodes, der durch Binärcodeanalyse gewonnen werden soll. Dabei ist es wichtig, dass Obfuscation zum einen das ursprüngliche Programmverhalten erhält und zum anderen dabei effizient bleibt, d. h. die zusätzliche Komplexität der Verschleierung ergibt keinen merklichen Laufzeitverlust. Die resultierende Laufzeit sollte vergleichbar mit der des ursprünglichen Programms sein. Ziel von Obfuscation ist es vor allem, den Kontrollfluss so zu verkomplizieren, dass dessen eigentliche Funktionalität nicht mehr erkennbar ist.

Obfuscation wird keineswegs nur für illegale Zwecke eingesetzt. Durch Obfuscation kann Code vor unerlaubtem (Lese-)Zugriff geschützt werden. Beispiele für legale und praktische Anwendungen dieser Art sind:

- Legitime Software gegen Cracker schützen (Kopierschutz und Lizenzierung).
- Geistiges Eigentum vor der Konkurrenz schützen.
- DRM (*digital rights management*) gegen Kunden durchsetzen.
- Virtuelle Maschinen und Software in einer Cloud schützen.
- Schutz eines Betriebssystems gegen Viren bzw. vorhergehende Codeanalyse des Angreifers.

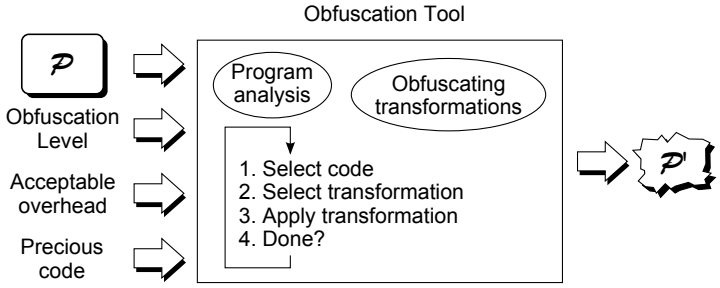
Obfuscation als Anti-Forensik-Technik

Obfuscation wird des Weiteren als Anti-Forensik-Technik eingesetzt, um bspw. nicht nachweisbare Code-Plagiate zu erstellen, Wasserzeichen zu entfernen, die Zuordnung von Malware zum Autor zu erschweren oder sich vor Antiviren-Software zu verstecken. Dabei wird angenommen, dass sich das Programm unter der vollen Kontrolle des Rechners befindet, auf dem es läuft. Der Administrator, Antiviren-

Software oder Analysten können also bspw. auf den kompletten Code des Programms zugreifen oder die aktuell verwendeten Daten verändern.

Obfuscation mit dem Ziel, die statische Analyse[3] zu behindern, ist schon mit einfachen Mitteln gut umsetzbar.[4] Solche Obfuscation findet oftmals mittels eigener Programme statt, da die Verschleierung „per Hand“ fehleranfällig und aufwendig ist. Diese Obfuskatoren ähneln Compilern, nur dass sie möglichst unleserlichen Code produzieren. Die Vorgehensweise solcher Tools ist in Abbildung 5 dargestellt. Stellen im Code, die als wichtig markiert wurden (*precious code*), werden über mehrere Iterationen analysiert und mittels vorgegebener Codetransformationen umgewandelt. Zum Schluss entsteht aus dem Programm P das obfuskierte Programm P'.

Abb. 5: Obfuskator



Schwieriger ist die Behinderung der dynamischen Analyse. Nichtsdestotrotz existieren viele Techniken, die die dynamische Analyse durch Obfuscation erschweren.

Obfuscation-Verfahren lassen sich grob in drei Gruppen aufteilen:

1. Allgemeine Methoden, die weitgehend unabhängig vom konkreten Zielsystem sind.
2. Betriebssystemabhängige Methoden (z.B. *Import Hiding*, *Hashing* und *SEH* unter Windows).
3. Verfahren zur Behinderung der Disassemblierung. In dieser Gruppe sind die raffiniertesten Methoden zu finden, die speziell bei der Entwicklung von Malware verwendet werden.

Wir beschäftigen uns im Folgenden nur mit der ersten Gruppe der Obfuscation-Verfahren. Die weiteren werden im Mikromodul MM-108 behandelt.

3.1 String Obfuscation

In den Datenbereichen eines Programms enthaltene Strings können einige Rückschlüsse auf das Verhalten eines Programms zulassen. Bei der *String Obfuscation* werden verräterische Strings bereits im Quellcode verschlüsselt und erst zur Laufzeit entschlüsselt. Bereits eine ganz einfache Verschlüsselung (z. B. `xor`) macht die Erkennung vielsagender Strings im Programm sehr schwer. Das folgende kleine Programm gibt „Hello World“ auf der Konsole aus, im *String window* von IDA ist das aber natürlich nicht erkennbar.

[3] Bei der statischen Analyse wird der Code im Gegensatz zur dynamischen Analyse nicht unter der Kontrolle eines Tools ausgeführt, sondern nur statisch betrachtet.

[4] „Gute“ Beispiele für Codeverschleierungen sind sehr einfach zu finden, da auf diesem Gebiet sogar Wettbewerbe (z. B. Obfuscated C Contest und Obfuscated Perl Contest) stattfinden.

Q

Quelltext 18

```
1 char* decrypt(char *str) {
2   int i;
3   for (i = 0; i < strlen(str); i++)
4     str[i] ^= 0xAA;
5   return str;
6 }
7
8 int main(void) {
9   char str[] = "\xE2\xCF\xC6\xC6\xC5\x8A\xFD\xC5\xD8\xC6\xCE\
      xA0";
10  printf(decrypt(str));
11  return 0;
12 }
```

3.2 Junk Code Insertion

Bei der *Junk Code Insertion* werden nutzlose Instruktionen eingefügt, welche die Semantik eines Programms nicht verändern. Dies kann sowohl auf Hochsprachenebene als auch im Assemblercode erfolgen. Im Gegensatz zu *Dead Code* wird Junk Code ausgeführt, bewirkt aber nichts. Durch Junk Code wird die Lesbarkeit und somit die Analysierbarkeit eines Programms verschlechtert. Ohne Hilfsmittel müssen letztendlich die Instruktionen einzeln untersucht werden, und es muss analysiert werden, ob die Ergebnisse von Berechnungen im späteren Programmverlauf verwendet oder verworfen werden.

Es existieren zwar Tools, die den Analysten bei der Identifizierung von Junk Code unterstützen, jedoch kann aus gewissen Gründen (Nicht-Entscheidbarkeit des Halteproblems) Junk Code nicht generell automatisch erkannt und beseitigt werden.

3.3 Code Permutation

Code Permutation unterscheidet sich prinzipiell nicht von den in Abs. 2.7 vorgestellten Idiomen, verfolgt jedoch im Gegensatz zu Idiomen primär die Absicht der Verschleierung und nicht die der Optimierung. Eine gewünschte Funktionalität kann schlichtweg auf vielfache Weise implementiert werden. Das folgende Beispiel zeigt einige Variationen eines einfachen Befehls (`mov [X],42`); der Fantasie sind hierbei keine Grenzen gesetzt.

Q

Quelltext 19

```
1 mov [X], 42
2
3 push 42
4 pop [X]
5
6 mov eax, X
7 mov [eax], 42
8
9 mov edi, X
10 mov eax, 42
11 stosd
```

```
12
13 push X
14 pop  edi
15 push 42
16 pop  eax
17 stosd
```

K

Kontrollaufgabe 3

Wissen Sie, wie der IA-32-Befehl `stosd` funktioniert?

3.4 Opaque Predicates

Die Komplexität eines Programms wächst mit der Anzahl der darin enthaltenen Verzweigungen aufgrund von Bedingungen (Prädikate). *Opaque Predicates* sind Prädikate, deren Auswertung immer denselben Wert liefert. Dieser Wert ist dem Programmierer zwar bekannt, für den Analysten aber möglicherweise schwer erkennbar, z. B. bei:

Q

Quelltext 20

```
1 if (b == true)
2   //...
3
4 if (b == 2|(x^2+x))
5   //...
```

Diese beiden Verzweigungen sind auf das Ergebnis bezogen identisch, jedoch ist das Ergebnis der zweiten Bedingung nicht unmittelbar ersichtlich. Opaque Predicates können in vielfacher Weise für eine Obfuscation verwendet werden, z. B. bei:

- Bogus Control Flow / Dead Code
- Fake Loops
- Random Predicates

Es folgen die Ausführungen zu den einzelnen Punkten.

Bogus Control Flow / Dead Code

Der *else*-Zweig nach Auswertung eines immer „wahr“ liefernden Opaque Predicate wird niemals ausgeführt und enthält daher Dead Code. Dies kann genutzt werden,

um den Quellcode künstlich zu vergrößern, ohne dass die Performance merklich eingeschränkt wird. Hier ein Beispiel:

Quelltext 21

Q

```
1 a();
2 b();
3
4 //gleichwertig:
5
6 a();
7 if(opaque_true)
8    b();
9 else
10    deadcode();
```

Fake Loops

Durch das Einfügen von zusätzlichen Schleifen, die – wie im folgenden Beispiel zu sehen – nicht durchlaufen werden, kann zusätzliche Verwirrung gestiftet werden:

Quelltext 22

Q

```
1 for(int i = 0; i < size; ++i){
2   if(opaque_true){
3     break;
4   }
5   //..do something
6 }
```

Die Performance bleibt hier annähernd gleich, da der eigentliche Coderumpf der Schleife nie ausgeführt wird. Nichtsdestotrotz fallen für die Schleife und die Variable `i` Kosten in Form von Laufzeit und Speicherplatz an.

Random Predicates

Bei *Random Predicates* wird im Gegensatz zu Opaque Predicates eine Verzweigung zufällig gewählt, jedoch danach derselbe Code, unterschiedlich verschleiert, ausgeführt. Nach der Obfuscation ist nicht mehr ohne Weiteres erkennbar, dass in beiden Fällen dasselbe berechnet wird, was den Analyseaufwand verdoppelt:

Quelltext 23

Q

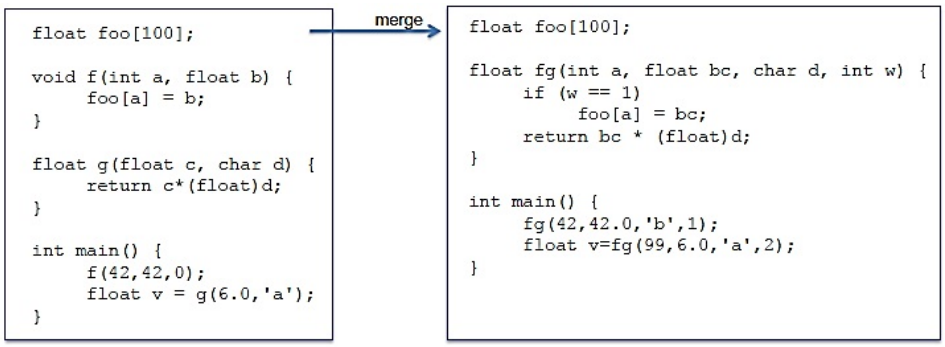
```
1 if(random){
2     OBF_1(b());
3 }
4 else{
5   OBF_2(b());
6 }
```

3.5 Merging

Zwei Funktionen, die möglichst wenig miteinander zu tun haben, werden zu einer Funktion verschmolzen. Dadurch entsteht der Eindruck, dass ihr Code zusammen gehört, weil sie über dieselbe Schnittstelle erreichbar sind.

Im Beispiel von Abb. 6 werden die Funktionen `f` und `g` verschmolzen. Die Schnittstellenfunktion `fg` enthält nun einen zusätzlichen Parameter `w`, der darüber entscheidet, welche Funktion ausgeführt werden soll.

Abb. 6: Merging von Funktionen

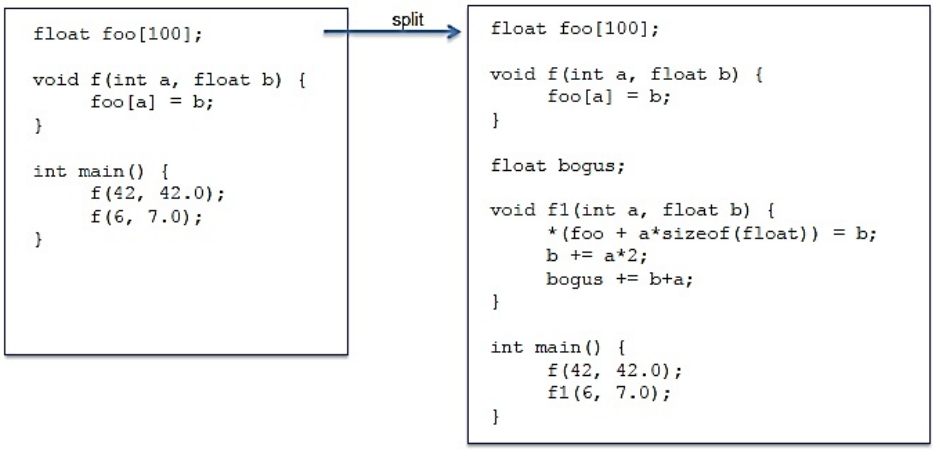


3.6 Splitting

Analog zur Verschmelzung von Funktionen ist auch das Umgekehrte denkbar, nämlich das Aufteilen von Funktionen. Im Ergebnis wird der Zusammenhang der Codeteile verschleiert.

Im Beispiel (Abb. 7) sind die Ausdrücke $*(foo + a * sizeof(float)) = b$ und $foo[a] = b$ gleichwertig, da ein Array lediglich ein zusammenhängender Bereich im Speicher ist. Es kann daher auch einfach der Pointer `foo` um `a` Einträge verschoben werden.

Abb. 7: Splitting von Funktionen



3.7 Zahlen „verschleiern“

Zahlen und Berechnungen können durch Arithmetik, Lookup Tables und Teil-Homomorphie vor dem Betrachter verborgen werden. Der Übergang von Optimierung zu Obfuscation ist hierbei fließend, insbesondere im Fall der Arithmetik. Beispiele für Compiler-Arithmetik wurden bereits an der genannten Stelle gezeigt.

Einfache Arrays, die Zahlenwerte speichern (*Lookup Tables*), können Zahlen im Code ersetzen. So könnte z. B. statt `42` ein `lt[10]` oder `lt[i]` verwendet werden,

wenn ein mit passenden Werten gefülltes Array `lt` existiert. Allerdings ist dies nur sinnvoll, wenn dazu ein kleines Array verwendet werden kann.

Tabelle 1 zeigt den Speicherbedarf einer Funktion auf Integer-Werten, wenn diese komplett in Form einer Lookup Table gespeichert werden soll. Eine Funktion `f(x,y)` mit je 32-Bit-Werten würde als Lookup Table demnach 236 GB benötigen, was eindeutig nicht sinnvoll umsetzbar ist.

Funktion	**Werte**	**Summe**	**Wertung**
f(x,y)	32 Bit	236 GB	zu viel
f(x)	23 Bit	16 GB	zu viel
f(x)	16 Bit	128 kB	okay
f(x,y)	8 Bit	64 kB	okay

Tabelle 1: Abschätzung Lookup Table

Bei Teil-Homomorphie wird die Tatsache ausgenutzt, dass bestimmte Aktionen trotz Verschlüsselung (z. B. bei Modulo oder RSA) mit diesen Zahlen noch möglich sind. Hier ein Beispiel für eine Modulo-Verschlüsselung:

Teil-Homomorphie

Q

Quelltext 24

```
1  #define N 3127
2
3  int enc(int e, int p){
4    return p*N + e;
5  }
6  int dec(int e){
7    return e % N;
8  }
9
10 int encryptedadd(int a, int b){
11   return a + b;
12 }
13 int encryptedmul(int a, int b){
14   return a * b;
15 }
```

Die Verschlüsselung einer ganzen Zahl e erfolgt durch Addition von e auf das Produkt einer beliebigen positiven ganzen Zahl p und einer Konstanten N. Zur Entschlüsselung wird der Rest einer Division durch N berechnet, also eine Modulo-Operation %.

Es gilt dann für ganze Zahlen a und b und beliebige p_1, p_2 und p_3 :

1. $dec(enc(a+b,p_1)) = dec(enc(a,p_2)+enc(b,p_3))$
2. $dec(enc(a*b,p_1)) = dec(enc(a,p_2)*enc(b,p_3))$

Nachteil dieser Verschleierung ist, dass alle vorkommenden Zahlen, also auch $a+b$ und $a*b$ kleiner als N sein müssen, denn sonst kommt es zu einem Überlauf.

Außerdem sind Vergleiche so verschlüsselter Zahlen nicht mehr möglich, da für eine Zahl durch die Variation von p mehrere Codierungen existieren können.

K

Kontrollaufgabe 4

Zeigen Sie anhand von drei Beispielen, dass diese Aussagen für Addition, Multiplikation und Vergleiche gelten!

Ein weiteres Beispiel für Teil-Homomorphie ist das Verschlüsselungsverfahren RSA. RSA[5] ist bzgl. der Multiplikation homomorph, d. h. bzgl. der Verschlüsselung E zweier Zahlen x und y gilt:

$$E(x * y) = E(x) * E(y)$$

3.8 Aliasing

Aliasing lässt sich am besten anhand eines Beispiels erklären.

Q

Quelltext 25

```
1 int alias(int *a, int *b){
2   *a = 10;
3   *b = 5;
4   if((*a - *b) == 0)
5     code();
6 }
```

Auf den ersten Blick scheint es so, als würde die Funktion `code()` niemals ausgeführt werden. Wenn man aber den folgenden Aufruf betrachtet, wird schnell klar, dass `code()` doch aufgerufen werden kann.

Q

Quelltext 26

```
1 int a = 0;
2 alias(&a, &a);
```

Die automatische Alias-Analyse ist ein NP-schweres Problem und kann daher nicht effizient gelöst werden. Es ist somit meist nicht möglich, solche scheinbar unerreichbaren Codeabschnitte automatisch zu finden.

3.9 Pointer Indirection

Bei *Pointer Indirection* wird ein Pointer auf die entsprechende Stelle (Funktion, Klasse, Variable) gesetzt und dieser statt des eigentlichen Zugriffs verwendet. Hier ein Beispiel:

[5] siehe z. B.: http://de.wikipedia.org/wiki/RSA-Kryptosystem

Q

Quelltext 27

```
1  void f() {
2    h();
3  }
4
5  void g() {
6    f();
7  }
8
9  void h() {
10   f();
11   g();
12 }
13
14 void k() {}
15
16 int main(){
17   h();
18   void (*p)() = &k;
19   p();
20 }
```

Nach dem Kontrollflussgraphen in der Abb. 8 wird die Funktion `k` nicht aufgerufen. Dies ist jedoch nicht richtig, da der Funktionszeiger `p` auf `k` zeigt und `p` aufgerufen wird.

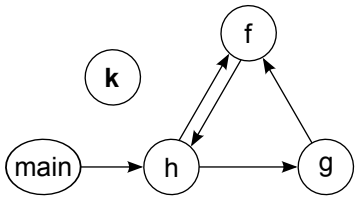


Abb. 8: Pointer Indirection

3.10 Control Flow Flattening (Chenxify)

Indem man Schleifen ausrollt und den Kontrollfluss, z.B. durch eine *switch*-Anweisung, abflacht, kann die Codekomplexität erhöht werden. Abb. 9 zeigt ein Beispiel. Im Normalfall wird der Kontrollfluss zwischen Basic Blocks durch bedingte und unbedingte Sprünge (bzw. die sequentielle Anordnung der Basic Blocks im Code) gesteuert, wie dies im linken Teil der Abbildung zu sehen ist. Einen solchen Kontrollfluss können Analysetools wie IDA auch sehr gut graphisch darstellen.

Im rechten Teil der Abbildung wird der Kontrollfluss über eine *switch*-Anweisung und eine Zustandsvariable `next` gesteuert. Der Zustand von `next` bestimmt, welcher Basic Block als nächstes ausgeführt wird. Am Ende jedes Basic Block erhält `next` einen i.d.R. neuen Wert (Folgezustand), dann wird wiederum die *switch*-Anweisung ausgeführt. Der ursprüngliche Kontrollfluss ist damit verschleiert und muss aufwändig rekonstruiert werden.

Es existieren viele weitere Möglichkeiten Code zu verschleiern, beliebt sind dabei auch Kombinationen verschiedener Methoden, die das Nachvollziehen der Programmabläufe weiter erschweren.

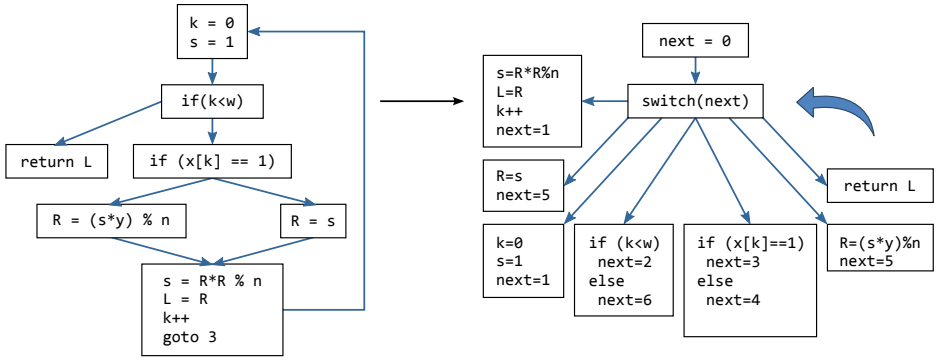


Abb. 9: Control Flow Flattening

4 Zusammenfassung

Die Analyse von Maschinencode wird durch Optimierungsverfahren des Compilers erschwert. Selbstverständlich können solche Optimierungsverfahren auch aus der Hand des Programmierers selbst stammen. Der Effekt, dass die Analyse von optimiertem Code schwieriger ist als die von nicht optimiertem Code, ist lediglich ein Nebeneffekt der Optimierung.

Sobald primär die Absicht im Vordergrund steht, einen Code schwerer verständlich zu machen, spricht man von Obfuskierung. Dabei ist der Übergang von Codeoptimierung zu Codeverschleierung fließend. Für den Analysten ist nicht immer klar erkennbar, was noch Optimierung und was bereits Obfuskierung ist.

Stichwörter

Aliasing, 26

Bogus Control Flow, 22
Branch-to-Branch Elimination, 14
Branchless Code, 15

Code Permutation, 21
Compiler-Arithmetik, 16
Control Flow Flattening, 27

Dead Code Elimination, 8

Fake Loops, 23
Frame Pointer Omission, 18

Hot and Cold Parts, 18

Idiom, 16
Inlining, 9

Junk Code Insertion, 21

Konstantenpropagierung, 7
Kontrollflussoptimierung, 13

Lookup Table, 24
Loop Inversion, 12
Loop Unrolling, 12

Malware, 7
Merging, 24

Obfuscation, 19
Opaque Predicates, 22

Pipelining, 13
Pointer Indirection, 26

Random Predicates, 23
RSA, 26

Schleifenoptimierung, 11
Splitting, 24
String Obfuscation, 20
Sub-Expression Elimination, 15

Teil-Homomorphie, 25

Unswitching, 11

Verschleierung, 19

Liste der Lösungen zu den Kontrollaufgaben

Lösung zu Kontrollaufgabe 1 auf Seite 16

Es wird hier angenommen, dass der Inhalt der Variablen `x` in Register `edi` steht. Ist $x = 0$, so liefert `neg edi` das Resultat `0`, und das Carryflag wird gelöscht; andernfalls wird das Carryflag gesetzt, und das Ergebnis ist ungleich `0`.

Der Subtraktionsbefehl `sbb edi, edi` liefert nun das Resultat `0` oder `-1` in Abhängigkeit vom Carryflag. Im Zweierkomplement wird `-1` als `FFFFFFFFh` dargestellt. `and` mit der Maske `FFFFFFC7h` liefert demnach das Ergebnis `0` oder `FFFFFFC7h`. Der Additionsbefehl `add edi, 47h` erzeugt schließlich das Ergebnis `47h` bzw. `0Eh`.

Lösung zu Kontrollaufgabe 2 auf Seite 17

Im ersten Beispiel wird `eax` mit seinem verdoppelten Wert addiert. Es werden also eine Addition und ein Links-Shift um 1 Bit ausgeführt statt einer Multiplikation mit 3.

Im zweiten Beispiel wird die Multiplikation mit 17 in eine Folge von Additionen aufgelöst. Mit dem ersten Befehl `add eax, eax` wird `eax` verdoppelt, mit dem zweiten vervierfacht usw., bis `eax` das 16-fache des ursprünglichen Wertes beinhaltet. Schließlich wird noch einmal der Anfangswert aufaddiert, der in `ebx` zwischengespeichert wurde.

Lösung zu Kontrollaufgabe 3 auf Seite 22

Das Doppelwort aus `eax` wird an die Stelle im Speicher kopiert, auf die `edi` zeigt.

Lösung zu Kontrollaufgabe 4 auf Seite 26

Seien $N = 3127$, $a = 17$, $b = 12$, $p_1 = 14$, $p_2 = 9$ und $p_3 = 10$.

1. $dec(enc(a+b, p_1)) = dec(enc(29, 14)) = dec(43807) = 29$

   $dec(enc(a, p_2) + enc(b, p_3)) = dec(enc(17, 9) + enc(12, 10)) = dec(28160 + 31282) = 29$

2. $dec(enc(a*b, p_1)) = dec(enc(204, 14)) = dec(43982) = 204$

   $dec(enc(a, p_2) * enc(b, p_3)) = dec(enc(17, 9) * enc(12, 10)) = dec(28160 * 31282) = 204$

3. $a > b$, aber $enc(a, p_2) = 28160 < enc(b, p_3) = 31282$

Verzeichnisse

I. Abbildungen

Abb. 1: Übersetzung ohne Inlining . . . 9
Abb. 2: Übersetzung mit Inlining . . . 9
Abb. 3: Branchless Code . . . 16
Abb. 4: Hot and Cold Parts . . . 19
Abb. 5: Obfuskator . . . 20
Abb. 6: Merging von Funktionen . . . 24
Abb. 7: Splitting von Funktionen . . . 24
Abb. 8: Pointer Indirection . . . 27
Abb. 9: Control Flow Flattening . . . 28

II. Exkurse

Exkurs 1: **Dead Code Elimination** . . . 8
Exkurs 2: **Pipelining** . . . 13

III. Tabellen

Tabelle 1: Abschätzung Lookup Table . . . 25

IV. Literatur

Felix Freiling, Ralf Hund, and Carsten Willems. *Software Reverse Engineering*. Vorlesung, Universität Mannheim, 2010.

Thorsten Holz. *Program Analysis*. Vorlesung, Ruhr-Universität Buchum, 2012.

Gary Nebbett. *Windows NT/2000 Native API Reference*. Macmillan Technical Publishing, 2000.

Rolf Rolles. Binary literacy – optimizations.
http://www.openrce.org/repositories/users/RolfRolles/Binary%20Literacy%20--%20Static%20--%206%20--%20Optimizations.ppt, 2007.

M. E. Russinovich and D. A. Solomon. *Windows Internals*. Microsoft Press Corp, 2012.

Sebastian Strohhäcker. Malicious code: Code obfuscation.
www.reverse-engineering.info/OBF/strohhaecker.pdf, 2004.