Technische Hochschule Ingolstadt

Seminar
Control Flow Checking

Sommersemester 2016

Seminararbeit

Control Flow Integrity
Principles, Implementations, and Applications

von

Stefan Braun

1 Über rücksprungadressbasierte Angriffe

Im Gebiet des themenverwandten *Control Flow Checking* sollen Kontrollflussfehler entdeckt werden, welche durch zufällig gekippte Bits entstehen – hervorgerufen durch Spannungsschwankungen oder Ionisierung über radioaktive Teilchen. Insbesondere Letzteres ist etwa in der Raumfahrt von eminenter Bedeutung.

Konträr dazu steht bei der hier thematisierten *Control Flow Integrity* ein potentieller Angreifer im Fokus, welcher durch gezielte Manipulation des Speichers Code in einer nicht erwünschten Reihenfolge zur Ausführung bringt. Der Umfang eines derartigen Angriffes reicht von einer Störung der Integrität des betroffenen Programms bis hin zur Ausführung beliebigen Codes auf dem zugrundeliegenden Rechner. Control Flow Integrity als Methode der statischen Codeanalyse versucht nun solche Angriffsmuster anhand von vorgeschalteten Prüfmechanismen bei Sprunganweisungen zu verhindern.

Basierend auf dem Paper *„Control-Flow Integrity Principles, Implementations, and Applications“* von Abadi et al. [1] wird in der vorliegenden Arbeit eine Implementierung von CFI diskutiert.

1.1 Vorausgesetzte Kenntnisse

Die folgenden Inhalte setzen sowohl ein Grundverständnis der Programmierung in C und Assembler[1], sowie der grundlegenden Funktionsweise der x86-Architektur voraus. Eine Einführung hierzu bietet beispielsweise [5, S.7-91].

[1] verwendet wird die Intel Syntax

1.2 Grundlegendes Szenario

Von großer Bedeutung für die Implementierung einer Abwehrmaßnahme ist die Definition eines Angriffsszenarios. Die hier vorgestellten Methoden zur Überprüfung des Kontrollflusses sollen CFI unter der Prämisse sicherstellen, dass ein Angreifer in der Lage ist das gesamte Datensegment eines Prozesses zu manipulieren [1]. Als möglicher Angriffsvektor für eine derartige Kontrolle über Hauptspeichersegmente könnte eine *Buffer Overflow*-Schwachstelle ausgenutzt werden. Ein Beispiel hierfür zeigt Listing 1.

```c
#include <stdio.h>

int main(){
    int a = 5;
    char str[4];
    printf("Enter a string: ");
    gets(str);
    printf("%i\n", a);
    return 0;
}
```

Listing 1: Beispiel eines für einen Buffer-Overflow anfälligen C-Programmes

Obiges Programm liest einen String von der Konsole und speichert diesen in einem vier Byte großen Array ab. Übersteigt die Länge der eingegeben Zeichenkette die Größe des Arrays, so werden die weiteren Bytes ebenfalls abgespeichert – und überschreiben so die Variable `a`. Während für die Zeichenkette `"AAA"` das Array `str` ausreichend groß ist, wird bei `"AAAAA"` der Zahlenwert 65 ausgegeben.[2] Umgewandelt in das Hexadezimalsystem ergibt dies 41 – die ASCII-Repräsentation des Buchstaben A.

Damit das Beispiel nach Kompilierung mit gcc wie oben beschrieben ausgenutzt werden kann, muss gegebenenfalls per `-fno-stack-protector` der *Stack-Protecting* Mechanismus deaktiviert werden. Während in Listing 1 ein Buffer-Overflow auf dem Stack demonstriert wird, gibt es äquivalente Schwachstellen auch für Speicherbereiche auf dem Heap.

CFI zielt jedoch nicht darauf ab, einen Buffer-Overflow an sich zu verhindern. Vielmehr ist diese Methode auf die möglichen Auswirkungen einer solchen Lücke ausgerichtet. Damit die Abwehrmechanismen zuverlässig funktionieren können, müssen weitere Einschränkungen gelten, welche im Abschnitt 3.1 vorgestellt werden. Des Weiteren setzt [1] eine x86-Architektur mit Windows XP SP2 Betriebssystem voraus – inwiefern die vorgestellten Mechanismen auch unter anderen Systemen funktionieren, wurde nicht geprüft.

1.3 Shellcode im Stack

Um das Beispiel aus Listing 1 in ein realistisches Umfeld zu übertragen: Ein Webserver nimmt Anfragen entgegen, die URL wird ohne Überprüfung in ein Array geschrieben. Ein Angreifer kann somit durch zu lange URLs Daten im Hauptspeicher überschreiben. Das übliche Ziel des Angreifers ist es nun, über die gefundene Schwachstelle beliebigen Code auf dem Server zur Ausführung zu bringen.

Eine mögliche Herangehensweise hierzu stellt das direkte Einfügen von Shellcode auf dem Stack dar. Sie basiert grundsätzlich darauf, dass eine `ret`-Anweisung stets an eine Codestelle springt, deren Adresse

[2] Strings werden in C mit einem impliziten Nullbyte terminiert, daher ist die Zeichenkette `"AAA"` effektiv vier Byte groß.

üblicherweise zuvor von der aufrufenden Funktion auf den Stack gelegt wurde. Wird diese Adresse allerdings durch eine Buffer-Overflow Lücke manipuliert, wird der Sprung stattdessen an eine vom Angreifer definierte Zieladresse durchgeführt.

Ein solcher Shellcode ist von einigen Einschränkungen betroffen – beispielsweise darf er kein Nullbyte enthalten, da der Shellcode zum Zeitpunkt der Einfügung als String gespeichert wird. Ein Nullbyte würde diesen jedoch vorzeitig terminieren.

Die Ausführung von Code in Datensegmenten wie dem Stack ist jedoch auf modernen Systemen üblicherweise nicht möglich – wie in Abschnitt 3.1 beschrieben, wird dies durch die *Data Execution Prevention* verhindert.

1.4 ret2libc

Um einen Buffer-Overflow trotz geschützter Datensegmente zur Ausführung von Code zu nutzen, kann auch bereits im Arbeitsspeicher bestehender Code verwendet werden. Ein Angriff, welcher hierbei auf die Standard-C-Bibliothek *libc* zugreift, wird als *return to libc* oder kurz *ret2libc* bezeichnet [5, S. 412].

Ein gängiges Ziel ist hierbei die Funktion `system()` zum Zugriff auf eine Kommandozeile. Unter Linux geschieht dies durch `system("/bin/sh")`. Hierzu wird zum einen die Speicheradresse der `system()`-Funktion benötigt. Ebenfalls benötigt wird ein Zeiger auf das Argument für den Funktionsaufruf – im Beispiel auf den String `"/bin/sh"`. Somit ist es möglich, durch Überschreiben der Rücksprungadresse den beschriebenen Funktionsaufruf durchzuführen.

Adressrandomisierung erschwert derartige Angriffe, da die `system()`-Funktion nicht länger bei jedem Aufruf unter der selben Speicheradresse zu erreichen ist.

1.5 Return Oriented Programming

Return Oriented Programming oder kurz ROP stellt eine Verallgemeinerung von ret2libc dar. Statt eine komplette bestehende Funktion aufzurufen, setzt sich der Exploit aus mehreren Sprüngen zu einzelnen Maschinencodeanweisungen vor einem `ret` zusammen. Durch geschickte Verkettung mehrerer dieser *Gadget* genannten Bausteine entsteht das ROP Programm.

Für Turing-Vollständigkeit reicht bereits eine sehr kleine Menge an nutzbaren Instruktionen aus – bereits Subtraktion, „less-than“-Vergleich und bedingte Sprünge genügen [7, S.3]. Um jedoch die benötigten Anweisungen anspringen zu können, werden deren Adressen benötigt. Diese können anhand der Binaries und einem Tool wie dem frei verfügbaren *ropper*[3] bestimmt werden.

Ist es beispielsweise das Ziel, einen Wert vom Stack in das Register `rdi` zu schreiben, so kann dies mit dem Codeabschnitt aus Listing 2 erfolgen. Die hierfür benötigte Anweisung `pop rdi` wird hexadezimal als `x5f` kodiert – und ist unter der Adresse `x4006a3` in der Anweisung `pop r15` enthalten.

Wie auch bei *ret2libc* werden derartige Angriffe durch ALSR erschwert.

[3] vergleiche `https://github.com/sashs/Ropper`

```
400696: 48 83 c4 08           add    rsp,0x8
40069a: 5b                    pop    rbx
40069b: 5d                    pop    rbp
40069c: 41 5c                 pop    r12
40069e: 41 5d                 pop    r13
4006a0: 41 5e                 pop    r14
4006a2: 41 5f                 pop    r15
4006a4: c3                    ret
```

Listing 2: Ausschnitt eines C-Programmes als Assembler, erzeugt mit *objdump*

1.6 Zusammenfassung der Problematik

Alle der drei soeben vorgestellten Angriffe stellen einen Ausbruch aus dem vorgesehen Kontrollfluss dar, welcher durch überschreiben der Rücksprungadresse erreicht wird. An diesem Punkt setzen nun die im Folgenden präsentierten Methoden zum Schutz des Kontrollflusses an. Ziel ist es hierbei nicht, den Buffer-Overflow an sich zu verhindern. Dies ist in erster Linie Pflicht des Programmierers sowie zusätzlicher Schutzmechanismen des Kompilers. Ziel ist es stattdessen, für ein gegebenes Binary – welches möglicherweise für einen Buffer-Overflow anfällig ist – eine Verletzung des Kontrollflusses zu verhindern. Es handelt sich bei CFI somit um eine mitigierende Maßnahme.

Insbesondere bei nicht länger gewarteter oder veralteter Software würde sich ein derartiger Schutz anbieten. Möglicherweise stellt auch das Betriebssystem fortgeschrittene Maßnahmen wie Stack Protection oder ALSR nicht zur Verfügung. In diesem Fall kann CFI dennoch ein Mindestmaß an Sicherheit bieten.

2 CFG - Kontrollflussgraphen

Essentiell für die Implementierung von Control Flow Checking ist das Aufstellen eines Kontrollflussgraphen. Dies geschieht durch Analyse der Maschinenbefehle eines Programms. Von besonderem Interesse sind hierbei Sprunganweisungen, deren Ziel dynamisch berechnet wird.

Schutzmechanismen auf Basis von CFI werden nun genau dann benötigt, wenn die Sprungadresse dynamisch bestimmt wird. Ein `call [eax]` ist hiervon ebenso betroffen, wie ein `ret`. Wird als Sprungziel jedoch ein Label verwendet, beispielsweise `call square`, so kann dieser Sprung nicht manipuliert werden und bedarf auch keiner CFI-Kontrolle. Der Grund hierfür ist der in Abschnitt 3.1 vorausgesetzte Schreibschutz des Textsegmentes. Auch sämtliche Varianten von `jmp` können dynamische Ziele verwenden.

Dementsprechend wird auf Basis dieser Maschinenbefehle ein Kontrollflussgraph erstellt: Jede der genannten Sprunganweisungen entspricht einer Kante im Graphen. Alle möglichen Ziele einer Sprungquelle werden anschließend in einer Äquivalenzklasse zusammengefasst. Die Äquivalenzklasse erhält eine eindeutige ID – auf Basis dieser ID wird im Anschluss die Validierung eines Sprunges durchgeführt.

3 CFI - Control Flow Integrity

3.1 Voraussetzungen

Für die Implementierung von CFG gelten einige Voraussetzungen, die für einen funktionierenden Schutz des Kontrollflusses gelten müssen. Sind diese nicht oder nur teilweise erfüllt, so sinkt auch die Effektivität der hier vorgestellten Sicherheitsmaßnahmen entsprechend.

Zum einen wird der Aktionsradius des Angreifers definiert. Wie bereits eingangs erörtert, erlangt ein potentieller Angreifer Kontrolle über das Datensegment eines Prozesses im Arbeitsspeicher. Es ist ihm sowohl möglich, Daten auszulesen als auch zu manipulieren. Hierzu liegen allerdings Einschränkungen vor.

So erfordert die CFI Implementierung nach [1], dass Code-Segmente prinzipiell nicht beschreibbar sind.[4] Es darf einem Angreifer etwa nicht möglich sein, eingefügte Überprüfungen zu überschreiben. Des Weiteren wird davon ausgegangen, dass Data-Segmente nicht ausführbar sind. Diese als *Data Execution Prevention* bekannte Einschränkung wird von aktuellen Architekturen durch ein gesetztes *NX-Bit* realisiert. Ist diese Sicherheitsvorkehrung aktiviert, so stürzen Prozesse automatisch ab, sobald sie versuchen, Inhalte von Datensegmenten als Code auszuführen [6].

Damit die Überprüfung selbst nicht manipuliert werden kann, dürfen die verwendeten Register nicht von anderen Prozessen beschreibbar sein. Dies ist in gängigen Betriebssystemen gegeben. Schließlich muss es möglich sein, eine ausreichende Anzahl von einzigartigen ID-Werten für die Validierung der Sprünge zu erstellen. So wird für jedes mögliche Sprungziel eine eindeutige ID benötigt, die zudem nicht im restlichen Code vorkommen darf. Zudem beschreibt [1] die Implementierung von CFI unter einer x86 Architektur sowie dem Betriebssystem *Windows XP SP2*. Auf die Funktionalität der vorgestellten Methoden unter unixoiden Systemen oder unter 64-Bit Betriebssystemen geht die Quelle nicht ein. Da sich diese Systeme auf dieser Ebene jedoch nicht grundlegend voneinander unterscheiden, sollte CFI äquivalent anwendbar sein.

3.2 Implementierung

Für bestehendes Binary wird ein Kontrollflussgraph erzeugt, anhand dessen das Binary um CFI ergänzt werden kann. Man erhält erneut ein ausführbares Binary.

Abbildung 1: Prozess zur Implementierung von CFI

Im Folgenden soll gezeigt werden, wie ein Funktionsaufruf auf Basis von CFI nach [1] abgesichert werden kann. Ein Funktionsaufruf beinhaltet zwei Sprunganweisungen, die getrennt voneinander betrachtet werden. Zuerst wird mit `call ptr` die Codezeile mit der Adresse `ptr` aufgerufen, zudem wird die Adresse der auf `call` folgenden Anweisung als Rücksprungadresse auf den Stack gelegt. Zum anderen wird bei dem nächsten Aufruf einer `ret`-Anweisung ebenjene Rücksprungadresse vom Stack gelesen und als Sprungziel verwendet.

[4] Nonwritable Code(NWC)

Wenn entschieden wurde, dass für eine bestimmte Sprunganweisung zu einem bestimmten Ziel eine Absicherung benötigt wird, so wird der Maschinencode an diesen beiden Stellen modifiziert. Wie die Überprüfung umgesetzt wird, zeigen die Listings 3 und 4.

```
1 FF 53 08              call [ebx+8]           ; Aufruf eines Funktionspointers
2 ...
3 8B 44 24 04           mov eax, [esp+4]       ; Ziel der Funktion
```

Listing 3: Ursprüngliche Callanweisung und Sprungziel

```
1 8B 43 08              mov eax, [ebx+8]       ; Zeiger nach eax schreiben
2 3E 81 78 04 78 56 34 12 cmp [eax+4], 12345678h  ; Test des Labels vor Sprungziel
3 75 13                 jne error_label        ; Fehlerbehandlung
4 FF D0                 call eax               ; Funktionsaufruf
5 ...
6 3E 0F 18 05 78 56 34 12 prefetchnta [12345678h] ; Label als Argument für prefetch
7 8B 44 24 04           mov eax, [esp+4]       ; Ziel der Funktion
```

Listing 4: Listing 3 ergänzt um CFI Überprüfung

Listing 3 enthält einen `call`-Aufruf, welcher einen Funktionspointer verwendet. Das Ziel des Sprunges liegt im Arbeitsspeicher unter der Adresse `ebx+8` und ist möglicherweise durch einen Bufferoverflow modifizierbar. Ein mögliches Ziel dieses Aufrufs zeigt Zeile 3. In Listing 4 wird nun diese Sprunganweisung um eine CFI-Überprüfung ergänzt.

Zuerst wird die Zieladresse in das `eax`-Register kopiert. Anschließend wird überprüft, ob die vier Bytes *vor* dem Sprungziel identisch mit dem gewählten Label sind. Ist dies nicht der Fall, so wird mit `jne error_label` eine Fehlerbehandlungsroutine aufgerufen, welche beispielsweise den Prozess beendet. Bei einem validen Sprungziel erfolgt abschließend der Funktionsaufruf, die Zieladresse befindet sich bereits im `eax`-Register. Vor dem Sprungziel muss nun ebenfalls das zugehörige Label eingefügt werden. Eine Möglichkeit, dies zu realisieren, stellt die seiteneffektfreie `prefetchnta`-Anweisung dar. Sie erhält als Argument das entsprechende Label. `prefetchnta` selbst lädt einen Wert in den Cache und markiert ihn für eine einmalige Verwendung.

Ähnlich wie in Listing 4 können auch `ret`- oder `jmp`-Anweisungen modifiziert werden.

3.3 Vorteile

Die gezeigten CFI-Maßnahmen stellen in gewissen Rahmen den kontrollierten Ablauf eines Programms sicher. Insbesondere Sprünge an beliebige Codestellen können hierdurch effizient verhindert werden. Dass CFI dennoch keinen absolut wirksamen Schutz bieten kann, wird im folgenden Abschnitt 3.4 dargestellt. Um die Effektivität ihrer CFI-Implementierung zu testen, verwendeten [1] eine Suite von 18 für Buffer-Overflows anfällige Programme. An jedem dieser Programme wurden Shellcodes, ret2libc-Angriffe und Zeigermanipulationen gestartet. Nach Anwendung von CFI gelang es in allen 18 Fällen Angriffe auf den Kontrollfluss zu verhindern.

Der grundlegende Prozess zur Implementierung nach [1] beinhaltet, dass CFI nicht nur bei der Kompilierung eingebunden werden kann: Auch bereits erstellte Binaries sollen noch mit entsprechenden Ergänzungen versehen werden können. Dies erlaubt die Anwendung von CFI auch bei Programmen, für

die die Kompilierung nicht länger möglich oder mit großem Aufwand versehen ist – beispielsweise weil der Quellcode nicht vorhanden ist.

3.4 Nachteile

Neben den aufgezeigten Vorteilen in Bezug auf erhöhte Sicherheit und Abwehr von Angriffen auf Sprungadressen bringt die Anwendung von CFI auch Nachteile bzw. Einschränkungen mit sich.

So stellt CFI an sich keinen Schutz vor Buffer-Overflows dar. Hierfür sind andere Schutzmechanismen wie SSP oder sichere Funktionen zuständig. Beispielsweise sollte statt der Funktion `gets()` aus Listing 1 stets `fgets()` verwendet werden. Aufgrund dessen und der Definition von CFI können Angriffe, die den Kontrollflussgraphen nicht verletzen, auch nicht erkannt werden. Wird etwa in Listing 5 die Variable `command` überschrieben, kann eine beliebige Datei mit den selben Rechten wie das ursprüngliche C-Programm ausgeführt werden.

```
1 char command[] = "/harmless/executable";
2 system(command);
```

Listing 5: C-Programm mit `system()` Aufruf

Des Weiteren ist *Fault Tolerance* kein Ziel von CFI. Wie eingangs erwähnt, befasst sich hiermit das Gebiet des Control Flow Checkings. Wenn auch manche durch zufällig gekippte Bits entstandene Kontrollflussfehler entdeckt werden können, wird prinzipiell davon ausgegangen, dass manche Speicherbereiche unveränderbar sind. Hierunter fallen etwa die Register oder die Textsegmente. Fehler durch radioaktive Strahlung können jedoch im gesamten Arbeitsspeicher auftreten. Somit sind die Prüfmechanismen nicht länger zuverlässig.

Eine weitere Problematik stellt die Definition des Kontrollflusses dar: Wird dieser fälschlicherweise zu stark einschränkend definiert, so kann das Programm auch ohne Angriff abstürzen.[5] Auch das Gegenteil ist problematisch. Erlaubt der Kontrollflussgraph zu viele mögliche Sprungziele, könnte dies einen Angriff ermöglichen.

Auch die Implementierung von CFI hat Nachteile technischer Natur. So wird für die Überprüfung in Listing 4 das `eax`-Register verwendet. Möglicherweise ist der folgende Programmteil jedoch abhängig vom aktuellen Wert des Registers – es könnten somit Seiteneffekte auftreten, die den weiteren Programmablauf stören. Auch die `cmp`-Anweisung setzt je nach Resultat verschiedene Flags, was zu selbigem Problem führen könnte. Um dies zu umgehen, müssten die Register vor der CFI-Überprüfung etwa auf dem Stack zwischengespeichert werden. Während ihrer Arbeit zu CFI entdeckten [1] lediglich eine geringe Anzahl an Codestellen, an welchen eine Sicherung der Register notwendig gewesen wäre. Insbesondere handgeschriebener – also nicht kompilergenerierter – Code sei demnach hierfür anfällig.

Jegliche Art von dynamisch eingebundenem Code ist problematisch für CFI: Werden etwa dynamisch gelinkte Bibliotheken verwendet, die ohne CFI kompiliert wurden, so liegen an diesen Stellen wiederum angreifbare Schwachstellen vor. Auch selbstmodifizierender Code oder Code der zur Laufzeit erzeugt wird ist nicht Teil der Kontrollflussüberprüfung.

Zudem erlaubt die Implementierung aus Listing 4 selbst eine Möglichkeit zur Verletzung des Kontrollflusses: Zur Validierung des Sprungzieles wird die `cmp`-Anweisung verwendet, sie erhält als Argumente

[5] *false positive*

eine Speicheradresse vor dem eigentlichen Sprungziel und das Label auf welches hin getestet wird. Somit steht an der Stelle des Vergleiches selbst auch das Label, ein Sprung an die auf `cmp` folgende Anweisung ist also möglich. Dies resultiert in einer Endlosschleife zwischen den Zeilen 3 und 4 des Listings. [1] betrachten dies nicht als Einschränkung der Funktionsfähigkeit von CFI, da an Angreifer mit voller Kontrolle über das Datensegment viele Möglichkeiten zur Erzeugung von Endlosschleifen habe.

Das Einfügen von Maschinenbefehlen zur Überprüfung eines Sprunges führt dazu, dass relative Sprungadressen nicht länger an die korrekte Stelle zeigen. Diese müssen anschließend korrigiert werden.

Durch die zusätzlichen Maschinenbefehle entsteht ein gewisser Overhead – sowohl bei der CPU-Auslastung während der Ausführung als auch in Bezug auf die Größe der Binaries. Gegebenenfalls muss der Zugewinn an Sicherheit gegen die dadurch entstehenden Nachteile aufgewogen werden. Genauere Betrachtungen zum Overhead erfolgen in Abschnitt 3.6.

Der gravierendste Nachteil der vorgestellten Implementierung liegt jedoch in der Zustandslosigkeit des zugrundeliegenden Kontrollflussgraphen. Wird eine Funktion `x()` aufgerufen, so wird sowohl beim Aufruf als auch beim Return eine CFI-Überprüfung durchgeführt. Üblicherweise sollte ein Return wieder an jene Stelle im Code zurückkehren, von welcher die Funktion aus aufgerufen wurde. Beide Überprüfungen sind jedoch voneinander unabhängig. Wird nun die Rücksprungadresse manipuliert, so werden Sprünge an alle anderen Adressen, von denen aus `x()` aufgerufen wird, nicht als Verletzung des Kontrollflusses erkannt. Im Beispiel von Listing 6 wird innerhalb der `main()`-Funktion `someFunction()` aufgerufen, welche wiederum `x()` ausführt. Wird nun die Rücksprungadresse von `x()` manipuliert, so kann auch die Funktion `secretFunction()` als Sprungziel verwendet werden. Insbesondere Standardfunktionen wie `malloc()`, `printf()` oder `strcpy()` werden üblicherweise von vielen Codestellen aus aufgerufen. Gelingt es, eine dieser Funktionen entsprechend anzugreifen, so kann ein Angreifer eine Vielzahl möglicher Sprungziele verwenden.

```
1  int main(){
2      ...
3      someFunction();
4      ...
5  }
6
7  int someFunction(){
8      x();
9  }
10
11 int secretFunction(){
12     x();
13     puts("secret password");
14 }
```

Listing 6: Beispiel eines C-Programmes mehreren Methodenaufrufen

Funktionen, die in Abhängigkeit von den übergebenen Argumenten die eigene Rücksprungadresse überschreiben können, werden als *dispatcher functions* bezeichnet. Beispiele hierfür aus der Standardbibliothek von C werden in Listing 7 aufgelistet. Auch Funktionen, welche eine *dispatcher function* aufrufen, sind dementsprechend selbst ebenfalls *dispatcher functions*, da auch deren Rücksprungadresse möglicherweise überschrieben werden kann. Aus den genannten Gründen betrachtet [2, S. 167] CFI

```
memcpy();
printf();
strcat();
fputs();
```

Listing 7: Beispiele für *dispatcher functions*

in dieser Form prinzipiell als defekt. Um derartige Angriffe zu verhindern wird ein *protected shadow call stack* benötigt. Hierbei werden die Rücksprungadressen in einem vor Manipulation geschützten Speicherbereich hinterlegt. Bei jedem Return wird die eigentliche Rücksprungadresse durch einen Vergleich mit der Rücksprungadresse vom shadow call stack validiert. Damit wird ein möglicher Angriff auf Rücksprungadressen auf das bloße Rückabwickeln des Callstacks reduziert. Ein *protected shadow call stack* bringt jedoch zwei Probleme mit sich: Zum einen stellt sich die Frage der technischen Realisierung eines geschützten Speicherbereichs, zum anderen wird weiterer Overhead angefügt [2, S.164].

3.5 Hardwarerealisierung

Neben der vorgestellten Implementierung als Sequenz einfacher Maschinenbefehle wäre es denkbar, dass die benötigten Operationen zur Absicherung des Kontrollflusses in den Befehlssatz eines Prozessors integriert werden. Um dies zu erreichen wird neben einem `label ID` eine Anweisung `call ID, DST` benötigt, die einen Sprung an die Adresse `DST` nur durchführt, falls an dieser Codestelle die entsprechende `ID` als Label eingetragen ist. Identisch hierzu prüft auch `ret ID` beim Rücksprung. Laut [1, S.9] ist nicht zu erwarten, dass eine derartige hardwarenahe Implementierung von CFI in kommende Prozessorgenerationen Einzug findet. Da die genannten Operationen im Vergleich zu den vorgestellten Maschinencodebefehlssequenzen atomar sind, würden sie möglicherweise einen höheren Grad an Sicherheit bieten. Außerdem wären diese Befehle frei von Seiteneffekten, überschreiben also keine Register. Es ist ferner anzunehmen, dass eine derartige hardwarenahe Implementierung effizienter als die softwareseitige Lösung ist – sowohl was Umfang und Komplexität der Maschinenbefehle als auch deren Ausführungszeit angeht.

3.6 Overhead

Um festzustellen, welche zusätzliche Last durch die Anwendung von CFI erzeugt wird, wurden Tests mit dem *SPEC2000* Benchmark durchgeführt. Dieser Benchmark beinhaltet die Ausführung diverser auf C/C++ basierender Programme, wie etwa gcc oder gzip. In Abbildung 2 wird dargestellt, wie stark sich die Ausführungsdauer mit CFI erhöht. Im Durchschnitt benötigten die Programme 16 % mehr Zeit.

Die Erstellung des Kontrollflussgraphen und die Anpassung der Maschinenbefehle – also der gesamte Rewritingprozess – benötigte pro Binary etwa zehn Sekunden. Durch die zusätzlichen Maschinencodebefehle wuchsen die Dateigrößen im Mittel um 8 % an.

3.7 Anwendung von CFI in der Praxis

Sowohl Clang als auch Visual Studio unterstützen Kompilierung von C-Programmen mit Überprüfung des Kontrollflusses. Unter Clang wird dies durch `-fsanitize=cfi` aktiviert, das als *Control Flow Guard* bezeichnete Gegenstück unter benötigt die Option `/guard:cf` [4, 8]. Die genannten Compiler setzen CFI jedoch nicht zwangsweise so um, wie in [1] beschrieben wurde. Eine offizielle Implementierung des Coderewritingprozesses nach [1] steht nicht zur Verfügung.

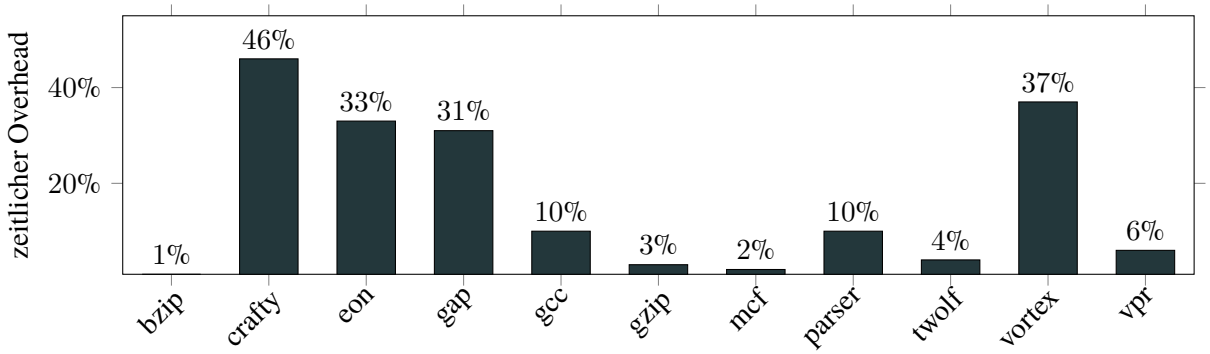

Abbildung 2: Zeitlicher Overhead bei der Ausführung des SPEC2000 Benchmarks mit CFI

Das Chromium-Projekt plant, zukünftige Releases mit der entsprechenden Option von Clang zu kompilieren. Tests ergaben hierbei 1% CPU Overhead und 7 - 9% größere Binaries [3].

```
1  int square(int a) {
2      return a*a;
3  }
4
5  int main(int argc, char *argv[]) {
6      int a = 0xCAFE;
7      int (*functionPtr)(int);
8      functionPtr = &square;
9      int b = functionPtr(a);
10     return 0;
11 }
```

Listing 8: C-Programm mit Funktionspointer zur Analyse von CFI unter Visual Studio

Das C-Programm aus Listing 8 ruft die Funktion `square()` über einen Funktionspointer auf. Hierdurch wird erreicht, dass Visual Studio bei der Kompilierung mit der vorgestellten Option `/guard:cf` Überprüfungscode einfügt.

Basierend auf dem Kompilat von Listing 8 wurde mit dem Tool *dumpbin*[6] ein Auszug aus den generierten Maschinenbefehlen erzeugt, welche in Listing 9 dargestellt werden. Neben kleineren Offsetanpassungen werden hierbei die Zeilen neun bis zwölf eingefügt, welche den Pointer `___guard_check_icall_fptr` aufrufen. Die aufgerufene Funktion `square()` bleibt von der CFI-Maßnahme unberührt und wird nicht verändert. `___guard_check_icall_fptr` verweist auf die entsprechenden Prüfroutinen in der geladenen *ntldll.ddl*. Insgesamt werden hierbei 15 zusätzliche Assembleranweisungen pro Funktionsaufruf ausgeführt. Tests unter Visual Studio 2015 ergaben jedoch, dass nur überprüft wird, ob das Sprungziel Beginn einer gültigen Funktion ist. Echte Kontrollflussüberprüfung konnte nicht festgestellt werden. Diese Funktion wird scheinbar durch eine Liste in den Headerdaten der ausführbaren Datei realisiert. Diese „Guard Cf Function Table“ kann mit `dumpbin /headers /loadconfig any.exe` ausgelesen werden.

[6] Äquivalent zu objdump unter gcc.

```
_main:
    push    ebp
    mov     ebp,esp
    sub     esp,10h
    mov     dword ptr [ebp-8],0CAFEh
    mov     dword ptr [ebp-0Ch],offset ?square
    mov     eax,dword ptr [ebp-8]
    push    eax
    mov     ecx,dword ptr [ebp-0Ch]
    mov     dword ptr [ebp-4],ecx
    mov     ecx,dword ptr [ebp-4]
    call    dword ptr [___guard_check_icall_fptr]
    call    dword ptr [ebp-4]
    add     esp,4
    mov     dword ptr [ebp-10h],eax
    xor     eax,eax
    mov     esp,ebp
    pop     ebp
    ret
```

Listing 9: Assemblerbefehle der Funktion `main()` aus Listing 8

4 Fazit / Persönliche Einschätzung

Neben CFI werden bereits eine Vielzahl anderer Schutzmechanismen gegen derartige Angriffe eingesetzt. Hierzu zählen beispielsweise ALSR zur Randomisierung der Adressen von eingebundenen Bibiotheken oder auch *stack canaries*, welche zufällige Bytes am Ende von Puffern oder vor der Rücksprungadresse ablegen. Nach Schreiboperationen auf dem Puffer oder vor dem Return kann überprüft werden, ob das stack canary geändert wurde – was auf einen Bufferoverflow hindeutet.

All diese Maßnahmen können jedoch keinen absoluten Schutz bieten. In erster Linie ist nach wie vor in der Verantwortung des Programmierers, seinen Code so zu organisieren, dass die in Abschnitt 1 beschriebenen Angriffe möglichst verhindert werden. Da Irren nun mal menschlich ist, stellt die Anwendung dieser Maßnahmen dennoch ein zusätzliches Maß an Schutz dar.

Literaturverzeichnis

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson und Jay Ligatti. „Control-flow integrity principles, implementations, and applications“. In: *ACM Transactions on Information and System Security* 13 (1 2009), S. 1–40. ISSN: 10949224. DOI: 10.1145/1609956.1609960 (siehe S. 1, 2, 5–9).

[2] Nicholas Carlini, Antonio Barresi, Mathias Payer, David Wagner und Thomas R. Gross. „Control-Flow Bending: On the Effectiveness of Control-Flow Integrity“. In: *24th USENIX Security Symposium (USENIX Security 15)*. Aug, Washington, D.C.: USENIX Association, 2015, S. 161–176. ISBN: 978-1-931971-232. URL: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/carlini (siehe S. 8, 9).

[3] Chromium Projects. *Control Flow Integrity*. URL: https://www.chromium.org/developers/testing/control-flow-integrity (besucht am 21. 04. 2016) (siehe S. 10).

[4] Clang Team. *Control Flow Integrity*. 2015. URL: http://llvm.org/releases/3.7.0/tools/clang/docs/ControlFlowIntegrity.html (besucht am 21. 04. 2016) (siehe S. 9).

[5] Jon Erickson. *Hacking. Die Kunst des Exploits*. ger. Dt. Ausg. der 2. amerikan. Aufl. Heidelberg: Dpunkt.-Verl., 2009. 505 S. ISBN: 978-3-898645362. URL: http://deposit.d-nb.de/cgi-bin/dokserv?id=3079599&prov=M&dok_var=1&dok_ext=htm (siehe S. 1, 3).

[6] Enes Goktas, Elias Athanasopoulos, Herbert Bos und Georgios Portokalidis. „Out of Control: Overcoming Control-Flow Integrity“. In: *2014 IEEE Symposium on Security and Privacy (SP)*. (San Jose, CA). Hrsg. von IEEE. 2014, S. 575–589. DOI: 10.1109/SP.2014.43 (siehe S. 5).

[7] Andrei Homescu, Michael Stewart, Per Larsen, Stefan Brunthaler und Michael Franz. „Microgadgets: Size Does Matter in Turing-complete Return-oriented Programming“. In: *Proceedings of the 6th USENIX Conference on Offensive Technologies*. Hrsg. von USENIX. WOOT'12. Berkeley, CA, USA: USENIX Association, 2012, S. 7. URL: http://dl.acm.org/citation.cfm?id=2372399.2372409 (siehe S. 3).

[8] Microsoft. *Control Flow Guard*. 2016. URL: https://msdn.microsoft.com/en-us/library/windows/desktop/mt637065(v=vs.85).aspx (besucht am 21. 04. 2016) (siehe S. 9).