Masterarbeit

Generische Graphanalyse in LLVM:
Bessere Fehlerprüfung zur Compilezeit mithilfe von Programm- und Datenflussgraphen

Gerion Entrup

1. August 2017

angefertigt am
Institut für Systemsicherheit
Technische Universität Braunschweig

zusammen mit dem
Institut für Theoretische Informatik
Gottfried Wilhelm Leibniz Universität Hannover

Erstprüfender:	Prof. Dr. Heribert Vollmer
Zweitprüfender:	Prof. Dr. Konrad Rieck
Betreuer:	Christian Wressnegger

Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig und ausschließlich mit Hilfe der angegebenen Quellen und Hilfsmittel angefertigt habe.

Hannover, den 1. August 2017 Gerion Entrup

Inhalt

Inhalt ..... **i**
Tabellen iii
Abbildungen iii
Codeblöcke iv

1 Einleitung **1**
1.1 Einordnung 2
1.2 Aufbau der Arbeit 3

2 LLVM **5**
2.1 Aufbau 6
2.2 IR 7
2.2.1 Hierarchie 7
2.2.2 Aufbau einer Instruktion 8
2.2.3 Typen 9
2.2.4 Signedness 9
2.2.5 Debug-Informationen 10
2.3 API 11
2.3.1 LLVM-Pass 11
2.3.2 Abbildung der IR 11
2.3.3 Verknüpfungen 12
2.3.4 Use-Objekte 13

3 Gremlin 15
3.1 Geschichte 15
3.2 Sprachaufbau 15
3.2.1 Formaler Aufbau 16
3.2.2 Traversal-Schritte 18
3.2.3 Programmstrukturen in Gremlin 21
3.2.4 Turing-Vollständigkeit 23
3.3 libgremlin 25
3.3.1 Probleme 25
3.3.2 Algorithmik 27

4 Gremlin-Pass 31
4.1 Abgebildeter Graph 31
4.1.1 Knoten 31
4.1.2 Kanten 33

5 Bug-Klassifizierung 37
5.1 Ganze Zahlen in C und C++ 37
5.2 Integer-Schwachstellen 38
5.2.1 Vorzeichenfehler 38
5.2.2 Truncation 39
5.2.3 Unter- und Überlauf 39
5.3 Probleme des Architekturwechsels 40
5.4 Suchmuster 40
5.4.1 Vorzeichenfehler 40
5.4.2 Truncation 41
5.4.3 Unter- und Überlauf 42
5.4.4 Implementierung 43

6 Auswertung 49
6.1 Messungen 49
6.1.1 Testset 49
6.1.2 Testumgebung 50
6.1.3 Quantitative Messungen 50
6.1.4 Messung des Aufwands 52
6.2 Fallstudien 56
6.2.1 Übersicht 56
6.2.2 CVE-2007-1884 56
6.2.3 CVE-2013-0211 57
6.2.4 Überlauf in zlib 58
6.2.5 Überlauf in Audacity 59

7 Zusammenfassung 61
7.1 Ausblick 62

Anhang 63
A.1 Benutzung von libgremlin und dem Gremlin-Pass 63
A.2 Testsets im Detail 65

Literatur 67

Tabellen

4.1 Übersicht aller Attribute für verschiedene Knotentypen 32
5.1 Speicherbreiten der C-Datentypen für 32 und 64 Bit 38
5.2 Übersicht über alle Anfragen und deren Fehlerklassen 42
6.1 Absolute und relative Treffer der Anfragen 51

Abbildungen

2.1 Aufbau von LLVM 6
2.2 Hierarchiestufen der LLVM-IR 8
3.1 Gremlin-Beispielgraph 17
3.2 Graphrepräsentation des Eingabebandes der Turingmaschine 24
4.1 `HIERARCHIE_EDGE`s 33

4.2 CFG_EDGEs 34
4.3 USE_EDGEs 34
4.4 LOOP_EDGEs 35
4.5 Argument-Kanten 35
6.1 Verlängerung der Kompilierzeit auf T_{SPEC} 52
6.2 Verlängerung der Kompilierzeit auf T_{Time} 54
6.3 IR-Instruktionen abhängig von C-/C++-Quellcodezeilen 54
6.4 Zeitverbrauch abhängig von Anzahl IR-Instruktionen 55

Codeblöcke

2.1 IR-Debug-Informationen 10
3.1 Simulation einer Turingmaschine 24
3.2 Auswertung der Traverser 28
3.3 Auswertung der Barrieren 28
3.4 Auswertung eines Traversers 29
5.1 Beispiel für einen Vorzeichenfehler 39
5.2 Beispiel für eine Truncation 39
5.3 Beispiel für einen Überlauf 40
5.4 Erste Teilanfrage Q.T.1 und Q.T.2 43
5.5 Zweite Teilanfrage Q.T.1 43
5.6 Zweite Teilanfrage Q.T.2 44
5.7 Anfrage Q.S.1 45
5.8 Anfrage Q.S.2 46
5.9 Anfrage Q.S.3 47
5.10 Anfrage Q.O.1 47
5.11 Anfrage Q.O.2 48
6.1 Kritische Stelle von CVE-2013-0211 57
6.2 IR-Code der kritischen Stelle von CVE-2013-0211 58
6.3 Überlauf in zlib 58
6.4 Truncation in Audacity 59
6.5 Überlauf in Audacity 60
A.1 Benutzung und Ausgabe des Gremlin-Pass 63
A.2 Minimalbeispiel für die Benutzung der libgremlin 64

Kapitel 1
Einleitung

1998 schrieb Robert Tappan Morris das erste Programm, das sich selbstständig durch die Ausnutzung einer Sicherheitslücke verbreiten konnte. Es enthielt keinen Schadcode, legte aber alleine durch die Rechenlast seiner Verbreitungsroutine tausende Computer lahm [1]. Seitdem ist Schadsoftware zu einer realen Bedrohung geworden.

Auch in der heutigen Zeit ist Malware ein großes Thema, die in der immer vernetzteren Welt verheerenden Schaden anrichten kann. Als Beispiel sei die in diesem Jahr aufgetauchte Ransomware[1] „WannaCry“ genannt, die u. a. dadurch traurige Berühmtheit erlangte, die Infrastruktur mehrerer Krankenhäuser in England lahmgelegt zu haben [2].

Der Eintrittspunkt einer Malware ist zumeist eine Sicherheitslücke, also ein Bug, der dazu genutzt werden kann, Schadcode einzuschleusen und auszuführen. Aus diesem Grund ist es essentiell, Methoden und Techniken zu entwickeln, Sicherheitslücken (oder auch simple Programmfehler) schon möglichst früh zu finden und zu schließen.

Oft folgen Sicherheitslücken ähnlichen Gebrauchsmustern – „Fallen“, die oft durch das Design der Programmiersprache entstehen und von Programmierern unabsichtlich eingebaut werden. Solche Gebrauchsmuster kann man mit einer automatischen Suche im Quelltext finden und aufzeigen. Allerdings werden derartige Programme schnell sehr komplex und sind zumeist auf eine Programmiersprache beschränkt.

Wünschenswert wäre ein Tool, das möglichst einfach und schnell nach beliebigen Mustern suchen kann und dies nicht nur in einer Sprache. Mit LLVM existiert bereits ein Framework, das viele verschiedene Programmiersprachen zu einer einzigen Zwischensprache übersetzt. Thema dieser Arbeit ist es, ein Verfahren zu entwickeln, um genau auf dieser Zwischensprache möglichst einfach nach Mustern suchen zu können. Dazu wird Gremlin benutzt, eine Programmiersprache, die speziell dazu entworfen wurde, Graphen zu durchsuchen.

Als Teil dieser Arbeit wurde Gremlin in C++ reimplementiert und eine Schnittstelle geschrieben, die die in LLVM vorhandenen Graphstrukturen mit Gremlin durchsuchbar macht.

[1] Eine Malware, die Daten auf dem Rechner verschlüsselt und gegen Geld wieder entschlüsselt.

1.1 Einordnung

Das in dieser Arbeit geschriebene Programm analysiert den Quelltext eines Programms, ohne ihn auszuführen, und fällt damit in die Kategorie der *Static Analyser*.

Static Analyser sind zumeist spezifisch für eine Programmiersprache. Populär sind z. B. Findbugs für Java oder Lint für C[2] [3, 4]. Das LLVM Frontend Clang hat zudem einen eingebauten Static Analyser (für C und C++), der mit einer vorgefertigten Liste von Mustern nach gängigen Fehlern sucht, wie z. B. vergessene Speicherfreigaben. Eine vollständige Liste der Testfälle findet sich in [5].

Auf Basis der LLVM-Zwischensprache wurden die beiden Programme KLEE und KLOVER entwickelt (für C und C++). Sie erfordern spezielle Anweisungen im Quellcode und machen auf dieser Basis eine symbolische Ausführung um die Testfallabdeckung zu erhöhen bzw. Testfälle zu generieren [6, 7]. LAV hat sich zum Ziel gesetzt, mittels symbolischer Ausführung Programmfehler zu finden [8]. Mit LLBMC existiert eine statische Analyse der LLVM Zwischensprache, um Fehler in C-Code aufzudecken [8,9].

Eine musterbasierte Suche im Linux-Kernel-Umfeld lässt sich mit Coccinelle finden. Dieses Programm nimmt Pattern in einer speziellen Sprache, die dem Unified Diff-Format ähnelt, und kann so automatisch Code ersetzen, aber auch analysieren [10].

Durch Muster, die in einer speziellen Sprache geschrieben wurden, bei einer statische Analyse im Quellcode Sicherheitslücken zu finden, wurde u. a. in [11–14] erforscht.

Allen voran ist jedoch die Software Joern zu erwähnen, die von Fabian Yamaguchi im Rahmen seiner Dissertation entwickelt wurde. Joern ist ein Fuzzy Parser, der C- und C++-Quelltext analysiert und in Form eines speziell dafür entwickelten Graphen – dem Code Property Graphen – in eine Graphdatenbank einträgt. Diese Graphdatenbank kann dann mit Gremlin durchsucht werden [15,16]. Diese Arbeit basiert auf dem Ansatz von Fabian Yamaguchi, entsprechend wird ein ähnliches Pattern-Matching verwendet.

Trotz der ähnlichen Herangehensweise von Joern hat die Implementierung in dieser Arbeit in einigen Aspekten entschieden andere Eigenschaften:

[2] Lint war der erste Static Analyser überhaupt und wurde 1979 veröffentlicht.

- Es wird keine Datenbank aufgebaut, die erst anschließend durchsucht wird, sondern die Graph-Anfrage durchsucht direkt beim Übersetzen die LLVM-Datenstrukturen.
- Die LLVM-Zwischensprache ist nicht auf C und C++ beschränkt, sodass die Anfragen auch für andere Sprachen benutzbar sind.
- Der unterliegende Graph basiert in dieser Arbeit vollständig auf der LLVM-Zwischensprache und weist dadurch andere Eigenschaften als der Code Property Graph auf.

Außerdem gibt es das Programm *american fuzzy lop* (AFL), das zwar auf einem vollkommen anderen Ansatz basiert, allerdings oft ähnliche Probleme aufdeckt. AFL lässt das zu testende Programm massenhaft auf unterschiedlichen Eingaben laufen, die durch verschiedenartige Umkehrung von Bits einer „korrekten“ Eingabe gewonnen werden [17].

Alle bisher vorgestellten Ansätze sind allerdings sprachspezifisch, erfordern zusätzliche Zwischenschritte beim Entwickeln oder führen Tests anhand einer fest eingebauten Datenbank durch. Der hier vorgestellte Ansatz, die Zwischensprache von LLVM ohne Einschränkungen durchsuchbar zu machen, ist nach bestem Wissen der erste dieser Art.

1.2 Aufbau der Arbeit

In dieser Arbeit wird zuerst LLVM vorgestellt und ein Überblick über die Programmiersprache Gremlin gegeben. Danach werden die damit gesuchten Muster gezeigt und klassifiziert und anschließend evaluiert.

Kapitel 2
LLVM

Im Folgenden soll LLVM vorgestellt werden. LLVM ist ein Compiler-Framework, dessen Kern unabhängig von einer bestimmten Programmiersprache ist. In der Praxis wird es am häufigsten für C und C++ verwendet.

LLVM wurde 2002 von Chris Lattner im Rahmen seiner Masterarbeit bei Vikram Adve erstmalig entwickelt und vorgestellt [18]. Der Name LLVM stand ehemals für **L**ow **L**evel **V**irtual **M**achine, wurde aber inzwischen ohne die Bedeutung als Akronym zum Projektnamen erklärt [19,20].

2005 wurde Chris Lattner von Apple engagiert und wirkte dort maßgeblich an der Entwicklung von Clang mit [21], einem Compiler, der auf Basis von LLVM C, C++ und das von Apple präferierte Objective-C übersetzen kann [22, 23]. Clang strebt u. a. eine hohe Kompatibilität zu GCC an, war 2009 erstmals in der Lage einen BSD-Kernel zu übersetzen und kann inzwischen 94% des Debian-Archivs kompilieren [24,25]. Clang und LLVM haben als erklärtes Ziel, schneller als GCC zu sein und bessere Fehlermeldungen auszugeben. Auch durch die für Unternehmen oft günstigere Lizenz ist LLVM in Verbindung mit Clang deshalb eine vollwertige Konkurrenz zu GCC[3] [26]. Mit dem LLVMLinux Project existiert außerdem ein Projekt, das sich zum Ziel gesetzt hat, den Linux-Kernel mit Clang kompilierbar zu machen. Dies ist allerdings ohne zusätzliche Patches noch nicht möglich [27].

Mehrere Projekte basieren auf LLVM. Chris Lattner entwickelte während seiner Zeit bei Apple die Programmiersprache Swift und einen dazu passenden Compiler auf LLVM-Basis [21, 28]. Der Referenz-Compiler für die Programmiersprache Rust, die im Umfeld von Mozilla entwickelt wird und 2006 entstanden ist, benutzt LLVM [29]. GHC (der **G**lasgow **H**askell **C**ompiler) stellte 2010 ein Backend vor, das auf LLVM basiert [30]. Ein Überblick über Projekte, die LLVM verwenden, ist in [31] zu finden.

In diesem Kapitel wird zunächst der Aufbau von LLVM im Gesamtüberblick vorgestellt, anschließend einige Teile der LLVM eigenen Zwischensprache untersucht und zuletzt auf die Repräsentation der Daten und die Programmschnittstelle eingegangen.

[3] Programme auf Basis des GPL-lizenzierten Quellcodes von GCC müssen ebenfalls unter GPL lizenziert werden, bei LLVM und Clang ist das nicht der Fall.

2.1 Aufbau

Wenn ein Programm mit LLVM von einer bestimmten Hochsprache zu einer bestimmten Architektur übersetzt wird, muss es mehrere Zwischenstufen durchlaufen. Zuerst wird es von einem *Frontend* zu einer Zwischensprache – der Intermediate Representation (kurz IR) – übersetzt, durchläuft anschließend mehrere Optimierungsschritte und wird zuletzt von einem Backend in die Zielarchitektur übersetzt.

LLVM besteht im engeren Sinne nur aus den Optimierern und Backends. Die Frontends für bestimmte Programmiersprachen hängen zwar eng mit LLVM zusammen und werden – wie Clang – auch als Teilprojekt von LLVM entwickelt, sind aber dennoch abgegrenzte eigene Projekte.

Der Aufbau ist schematisch in Abbildung 2.1 dargestellt.

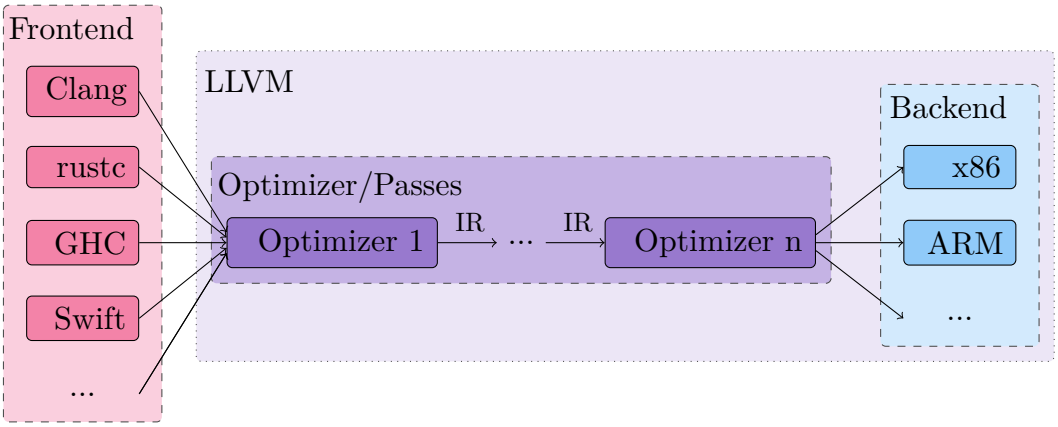


Abbildung 2.1 Schematischer Aufbau von LLVM. Zu erkennen ist die Aufteilung in Frontend, Optimizer und Backend. Für alle Übergänge (auch bei den nicht gekennzeichneten Kanten) wird die IR benutzt. Als LLVM wird meistens der hier gekennzeichnete Teil bezeichnet, LLVM wird aber auch als Name für das gesamte Projekt benutzt.

Eine Besonderheit, die LLVM von anderen Compilern abgrenzt, ist die Verwendung der IR in allen Zwischenschritten. Das macht es besonders einfach, neue Optimierungsschritte an beliebiger Stelle einzufügen.

LLVM beinhaltet weiterhin einige unabhängige Bausteine, die ihrerseits aber wieder auf der IR aufbauen, z. B. einen Just-in-Time Compiler oder einen Interpreter für die IR[4].

Zu den bekanntesten Frontends zählen der eben schon vorgestellte Clang und rustc.

[4] Auf Basis dieser Teile können neue Projekte entstehen, wie z. B. Cling, ein C++-Interpreter [32].

2.2 IR

Die IR wurde als Zwischensprache speziell für LLVM entworfen. Sie ist sowohl die Ausgabe der Frontends als auch aller Optimierungsstufen und kann sowohl zu vollwertigen Programmen verschiedener Architekturen kompiliert werden, als auch vom LLVM eigenen Interpreter interpretiert werden.

Die IR hat mehrere Designmerkmale, die im weiteren Verlauf noch genauer erklärt werden:

- Sie muss geeignet sein, um von allen Optimierern verwendet werden zu können.
- Sie soll ein kompaktes binäres Datenformat besitzen, aber im Textformat lesbar sein.
- Debug-Informationen dürfen bestehende Syntax nur unwesentlich verändern.
- Sie ist stark typisiert.
- Alle Instruktionen liegen im Single-State-Assignment (SSA) vor.

2.2.1 Hierarchie

LLVM und somit auch der IR liegt eine Hierarchie zugrunde, die an C oder Assemblercode orientiert ist.

Ein einzelner Befehl ist eine Instruktion (*Instruction*). Instruktionen haben einen sehr simplen Aufbau, der dem von Assembler ähnelt. Sie werden später noch genauer vorgestellt.

Instruktionen sind in einem einfachen Block (*BasicBlock*) gruppiert. Innerhalb eines einfachen Blocks kann nicht gesprungen werden, d. h. sämtliche Sprung- oder Return-Anweisungen führen zu einem anderen einfachen Block und sind somit immer die letzte Instruktion des Blocks.

Einfache Blöcke wiederum bilden den Rumpf einer Funktion (*Function*). Eine Funktion ist semantisch identisch zu einer Funktion in C.

Mehrere Funktionen sind in einem Modul (*Module*) zusammengefasst. Ein Modul bildet die größte Kompiliereinheit, die der Übersetzer verarbeitet und repräsentiert die gesamte Quellcode-Datei. Kompilierte Module werden anschließend mit einem Linker zu einem Programm verknüpft. Abbildung 2.2 visualisiert die verschiedenen Stufen.

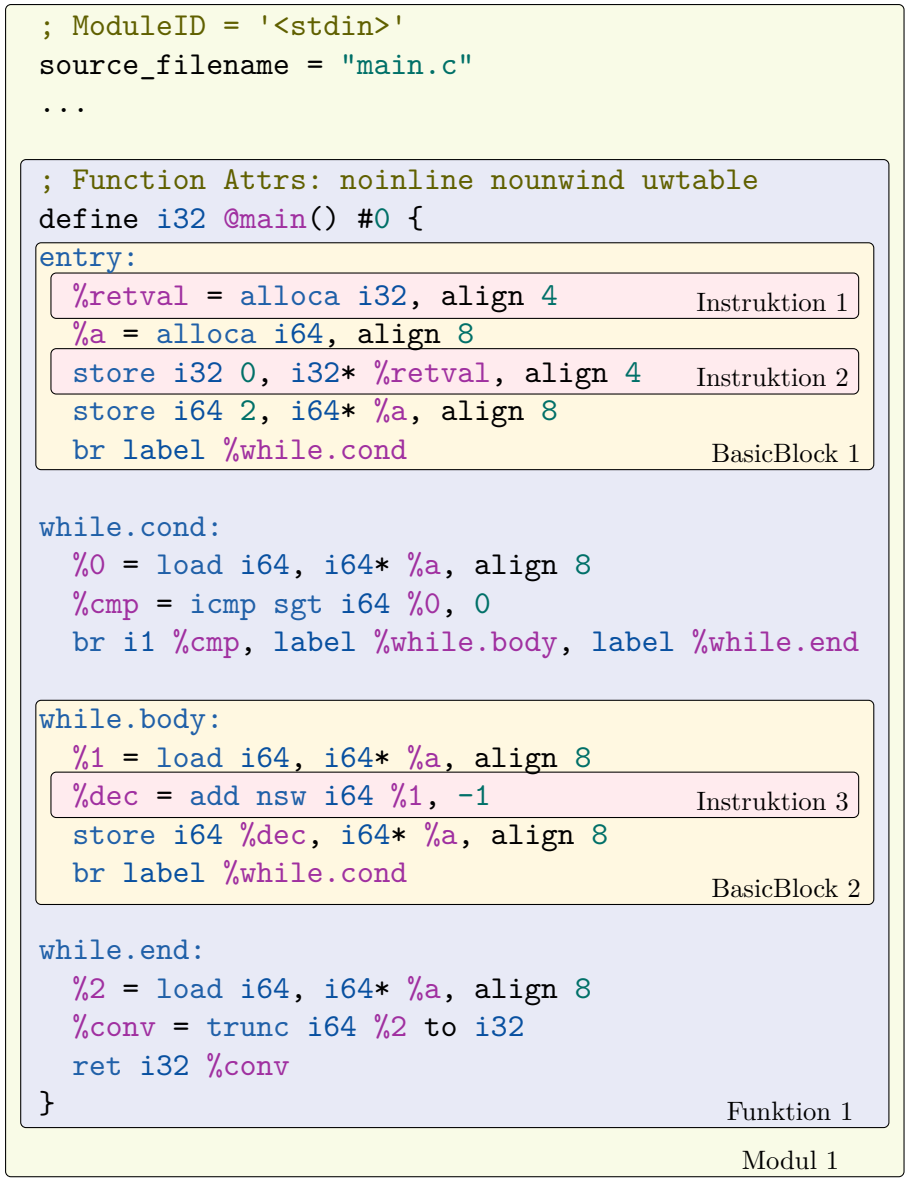


Abbildung 2.2 Hierarchiestufen der LLVM-IR. Dargestellt sind das Modul, die Funktion, zwei der vier einfachen Blöcke und drei der 15 Instruktionen.

2.2.2 Aufbau einer Instruktion

Eine Instruktion besteht aus verschiedenen Teilen.

- Optional einem Ziel
- Einem Opcode

- Einem oder mehreren Operanden
- Optionalen Debug-Informationen

Die in Abbildung 2.2 dargestellte Instruktion 3 hat z. B. als Ziel „`%dec`," als Opcode „`add nsw`" und als Operanden „`%1`" und „`-1`". An dem Beispiel kann man mehrere Merkmale der IR erkennen. Variablen, die mit einem „`%`" beginnen, sind „lokale Variablen". Variablen, die mit einem „`#`" beginnen, sind „globale Variablen". Absolutwerte werden direkt in den Code geschrieben. Variablen sind dabei keine Speicherbereiche im Arbeitsspeicher, sondern Register. Lese- und Schreibzugriffe auf den Arbeitsspeicher sind explizite Instruktionen („`load`" und „`store`").

Die IR arbeitet mit dem Single State Assignment (kurz SSA), d. h. die jede lokale Variable darf in ihrem Geltungsbereich nur einmal definiert bzw. beschrieben werden. Die Anzahl an lokalen Variablen ist entsprechend unbeschränkt. Der Name der Variable ist damit synonym für die die Instruktion.

2.2.3 Typen

Die IR ist stark typisiert, d. h. jeder Instruktion (bzw. jedem dadurch repräsentierten Datum) ist ein Typ zugeordnet. In Abbildung 2.2 sind verschiedene IR-Instruktionen dargestellt. Man erkennt dort unter anderem:

- „`i32`" und „`i64`" als Repräsentanten für 32 und 64 Bit breite Integer. Die IR ist dabei nicht auf Zweierpotenzen als Bitbreiten beschränkt, sondern kann eine beliebige Breite verarbeiten[5].
- „`i8*`" als Zeiger auf einen Speicherbereich mit – in diesem Fall – 8-Bit-Integern.

Außer diesen Typen gibt es noch weitere, wie z. B. Typen für Fließkommazahlen, Vektoren und Arrays.

2.2.4 Signedness

Im Gegensatz zu Hochsprachen wie C oder C++ haben LLVM-Datentypen kein Vorzeichen. Dies war eine bewusste Designentscheidung, die in früheren Versionen der IR noch anders war, zu einer Vereinfachung der Sprache geführt und bestimmte Optimierungen erst zugelassen hat [33].

Bestimmte Operationen von Hochsprachen (z. B. Vergleiche oder Expansionen) sind allerdings von dem Vorzeichen ihrer Operanden abhängig. In der IR werden

[5] Genaugenommen gibt es Grenzen, die in einer Breite zwischen 1 und $2^{23} - 1$ resultieren.

die Operationen daher in speziell für das Vorzeichen gültige Operationen aufgeteilt. Für einen „Größer als“-Vergleich zwischen Ganzzahlen gibt es darum z. B. sowohl eine vorzeichenbehaftete Operation („`icmp sgt`“, „signed greater than“, siehe auch Abbildung 2.2) und eine vorzeichenlose Operation („`icmp ugt`“).

2.2.5 Debug-Informationen

```
1  ...
2  %a = alloca i64, align 8
3  store i32 0, i32* %retval, align 4
4  call void @llvm.dbg.declare(metadata i64* %a, metadata !10, metadata !12), !dbg !13
5  ...
6  !10 = !DILocalVariable(name: "a", scope: !6, file: !1, line: 4, type: !11)
7  !11 = !DIBasicType(name: "long int", size: 64, encoding: DW_ATE_signed)
8  !12 = !DIExpression()
9  !13 = !DILocation(line: 4, column: 7, scope: !6)
```

Codeblock 2.1 Einbettung von Debug-Informationen in die IR

In der IR können optional Debug-Informationen eingebettet werden. Diese folgen dem Dwarf-Standard, der z. B. auch beim elf-Format[6] eingesetzt wird und von GDB[7] verarbeitet wird [34,35]. LLVM ist allerdings in der Lage, die Informationen in andere Formate zu konvertieren.

Die Debug-Informationen haben dabei folgende Design-Anforderungen:

- Sie sollen kompatibel mit Standard-Tools (wie GDB) sein.
- Sie müssen so eingebettet werden, das Code, der die IR verarbeitet, nicht angepasst werden muss.
- Optimierungen sollen Debug-Informationen nur in einem fest definierten Sinn verarbeiten.
- Sie dürfen die Quellsprache nicht einschränken.

In LLVM wird dies für C und C++ so gelöst, dass jede Anweisung einen „Verweis“ bekommen kann, der den ursprünglichen Ort im Quellcode beschreibt und seinerseits auf eine Debug-Einheit mit weiteren Informationen zeigt. Erstmals deklarierte Variablen werden zudem als Argumente in eine spezielle Funktion (`llvm.dbg.declare`) gegeben, die für alle Argumente Debug-Informationen – wie Typ oder Name – definiert.

[6] Das Binärformat, mit dem in unixoiden Systemen ausführbare Programme gespeichert werden.
[7] Der **G**NU **Deb**ugger, einer der populärsten Debugger.

In Codeblock 2.1 ist ein Beispiel für die Einbettung von Debug-Informationen mit Clang zu sehen. Die Zeilen 6 bis 9 befinden sind dabei ganz am Ende der Datei.

2.3 API

Ein großer Vorteil des LLVM-Compiler-Frameworks ist die öffentliche API. Von der API wird in dieser Arbeit extensiver Gebrauch gemacht, sodass sie im Folgenden kurz vorgestellt wird. Zuerst wird das Konzept eines „LLVM-Passes“ eingeführt, anschließend gezeigt, wie die IR über die Schnittstelle zugreifbar ist und welche Verknüpfungen bereitgestellt werden.

2.3.1 LLVM-Pass

LLVM (mit der Definition aus Abbildung 2.1) besteht ausschließlich aus verketteten Optimierungsschritten, denen zuletzt ein Backend folgt. Diese Optimierungsschritte heißen *Passes*. Um ausgeführt zu werden, muss sich ein Pass beim Framework registrieren. Er kann dabei optional Abhängigkeiten zu anderen Passes angeben. Die finale Ausführungsreihenfolge wird automatisch vom Framework berechnet.

Passes sind eng mit der LLVM-Hierarchie (siehe 2.2.1) verknüpft. Sie laufen nur auf einer bestimmten Ebene und dürfen diese Ebene nicht verlassen, um eine Ausführungsreihenfolge berechnen zu können. Unter anderem gibt es einen *ModulePass*, der auf dem gesamten Modul läuft, einen *FunctionPass* und *BasicBlockPass*, der für jede Funktion bzw. jeden einfachen Block aufgerufen wird, sowie einen *LoopPass*, der auf jeder Schleife (von innen nach außen) ausgeführt wird.

Die verschiedenen Passes sind als Klassen definiert. Um einen neuen Pass zu erstellen, muss von der entsprechenden Klasse geerbt und eine für diesen Pass spezielle Methode überschrieben werden.

2.3.2 Abbildung der IR

LLVM macht starken Gebrauch der Objektorientierung. Fast jede Einheit der IR wird auf eine eigene Klasse abgebildet, so haben z. B. Instruktionen, einfache Blöcke, Funktionen und Module eine eigene Klasse. Die Klassen haben dabei eine strikte Vererbungshierarchie. Da oft nur Basisklassen übergeben werden, aber Operationen auf den Unterklassen ausgeführt werden müssen, ersetzt LLVM die eher teure `dynamic_cast`-Operation durch eigene – deutlich schnellere – Aufrufe.

Die Basisklasse für fast alles ist `Value`. `Value` repräsentiert eine beliebige Einheit der IR, die einen Wert hat und benutzt werden kann. Sowohl Instruktionen, einfache Blöcke als auch Funktionen sind vom Typ `Value`.

Die meisten Klassen von LLVM – insbesondere alle Instruktionen – erben weiterhin von `User`. Die Klasse `User` repräsentiert eine Einheit, die Objekte vom Typ `Value` benutzt, siehe auch 2.3.4.

Eine Instruktion wiederum hat einen ganzen Unterbaum von spezialisierten Klassen, die die Art der Instruktion spezifizieren. Repräsentativ sollen hier ein paar Beispiele genannt werden:

- `CallInstr`: Repräsentiert einen Funktionsaufruf.
- `CmpInstr`: Repräsentiert einen Vergleich.
- `BinaryOperator`: Repräsentiert eine Verknüpfung auf zwei Werten und ist seinerseits wieder in Unterinstruktionen unterteilt.

Alle Objekte stellen einen Satz Methoden bereit, um entweder

- andere Objekte zu erreichen (siehe den nächsten Abschnitt) oder
- den Zustand zu lesen oder zu verändern.

Debug-Informationen sind versteckter abgebildet. Der aktuelle Ort im Hochsprachenquelltext ist direkt als Methode in der Klasse `Instruction` hinterlegt. Alles weitere, wie z. B. Typ-Informationen, muss allerdings aus dem Debug-Funktionsaufruf extrahiert werden.

2.3.3 Verknüpfungen

Die einzelnen Objekte der API sind bereits über eine Vielzahl von Methoden miteinander verknüpft. Die wichtigsten Verknüpfungen sind dabei: Hierarchie, Kontrollfluss und Benutzung.

Mit der Hierarchie-Verknüpfung ist die Verbindung zwischen Modul, Funktionen, einfachen Blöcken und Instruktionen gemeint. Die hierarchisch höhere Ebene stellt dabei jeweils einen Iterator für die hierarchisch tiefere Ebene bereit. Von einer hierarchisch tieferen Ebene kann über einen einfachen Methodenaufruf das jeweilige Objekt der höheren Ebene erreicht werden.

Kontrollflussübergänge existieren für einfache Blöcke und Instruktionen. Ein einfacher Block ist mit einem oder mehreren nachfolgenden Blöcken verknüpft und umgekehrt. Instruktionen innerhalb eines einfachen Blocks sind jeweils mit der Vorgänger- und Nachfolgeinstruktion verknüpft.

Darüber hinaus existieren noch je nach Objekttyp spezielle Verknüpfungen wie z. B. zwischen Funktionen und ihren Argumenten.

2.3.4 Use-Objekte

Mit `Use`-Objekten werden Benutzer („User“) und Operanden („Values“) miteinander verknüpft. Instruktion 3 in Abbildung 2.2 ist z. B. ein Benutzer, der die beiden Operanden `%0` und `-1` benutzt.

Die Verknüpfung geht dabei in beide Richtungen. Ein Benutzer kann alle Objekte abrufen, die er nutzt. Ein Objekt kann aber auch alle Objekte und damit Stellen abrufen, an denen es benutzt wird. In LLVM wird dies für bestimmte Optimierungen verwendet, dient aber im Kontext dieser Arbeit zur Realisierung von Fragestellungen wie, ob das Ergebnis der Instruktion danach noch im Rahmen einer Speicherzuweisung benutzt wird.
`Use`-Objekte sind überdies noch mit ihren Vor- und Nachfolgern, bezogen auf das `Value`-Objekt (also den Benutzten), verknüpft.

Kapitel 3
Gremlin

Gremlin ist eine Graph-Abfragesprache, die im Rahmen der Apache Foundation entwickelt wird. Die Sprache soll im Folgenden an einigen Beispielen eingeführt und anschließend formalisiert werden. Zuletzt wird die Reimplementierung in C++ vorgestellt.

3.1 Geschichte

Die Sprache wurde 2009 zusammen mit dem Tinkerpop-Projekt erstmalig vorgestellt [36]. 2015 wurde das gesamte Projekt als Incubator-Projekt in die Apache-Projektsammlung integriert und kurz darauf mit Version 3 eine stark überarbeitete und erweiterte Version vorgestellt. Seit 2016 ist Gremlin ein vollwertiges Apache-Projekt [37].

3.2 Sprachaufbau

Gremlin ist eine Sprache, um Graphen nach bestimmten Merkmalen zu traversieren. Sie ist dabei stark mit dem Tinkerpop-Projekt verbunden, das zum einen eine Graphdatenbank darstellt und zum anderen die Referenzimplementierung bereitstellt. Tinkerpop ist in Java implementiert.

Die Syntax von Gremlin ist darauf ausgelegt, in vielen bestehenden Sprachen nativ integrierbar zu sein. So ist Gremlin-Code (mit minimalen Anpassungen) u. a. valider Java-, Groovy-, Python-, C- und C++-Code.

Die Sprache besitzt bereits verschiedene Frontends, u. a. eine Implementierung für Java, Groovy und Python. Alle diese Frontends benutzten zur eigentlichen Auswertung die Tinkerpop-Implementierung.

Für Gremlin existiert eine Dokumentation, die allerdings viel mit Beispielen arbeitet, sowie die Dokumentation des Quellcodes und die Referenzimplementierung. Es gibt keine Spezifikation.

Marko Rodriguez hat 2015 die Sprache formalisiert und Turingvollständigkeit nachgewiesen [38]. Die Formalisierung wurde angepasst, um mit der Implementierung übereinzustimmen, die in dieser Arbeit entstanden ist. Beide Teile werden im Folgenden vorgestellt. Außerdem werden die Programmierparadigma von Gremlin angesprochen und übliche Sprachkonstrukte gezeigt.

3.2.1 Formaler Aufbau

Gremlin kann als Maschine verstanden werden, die einen Graphen traversiert. Die Maschine besteht aus drei Teilen:

1. Graph
2. Traversal
3. Traverser

Graph und Traversal werden dabei vom Benutzer vorgegeben, müssen allerdings einem bestimmten Schema entsprechen.

Graph

Der Graph, der Gremlin zugrunde liegt, ist ein attributierter gerichteter Graph[8]:

$$G = (V, E, \lambda, \beta) \tag{3.1}$$

U: Menge aller Objekte[9].
V: Menge der Knoten.
$E \subseteq (V \times V)$: Menge der Kanten.
Σ^*: Menge der Wörter des Alphabets Σ, das alle Buchstaben beinhaltet, die die Programmiersprache verarbeiten kann[10].
$\lambda : ((V \cup E) \times \Sigma^*) \to (U \setminus (V \cup E))$: Funktion, die Attributen von Knoten und Kanten einen Wert zuordnet.
$\beta : (V \cup E) \to \Sigma^*$: Funktion, die Knoten und Kanten eine Zeichenkette („Label“) zuordnet.

Wie der Funktion λ zu entnehmen ist, sind Attribute Schlüssel-Wert-Paare, bei denen der Schlüssel eine Zeichenkette ist und der Wert alles außer ein Knoten oder Kante sein kann.

[8] Gremlin ist allerdings in der Lage, die Kanten in „beide“ Richtungen zu gehen.
[9] Damit ist in der Referenzimplementierung ein Objekt des Typs „Object“ gemeint, also alles, was in Java darstellbar ist. Für die spätere Implementierung in C++ muss auf Objekten der Menge U eine Ordnungsrelation definiert sein.
[10] Damit ist im praktischen Sinn eine beliebige Zeichenkette der jeweiligen Programmiersprache gemeint.

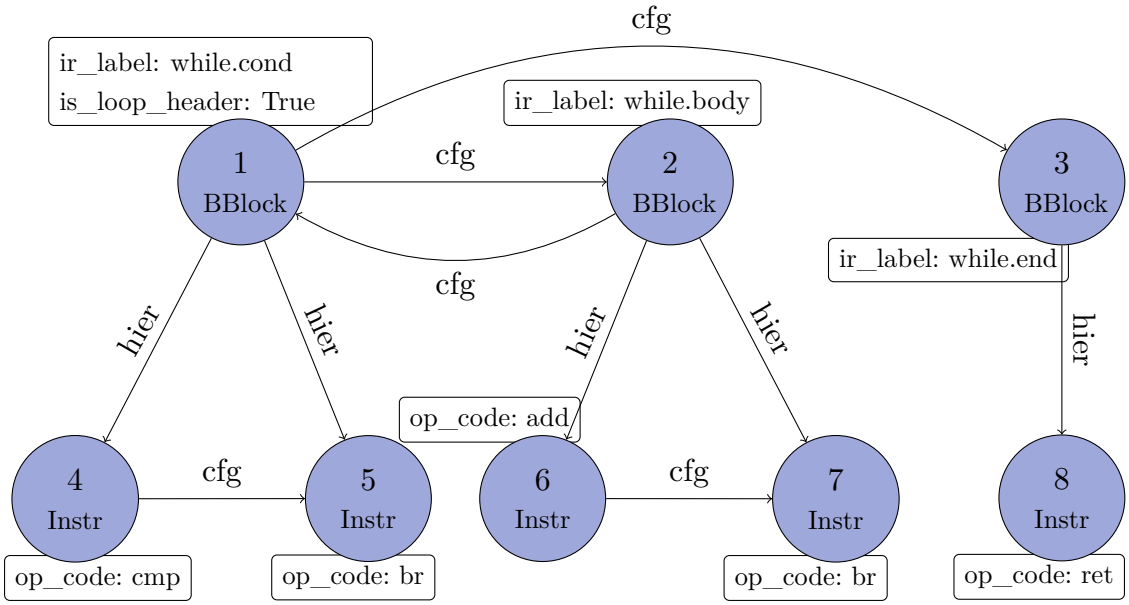


Abbildung 3.1 Gremlin-Beispielgraph. Auf die Kantenattribute wurde verzichtet. Abgebildet ist eine vereinfachte Form einer While-Schleife in der IR.

In Abbildung 3.1 ist ein Graph dargestellt, der den Kriterien entspricht und als Grundlage für nachfolgende Beispiele dienen soll. Der Graph repräsentiert in vereinfachter Form eine While-Schleife. Man erkennt als Knoten einfache Blöcke („BBlock“) und Instruktionen („Instr“), sowie Kontrollflusskanten („cfg“) und Hierarchie-Kanten („hier“).

Traversal

Eine Gremlin-Traversal ist der Quellcode, mit dem die Sprache arbeitet. Sie stellt einen Satz von Anweisungen bereit, die aufeinanderfolgend abgearbeitet werden und definieren, wie sich ein Traverser im Graph bewegen soll. Eine einzelne Anweisung wird *Schritt* genannt.

Ein Schritt ist eine Funktion f, die eine Menge an Traversern T auf eine andere Menge an Traversern abbildet.

$$f : T \to T \tag{3.2}$$

Sei als Beispiel folgende Traversal gegeben:

```
g.V().out("cfg").values("op_code")
```

Um die Funktionsweise dieser Traversal zu verstehen, soll sie schrittweise evaluiert werden. Sei V_i dabei der Knoten mit dem Index i:

1. `g.V()`: `g` bildet das Startobjekt und den ersten Traverser (ohne Ort). `V()` dupliziert diesen Traverser für jeden Knoten im Graphen und liefert als Ausgabe alle Knoten. Dies ist in diesem Fall die Menge $\{V_1, V_2, V_3, V_4, V_5, V_6, V_7, V_8\}$. Alternativ kann dem Schritt `V()` auch eine Menge von Zahlen als Argument mitgegeben werden. In diesem Fall werden nur Knoten mit der entsprechenden ID zurückgegeben.

2. `out("cfg")` lässt alle Traverser entlang aller ausgehenden Kanten mit der Markierung „cfg“ wandern. Gibt es mehrere dieser Kanten, wird der Traverser dupliziert. Falls keine Kante vorliegt, wird der Traverser verworfen. Die Ausgabe ist also die Menge $\{V_1, V_2, V_3, V_5, V_7\}$.

3. `values("op_code")` lässt alle Traverser auf das Wertepaar mit dem Schlüssel „op_code“ wandern. Gibt es das Paar nicht, wird der Traverser verworfen. Die Ausgabe lautet also $\{\texttt{br}, \texttt{br}\}$.

Traverser

Die Menge der Traverser ist definiert als (die Untermengen U_1 und U_2 dienen nur dem Zweck der Referenzierung):

$$T \subseteq ((U_1 \subseteq U) \times \Psi \times W \times (U_2 \subseteq U)) \tag{3.3}$$

Nachfolgende Erklärung bezieht sich auf einen Traverser:
$u_1 \in U_1$: Die Position des Traverser. Dies kann ein Ort im Graphen sein, aber auch das Ergebnis einer Rechnung oder ein Attribut.
$\psi \in \Psi$: Die Menge der von dem Traverser noch zu bearbeitenden Traversalschritte.
$w \in W$: Der zurückgelegte Weg, also eine Liste aller Orte, an denen der Traverser im Vorhinein war.
$u_2 \in U_2$: Der Beutel[11] des Traversers. Der Beutel ist ein Speicherplatz für ein beliebiges Objekt, das an den jeweiligen Traverser gebunden ist.

3.2.2 Traversal-Schritte

Um eine Traversal zu definieren, kann aus 95 vordefinierten Schritten (die meistens verschiedenartig überladen sind, sodass insgesamt 165 Möglichkeiten existieren) gewählt werden [39].

[11] in der (englischsprachigen) Projektdokumentation: „sack“.

Diese Schritte lassen sich in ihren Eigenschaften auf fünf Grundschritte und Schrittmodulatoren reduzieren. Zu beachten sind außerdem Barrieren und Prädikate. Alle Teile werden im Folgenden gesondert vorgestellt.

Grundschritte

Die fünf Grundschritte sind wie folgt definiert. T ist die Menge der Traverser.

filter Ein Filterschritt nimmt als Eingabe eine Menge an Traversern und eine Bedingung und verwirft dann anhand der Bedingung den jeweiligen Traverser oder lässt ihn passieren. Der Filter ändert den Traverser also nicht.

$$\text{filter}(T) \subseteq T$$

sideEffect Ein Schritt mit Nebenwirkung ändert den Traverser nicht und lässt ihn auf jeden Fall passieren, kann aber anhand des Traversers Daten in einer global übergebenen Datenstruktur N ändern. Sei s eine Funktion, die genau das macht.

$$\text{sideEffect}(T) = T \text{ und } s : T \times N \to N$$

branch Eine Verzweigung nimmt eine Menge an Traversern an und setzt den einzelnen Traverser abhängig vom Zustand auf eine der in der Verzweigung definierten Untertraversals. Er ändert damit nicht die Position, sondern fügt den Traversalschritten ψ des Traversers Schritte hinzu[12].

$$\text{branch}\left(T_{\psi_1}\right) = T_{\psi_2}$$

map Eine Abbildung nimmt eine Menge von Traversern und bildet sie auf eine gleich große Menge in einem anderen Zustand ab. Ein branch-Schritt ist ein Spezialfall eines map-Schrittes.

$$|\text{map}(T_1)| = |T_2|$$

flatMap Ein flatMap-Schritt nimmt eine Menge von Traversern und bildet sie auf eine beliebige andere Menge ab. Dies ist die generische Definition für einen Schritt und trifft auf alle Schritte zu.

$$\text{flatMap}(T_1) = T_2$$

[12] Die Notation T_{ψ_i} soll genau das ausdrücken.

Schrittmodulatoren

Einige Schritte in Gremlin können neben ihren Argumenten noch weitere Spezifizierungen erhalten. Diese werden als sogenannte Schrittmodulatoren übergeben, die in ihrer Syntax vollwertigen Schritten gleichen.

Ein Beispiel ist der `path`-Schritt. Dieser gibt den bisherigen Weg des Traversers aus. Sei folgende Anfrage gegeben:

```
g.V(7).in().in().path()
```

Ausgeführt auf Graph 3.1 ergibt diese das Ergebnis: $\{(V_7, V_6, V_2), (V_7, V_2, V_1)\}$.

Will man nicht die Indizes der Knoten, sondern andere Ausprägungen abrufen, kann man einen Schrittmodulator einsetzen:

```
g.V(7).in().in().path().by(_label)
```

Diese Anfrage gibt jeweils das Label der Knoten zurück und hat die Ausgabe: $\{(\text{Instr}, \text{Instr}, \text{BBlock}), (\text{Instr}, \text{BBlock}, \text{BBlock})\}$.

Je nach Schritt ist die mögliche Anzahl an Modulatoren begrenzt oder unbegrenzt. Bei einer unbegrenzten Anzahl werden die vorhandenen Modulatoren im Rundlauf-Verfahren („Round Robin“) auf den Schritt angewendet.

Barrieren

Gremlin folgt dem Paradigma der „lazy evaluation“, wertet also nur das aus, was nötig ist. Dies führt zu einer Art Tiefensuche im Graph, bei der ein einzelner Traverser so lange weiterarbeitet, bis er entweder durch das Ende der Traversal terminiert oder durch eine Barriere nicht mehr weiterarbeiten kann.

Barrieren sind Schritte, die nur dann vollständig ausgewertet werden können, wenn sie die Liste aller Traverser bekommen. Ein Beispiel für eine Barriere ist der `count`-Schritt, der die Anzahl der Traverser zählt:

```
g.V(1).out().count()
```

Die Anfrage liefert das Ergebnis $\{4\}$.

Als gewollter Nebeneffekt kann durch Barrieren die Auswertungsreihenfolge der Schritte geändert werden. Aus diesem Grund existiert der unbedingte `barrier`-Schritt, der bei wiederholter Einfügung die Auswertung zu einer Art Breitensuche zwingt.

Prädikate

Einige Schritte können als Eingabe Prädikate annehmen. Diese dienen hauptsächlich zur Spezifikation von Bedingungen und werden separat ausgewertet. Prädikate bestehen ausschließlich aus ineinandergeschachtelten Funktionen.

Ein Beispiel für eine Anwendung von Prädikaten ist der `is`-Schritt:

```
g.V().values("bit_width").is(not(gt(32)))
g.V().values("bit_width").is(lte(32))
```

Die erste Anfrage liefert Knoten der Bitbreite[13], die nicht größer als 32 ist. Die zweite Anfrage liefert Knoten, deren Bitbreite kleiner als oder gleich 32 ist. Beide Anfragen sind äquivalent.

Prädikate können als Argument sowohl den Wert, mit dem sie vergleichen sollen, annehmen, als auch eine Zeichenkette, die als Schlüssel für früher berechnete Werte gilt.

Ausgewählte Schritte

Für die folgende Beschreibung der Programmstrukturen und den Beweis der Turingvollständigkeit werden noch einige weitere ausgewählte Einzelschritte benötigt, die hier eingeführt werden sollen:

- `sack(a)`, `sack()`: Ein Zeichen „`a`“ in den Beutel hineinlegen, bzw. wieder herausholen.
- `in()`: Von einem Knoten zum anderen über die eingehende Kante wechseln.
- `choose(a).option(x, b).option(y, c)...`: Falls das Ergebnis der Traversal „`a`“ „`x`“ entspricht, „`b`“ ausführen. Bei einer Entsprechung von „`y`“ wird „`c`“ ausgeführt usw..
- `property(s, w)`: Schreibt den Wert „`w`“ an das Attribut mit dem Schlüssel „`s`“.

3.2.3 Programmstrukturen in Gremlin

Gremlin unterstützt sowohl die deklarative als auch die imperative Programmierung.

[13] Dieses Attribut ist im Beispielgraphen nicht vorhanden, wird aber für dieses Beispiel als gegeben angenommen.

Das Vergleichen von Mustern (z. B. mit Filterschritten) wird deklarativ beschrieben. Die Bewegung der Traverser im Graph erfolgt imperativ.

Hier sollen kurz die Entsprechungen von Standardgebrauchsmustern für imperative Sprachen vorgestellt werden.

Schleifen und Verzweigungen

Der Kontrollfluss in imperativen Sprachen wird hauptsächlich durch Schleifen und Verzweigungen gesteuert. In Gremlin werden dazu der `repeat`- und `choose`-Schritt verwendet.

Verzweigungen sind vergleichsweise einfach umgesetzt:

```
.choose(<condition>, <traversal1>, <traversal2>)
```

Die Auswertung führt `<traversal1>` aus, wenn `<condition>` wahr ist, ansonsten `<traversal2>`. Alternativ kann die – an die „switch-case“-Syntax aus Sprachen der C-Familie erinnernde – `choose().option()`-Syntax benutzt werden, die bereits vorgestellt wurde. Seit als Beispiel die folgende Anfrage gegeben:

```
g.V().choose(has("op_code"), in("hier"), out("cfg"))
```

Diese geht im Fall von einem „Instr“-Knoten (denn nur diese haben ein Attribut „op_code“) die eingehende „hier“-Kante entlang und ansonsten ausgehende „cfg“-Kanten. Auf Graph 3.1 ergibt sich $\{V_1, V_1, V_2, V_2, V_3, V_1, V_2, V_3\}$.

Schleifen werden mit mehreren aufeinanderfolgenden Schritten modelliert:

```
.repeat(<traversal1>).while(<condition>)
.while(<condition>).repeat(<traversal1>)
.repeat(<traversal1>).times(<number>)
```

Die erste Traversal wiederholt `<traversal1>` solange, bis `<condition>` zutrifft, führt aber `<traversal1>` zumindest einmal aus. Bei der zweiten Traversal wird zuerst `<condition>` geprüft, folgt aber ansonsten der gleichen Semantik. Die dritte Traversal wiederholt `<traversal1>` `<times>`-mal.

Jede dieser Kombinationen kann noch mit dem `emit`-Schritt versehen werden, der die Zwischenergebnisse zusätzlich ausgibt.

Zur Verdeutlichung sei die Beispielanfrage gegeben:

```
g.V(1).repeat(out()).until(has("op_code")).emit().path()
```

Diese geht ausgehend von V_1 solange ausgehende Kanten, bis ein Knoten mit einem Attribut „op_code“ erreicht wurde und gibt sämtliche Zwischenschritte in Pfadform aus. Es resultiert die Ergebnismenge $\{(V_1, V_4), (V_1, V_5), (V_1, V_3), (V_1, V_3, V_8)\}$.

Variablen

Es ist oft notwendig, Ergebnisse zwischenzuspeichern, um damit später weiterzuarbeiten. Gremlin unterstützt dazu zwei Methoden.

Zwischenergebnisse können mit einem Etikett versehen werden, das im Pfad des Traversers gespeichert wird. Später in der Traversal kann auf dieses Etikett zurückgegriffen werden. Der Ansatz hat mehrere Nebeneffekte. Zum einen zerstören manche (Barrieren-)Schritte den Pfad und löschen so alle markieren Schritte. Zum anderen greift man mit jedem Traverser einzeln auf das markierte Objekt zurück. Wenn sich der Traverser zwischen Markierung und Rückgriff geklont hat das markierte Objekt wieder hergestellt wird, kommt es zu Dopplungen. Mit dem `as`-Schritt kann ein Etikett gesetzt werden und mit `select()` wird das Objekt wieder hergestellt. Als einfaches Beispiel dient die Anfrage:

```
g.V(2).as("a").out().select("a")
```

Sie hat das Ergebnis $\{V_2, V_2, V_2\}$.

Die andere Methode, Daten zwischenzuspeichern, besteht im Speichern der Daten als Seiteninformationen. Dabei handelt es sich um eine globale Datenstruktur, die unabhängig von der Auswertung vorliegt. Objekte können dort hineinkopiert und auch wieder herausgenommen werden. Nachteile dieses Verfahrens sind neben einem komplexeren Zugriff und so längerer Laufzeit auch die nicht immer gegebene Unterstützung bei diversen Schritten. Eine von mehreren Möglichkeiten, Zwischenergebnisse in die Seiteninformationen zu legen und herauszuholen, besteht im Schritt `aggregate()` und `cap()`. So liefert die Anfrage:

```
g.V(2).aggregate("x").out().cap("x")
```

das Ergebnis $\{V_1, V_2, V_3, V_4, V_5, V_6, V_7, V_8\}$.

3.2.4 Turing-Vollständigkeit

Gremlin ist Turing-vollständig. Der Beweis liegt in der Konstruktion einer 1-Band-Turingmaschine (TM):

$$M = (Q, \Sigma, \Gamma, \delta, q_0, \square, F) \tag{3.4}$$

Q: Zustandsmenge
Σ: Eingabealphabet
Γ: Bandalphabet
$\delta : (Q \setminus F) \times \Gamma \to Q \times \Gamma \times \{L, N, R\}$: Übergangsfunktion
$q_0 \in Q$: Anfangszustand
$\square \in \Sigma \setminus \Gamma$: leeres Feld
$F \subseteq Q$: Menge der Endzustände

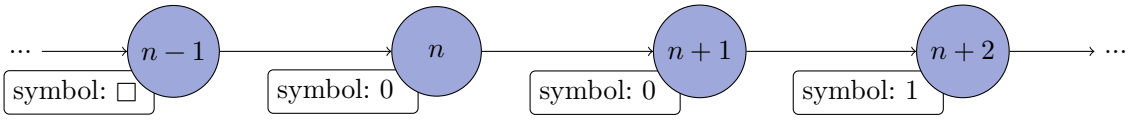


Abbildung 3.2 Graphrepräsentation des Eingabebandes der Turingmaschine

Das Band wird als Graph von aufeinander folgenden Knoten mit dem Attribut „`symbol`“ für den Inhalt an der jeweiligen Stelle repräsentiert (siehe Abbildung 3.2). Der aktuelle Zustand der TM ist im Beutel gespeichert. Eine Kopfbewegung wird auf `out()` und `in()` bzw. keinen Schritt abgebildet. Die Startposition des Kopfes ist der Knoten mit der Nummer 1.

Sei damit die TM M in Gremlin wie in Codeblock 3.1 definiert.

```
1  g.V(1).
2    sack(q_0).
3    repeat(choose(values("symbol")).
4      option(s_1 ∈ Γ, choose(sack()).
5        option(q_{i_1},
6          property("symbol",s_2 ∈ Γ).out().sack(q_{i_2})).
7        option(q_{i_3},
8          property("symbol",s_3 ∈ Γ).out().sack(q_{i_4})).
9        ...
10     option(s_4 ∈ Γ, choose(sack()).
11       option(q_{i_5},
12         property("symbol",s_5 ∈ Γ).out().sack(q_{i_6})).
13       ...
14     ...
15   until(sack().or(is(f_1 ∈ F), is(f_2 ∈ F), ...)
```

Codeblock 3.1 Simulation einer Turingmaschine

Die hervorgehobenen Zeilen repräsentieren eine Auswertung von δ:

$$\delta(q_{i_1}, s_1) = (q_{i_2}, s_2, R)$$

Die Traversal beginnt also im Startzustand (Zeile 2) an der Startposition des Eingabebandes (Zeile 1), liest dieses (Zeile 3) und den aktuellen Zustand (Zeile 4) und schreibt anhand dessen (Zeile 5) ein neues Zeichen auf das Band und einen neuen Zustand in den Beutel und bewegt sich im Graph (Zeile 6). Dies wird solange wiederholt, bis im Beutel ein Endzustand vorliegt (Zeile 15).

3.3 libgremlin

Um Gremlin in dieser Arbeit nutzen zu können, war eine Neuimplementierung in C++ nötig.

Diese wurde dabei als Bibliothek („*libgremlin*“) realisiert und unterscheidet sich dadurch fundamental von Tinkerpop. Eine Folge dieses Designs ist der vollständige Verzicht auf eine Datenbank. Der Graph ist als Schnittstelle in Form von abstrakten Klassen definiert. libgremlin macht dabei sehr wenig Einschränkungen bezüglich der Implementierung des Graphen, stellt aber als Referenz und zu Testzwecken eine Implementierung des „Graph Modern“ (ein vordefinierter Graph von Tinkerpop, auf dem fast alle Beispiele und Testfälle definiert sind) zur Verfügung.

Die Gründe für dieses Design lagen in der besseren Möglichkeit, Gremlin direkt in ein Programm zu integrieren, insbesondere in LLVM. Eine Traversal, die mit libgremlin ausgeführt wird, ermöglicht die direkte Verarbeitung der unterliegenden Graphdatenstrukturen. libgremlin ist allerdings nicht auf LLVM beschränkt.

Bei der Implementierung wurde außerdem darauf geachtet, außer der Standardbibliothek keine weiteren Abhängigkeiten zu benutzen. Eine Ausnahme besteht im Parser-Code, der Zeichenketten in Gremlin-Abfragen übersetzt. Dieser wurde mit den Bibliotheken von Clang realisiert, ist aber optional und kann beim Übersetzen deaktiviert werden.

Bei der Implementierung gab es einige Hürden. Hauptsächlich lag dies daran, dass Gremlin keine formale Spezifikation hat. Das exakte Verhalten einzelner Schritte musste darum intensiv an der Referenzimplementierung getestet werden.

Im Folgenden werden einige Probleme aufgezeigt und anschließend die wesentlichen Algorithmen zur Auswertung in der Implementierung vorgestellt.

3.3.1 Probleme

Die Aneinanderkettung der Gremlin-Schritte als Funktionen legt die direkte Implementierung in eben diesen Funktionen nahe, also die Ausführung des Schrittes bei Aufruf der jeweiligen Funktion. Dazu dürfen Funktionen, die von C++ zeitlich vorher ausgewertet werden, nicht von Funktionen abhängen, die später kommen.

Dieser Ansatz ist nicht realisierbar. Es sprechen hauptsächlich zwei Gründe dagegen.

Der erste liegt in der Problematik, Schrittmodulatoren auszuwerten. Schrittmodulatoren werden stets nach dem eigentlichen Schritt spezifiziert und ausgewertet und können zudem in beliebiger Anzahl auftreten. Sie ändern aber trotzdem das Verhalten des Schrittes, den sie modulieren. Eine Auswertung des Schrittes, bevor der Schrittmodulator verarbeitet wurde, ist somit nicht möglich. Es ist aber ebenfalls nicht sicher, ob überhaupt ein Modulator kommt und ob einem Modulator ein weiterer folgt.

Als zweiter Grund macht die Auswertungsreihenfolge von verschachtelten Funktionen eine direkte Auswertung unmöglich. C++ wertet innere Funktionen zuerst aus. Bei Gremlin hängen innere Traversals aber vom Kontext der umgebenen Funktion ab, müssen also nach bzw. von dem äußeren Schritt ausgeführt werden.

Bei einer direkten Auswertung ergeben sich noch weitere Probleme, wie fehlende Optimierungen etc. Eine Lösung ist es, die Implementierung der Schritte darauf zu reduzieren, diese in eine Liste einzutragen. Diese kann dann vorverarbeitet und anschließend ausgewertet werden.

Ein weiteres Problem ist, dass in der Dokumentation von Gremlin an einigen kritischen Stellen nicht definiert ist, wie die Traversal ausgewertet werden soll. Außerdem verhält sich die Referenzimplementierung stellenweise entgegen der Erwartung. Dies soll an einigen Beispielen verdeutlicht werden:

1. Es ist nicht definiert (auch nicht verboten), was bei Schritten mit Infix-Notation passiert, die nicht in einer Untertraversal ausgeführt werden, z. B.

   ```
   g.V().out().has("op_code").and().hasLabel("Instr")
   ```

   In dem Beispiel ist nicht klar, welche beiden Teile vor und nach dem `and`-Schritt zu diesem gehören. libgremlin verbietet darum Infix-Notationen, die nicht in einer Untertraversal ausgeführt werden.

2. Es ist nicht definiert, wie sich der `order`-Schritt auf Attributen verhält, die nicht sortiert werden können. libgremlin schreibt daher für selbst definierte Attributtypen eine Ordnungsrelation vor.

3. Schleifen werden mit einem bis drei Schritten definiert. Bei Anfragen mit mehreren Schleifen ist nicht klar, welche Schritte zu welcher Schleife gehören. Ein Beispiel ist die Traversal:

   ```
   g.V().repeat(out()).until(has("op_code")).repeat(in())
   ```

Da keine Prioritäten über Klammern gesetzt werden können, ist es nicht möglich zu sagen, ob der `until`-Schritt der ersten oder zweiten Schleife zugeordnet ist. libgremlin löst dieses Problem, indem doppeldeutige Schritte der Schleife zugeordnet werden, die in der Auswertung früher kommt und mit diesem Schritt noch wohldefiniert ist[14].

4. Schritte agieren auf globaler Ebene. Damit ist gemeint, dass Schritte innerhalb von Schleifen bei mehrfachem Durchlauf ihren Zustand „zwischenspeichern“ und damit implizit Seiteneffekte haben. Dies wird z. B. durch die folgenden beiden Anfragen verdeutlicht, die entgegen der Erwartung verschiedene Ergebnisse liefern:

   ```
   g.V().repeat(both().dedup()).times(2)
   g.V().both().dedup().both().dedup()
   ```

   libgremlin gestattet Kopien von Schritten, trägt aber bei kritischen Schritten den Zustand mit in die globale Seiteneffekt-Datenstruktur ein.

3.3.2 Algorithmik

Der Kern der Implementierung teilt sich in zwei Teile. Die Auswertung des einzelnen Traversers und die globale Auswertung inklusive Barrieren. Eine Traversal, die mit libgremlin ausgewertet wird, durchläuft hauptsächlich die Funktion `core::run()`, die die globale Auswertung vornimmt und dabei u. a. Traverser erzeugt, die dann separat ausgewertet werden. Ein Vorteil dieser Implementierung liegt darin, dass die Traverser mit einer geeignet synchronisierten Datenstruktur parallel ausgewertet werden können. Es wurde darauf geachtet, sämtliche Zugriffe entsprechend zu gestalten, die Auswertung läuft aber bislang nicht parallel[15].

Im Folgenden sollen beide Teile kurz vorgestellt werden.

Core

Der Core teilt sich hauptsächlich in zwei Phasen auf: die Auswertung der einzelnen Traverser und die anschließende Auswertung der Barrieren, die solange wiederholt werden, bis die Traversal abgearbeitet ist.

[14] Damit ist gemeint, dass sie nicht bereits durch einen weiteren Schritt, z. B. `times()` näher spezifiziert ist.

[15] Ungeklärt ist in diesem Zusammenhang außerdem die Auswertung von Seiteneffekten.

Die Auswertung der Traverser wird dabei zu einem Großteil vom Traverser selbst vorgenommen. Der Core ist nur für die Verwaltung und das Starten des Traversers zuständig.

Einzelne Traverser werden nach dem Ansatz der Tiefensuche in einem Kellerspeicher verwaltet. Jeder Schritt hat die Möglichkeit, weitere Traverser zu erzeugen und auf den Keller zu legen.

Zuerst wird ein Traverser vom Keller genommen und gestartet (Zeile 4 und 6). Falls er nach erfolgreicher Auswertung am Ende der Traversal ist, wird das Ergebnis seiner Berechnung in die Menge der Endergebnisse aufgenommen (Zeile 11) und ansonsten den Barrierenschritten zugeordnet (Zeile 9). Dies wird solange wiederholt, bis der Keller leer ist (Codeblock 3.2).

```
1  stack<Traverser> travs = make_stack(start_nodes);
2  ...
3  while(!travs.empty()) {
4      Traverser t = travs.pop();
5
6      result = t.run(travs)
7
8      if (t.has_remaining_steps()) {
9          barrier_steps.insert(t, result)
10     } else {
11         end_result.push_back(result);
12     }
13 }
```

Codeblock 3.2 Auswertung der Traverser

Im zweiten Schritt werden alle Traverser, die vor einer Barriere warten, ausgewertet. Dazu wird der Barrierenschritt auf der Liste aller Traverser ausgewertet (Zeile 3) und anschließend für jedes Ergebnis ein neuer Traverser mit den restlichen Schritten wieder auf den Keller gelegt (Zeile 6, Codeblock 3.3).

```
1  for (barrier_step in barrier_steps) {
2      // rem = remaining_traversal
3      result_nodes, rem = barrier_step.process()
4
5      for (node in result_nodes) {
6          travs.push(make_traverser(rem, node))
7      }
8  }
```

Codeblock 3.3 Auswertung der Barrieren

Beide Teile werden aufeinanderfolgend wiederholt, bis der Keller leer ist. Abschließend wird die Liste der Resultate zurückgegeben.

Traverser

Die Implementierung der Auswertung eines Traversers ist vereinfacht in Codeblock 3.4 dargestellt. Der Traverser arbeitet im Wesentlichen die Schritte der Traversal ab (Zeile 3), bis er vor einer Barriere steht (Zeile 5) oder am Ende der Traversal ist (Zeile 16).

Ein einzelner Schritt kann dabei sowohl neue Orte im Graph als auch neue Traversalschritte liefern (Zeile 8). Letzteres ist für die Implementierung von `branch`-Schritten notwendig, da der anhand des aktuellen Zustands gewählte Zweig oder – im Fall von Schleifen – die Schleife selbst als zusätzliche Traversal zurückgegeben wird.

Das erste Ergebnis wird vom Traverser selbst weiterverarbeitet (Zeile 13 und 14). Für alle anderen Ergebnisse wird ein Klon des aktuellen Traversers erstellt und auf den Keller gelegt (Zeile 11).

```
1  Traverser::run(stack<Traverser> travs) {
2      current_node = node;
3      for step in traversal {
4          if step.is_barrier() {
5              return current_node, remaining_steps
6          }
7          if current_node.invalid() { continue }
8          new_nodes, new_steps = step.process(current_node)
9
10         for new_node in new_nodes[1:] {
11             travs.push(this.clone(new_steps, remaining_steps))
12         }
13         current_node = new_nodes[0];
14         this.traversal.insert(new_steps)
15     }
16     return current_node, null
17 }
```

Codeblock 3.4 Auswertung eines Traversers

Kapitel 4
Gremlin-Pass

Der Kernbestandteil dieser Arbeit war es, die Datenstrukturen, die in LLVM auftreten, mit Gremlin durchsuchbar zu machen.

Dazu muss ein Graph, der von libgremlin verstanden wird, aus der IR mithilfe der LLVM API extrahiert werden. LLVM kennt dazu das Konzept von eigenen Passes. Diese können als dynamische Bibliothek implementiert werden, die dann während der Laufzeit von LLVM geladen und aufgerufen wird.

Der hier implementierte Pass *Gremlin-Pass* stellt eine Unterklasse des ModulePass dar, um eine möglichst umfangreiche Auswertung z. B. über Funktionsgrenzen hinweg zu ermöglichen. Er konstruiert dazu zum einen den *LLVM-Graphen*, eine Datenstruktur, die dynamisch aus dem LLVM-Modul einen für die libgremlin verständlichen Graphen erzeugt. Zum anderen definiert er die Anfragen, die auf diese Datenstruktur gestellt werden.

Der LLVM-Graph soll im Folgenden vorgestellt werden, die Anfragen sind Teil des nächsten Kapitels.

4.1 Abgebildeter Graph

Die Datenstrukturen und Methoden, die LLVM mit seiner API bereitstellt, besitzen bereits viele Eigenschaften eines Graphen, die übernommen werden können.

Der LLVM-Graph benutzt diese Eigenschaften, um daraus explizite Objekte zu erzeugen, die von libgremlin verstanden werden. Dabei wurde darauf geachtet, Referenzen oder Zeiger der eigentlichen LLVM-Objekte zu behalten, sodass darauf jederzeit zugegriffen werden kann.

Bestandteile des LLVM-Graphen sind die Graph-Klasse, die zur Verwaltung aller Knoten und Kanten dient und diese bei Bedarf neu konstruiert und anschließend in einem geeigneten Cache vorhält, sowie Klassen für Knoten und Kanten. Knoten sind im LLVM-Graph attributiert, Kanten besitzen nur ein Etikett zur Kennzeichnung.

4.1.1 Knoten

Vom LLVM-Graphen werden drei Knotentypen bereitgestellt. Diese referenzieren das Modul, Schleifen, die im Code vorkommen, und beliebige andere Objekte der Klasse `Value`.

Da in LLVM viele Entitäten von der Klasse `Value` erben, dient der letzte Knotentyp dazu, Funktionen, einfache Blöcke, Instruktionen und Argumente zu speichern. Je nach Typ werden dem Knoten weiterhin verschiedene Attribute zugewiesen. In Tabelle 4.1 ist eine Auflistung aller Attribute zu sehen.

Durch eine Eigenheit von LLVM bei der Analyse von Schleifen[16], besitzt ein Schleifen-Knoten kein LLVM-Objekt mehr, sondert speichert nur eine Liste der einfachen Blöcke, die zur Schleife gehören.

LLVM-Klasse[17]	**Attributname**	**Kommentar**
Module	`filename`	Dateiname der Quelldatei.
Value	`type`	Typ des Wertes, z. B. `i32` oder `void*(i32)`
	`name`	Name des Wertes, z. B. oder `@malloc`
BasicBlock	`is_loop_header`	Vorhanden, wenn der einfache Block Kopf einer Schleife ist.
Instruction	`code`	Lesbare Repräsentation der Instruktion.
	`op_code`	Opcode, Beispiel: `add`, `call`
	`directory`	Verzeichnisname[18]
	`filename`	Dateiname[18]
	`line`	Zeile[18]
	`column`	Spalte[18]
	`bit_with`	Speicherbreite (nur bei ganzen Zahlen)
CmpInstr	`signedness`	Art des Vergleichs bzgl. Vorzeichen, z. B. `signed`
	`comparison`	Vorhanden, wenn ein Vergleich vorliegt.
BinaryOperator	`signedness`	siehe „signedness“ bei CmpInstr.
CallInstr	`function_name`	Name der aufgerufenen Funktion oder „indirect call“ sonst

Tabelle 4.1 Übersicht aller Attribute für verschiedene Knotentypen, Schleifen haben keine zusätzlichen Attribute

[16] Schleifen werden erst mit einem speziellen Pass erkannt, und bei jedem weiteren Durchlauf des Pass dekonstruiert, haben also eine kürzere Lebensdauer als der LLVM-Graph.

[17] Die Klassen sind absteigend nach Klassenhierarchie sortiert. Außer „Module“ erben alle Klassen von „Value“.

[18] Existiert nur bei vorhandenen Debug-Informationen.

4.1.2 Kanten

Der LLVM-Graph kennt sieben verschiedene Kantentypen, die durch zwei Klassen realisiert werden. Gremlin arbeitet auf gerichteten Graphen, kann aber trotzdem in beide Richtungen gehen. Im LLVM-Graph wurde dieses Verhalten dadurch realisiert, dass Kanten zwar eine Richtung besitzen, aber von jedem Knoten in jede Richtung erstellt werden können.

Ein Kantentyp ist die `HIERARCHY_EDGE`. Sie stellt eine gerichtete Verknüpfung zwischen der LLVM-Hierarchie dar. Abbildung 4.1 illustriert diesen Kantentypen.

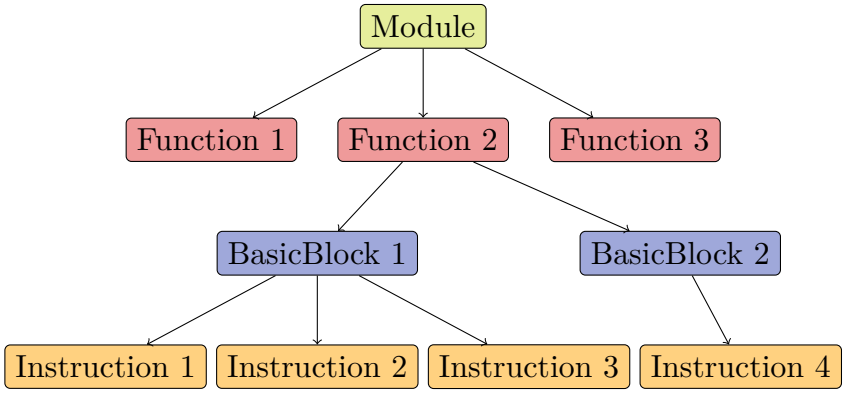


Abbildung 4.1 `HIERARCHIE_EDGE`s, Abbildung der LLVM-Hierarchie

Ein weiterer Kantentyp ist die `CFG_EDGE` und repräsentiert den Kontrollfluss. Kontrollflusskanten kommen sowohl zwischen Instruktionen als auch zwischen einfachen Blöcken vor.

Alle Instruktionen innerhalb eines einfachen Blocks (außer die erste und letzte) sind dabei mit ihren jeweiligen Vorgänger- und Nachfolgeinstruktionen verknüpft. Die ersten und letzten Instruktionen sind mit den jeweiligen (auch mehreren) Instruktionen in den vorhergehenden und nachfolgenden einfachen Blockes verknüpft.

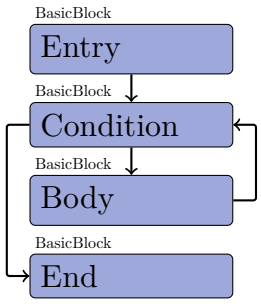
Einfache Blöcke sind untereinander anhand ihres Kontrollflusses verknüpft. Abbildung 4.2 zeigt ein Beispiel für Kontrollflusskanten.

Weiterhin gibt den Kantentyp `USE_EDGE`. Diese Kanten verknüpfen über die in Kapitel 2.3.4 vorgestellten Use-Objekte alle Operatoren mit ihren Operanden. `USE_EDGE`s sind nicht auf Instruktionen beschränkt, sondern können auch verschiedene Objekttypen miteinander verknüpfen. In Abbildung 4.3 sind verschiedene `USE_EDGE`s dargestellt.

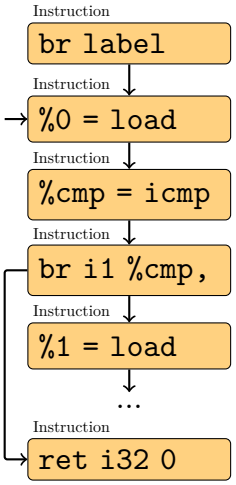
Etwas spezieller sind Kanten des Typs `LOOP_EDGE`. Mit diesen Kanten werden Schleifen gekennzeichnet. Die Informationen dazu werden durch einen separaten Pass bereitgestellt, der für jede Funktion aufgerufen wird. Bedingt durch das

```
1  ; ModuleID = '<stdin>'
2  source_filename = "main.c"
3  ...
4
5  ; Function Attrs: noinline nounwind uwtable
6  define i32 @main() #0 {
7  entry:
8    %retval = alloca i32, align 4
9    %a = alloca i32, align 4
10   store i32 0, i32* %retval, align 4
11   store i32 2, i32* %a, align 4
12   br label %while.cond
13
14 while.cond:
15   %0 = load i32, i32* %a, align 4
16   %cmp = icmp sgt i32 %0, 0
17   br i1 %cmp, label %while.body, label %while.end
18
19 while.body:
20   %1 = load i32, i32* %a, align 4
21   %dec = add nsw i32 %1, -1
22   store i32 %dec, i32* %a, align 4
23   br label %while.cond
24
25 while.end:
26   ret i32 0
27 }
```

(a)



(b)

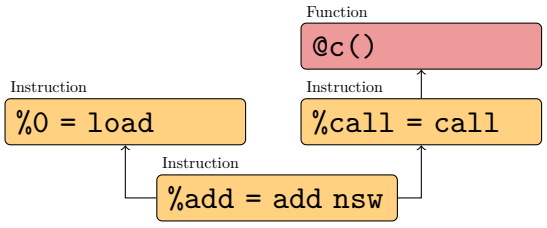


(c)

Abbildung 4.2 `CFG_EDGE`s, am Beispiel einer While-Schleife (a) zwischen einfachen Blöcken (b) und Instruktionen (c)

```
1  define i32 @c() #0 {
2    ...
3  }
4
5  define i32 @main() #0 {
6    ...
7    %0 = load i32, i32* %a, align 4
8    %call = call i32 @c()
9    %add = add nsw i32 %0, %call
10 }
```

(a)



(b)

Abbildung 4.3 `USE_EDGE`s am Beispiel eines Funktionsaufrufes (a) zwischen Instruktionen und Funktionen (b)

Design von LLVM wird diese Information für jeden Durchlauf überschrieben und muss so vom Gremlin-Pass zwischengespeichert werden. Abbildung 4.4 zeigt die Verknüpfung mit Kanten des Typs `LOOP_EDGE` auf dem Quellcodebeispiel in Abbildung 4.2.

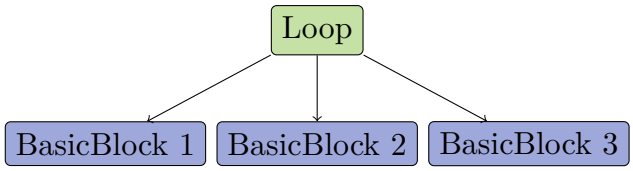


Abbildung 4.4 `LOOP_EDGE`s

Die restlichen Kantentypen verknüpfen Funktionen. Mit der `ARGUMENT_EDGE` wird eine Funktion mit ihren Argumenten verknüpft[19]. Eine `CALLED_FUNCTION_EDGE` verbindet eine Aufrufinstruktion mit der aufgerufenen Funktion[20]. Abschließend verbindet eine `USED_AS_ARGUMENT_EDGE` eine Instruktion, die im späteren Verlauf als Argument einer Funktion in einem Funktionsaufruf genutzt wird, mit genau diesem Argument und ermöglicht so ein Verfolgen der Benutzung über Funktionsgrenzen hinweg. Abbildung 4.5 zeigt einen Ausschnitt des LLVM-Graphen mit diesen Kanten.

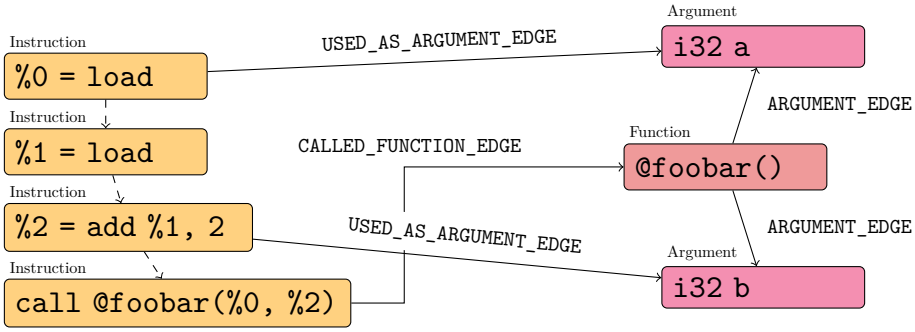


Abbildung 4.5 Kanten, die Funktionen und Argumente verknüpfen. Die gestrichelten Kanten bezeichnen `CFG_EDGE`s.

[19] Variable Argumente werden durch diese Kante nicht abgebildet.
[20] Die Kante existiert nicht bei indirekten Funktionsaufrufen.

Kapitel 5
Bug-Klassifizierung

Mit dem Gremlin-Pass kann innerhalb der IR nach beliebigen Mustern gesucht werden. Sicherheitslücken zeichnen sich zumeist durch spezielle Eintrittspunkte aus, die genau solche Muster darstellen.

Alle Eintrittspunkte bzw. Muster zu finden, würde den Rahmen dieser Arbeit sprengen. Aus diesem Grund wurde die Evaluierung und auch der Entwurf auf Sicherheitslücken ausgerichtet, die Ganzzahlen betreffen und vor allem beim Wechsel von 32 Bit auf 64 Bit entstehen und im Folgenden vorgestellt werden sollen.

Zuerst wird eine Einführung der Problematik gegeben, danach eine Klassifizierung der Schwachstellen vorgenommen und abschließend die Lücken im Detail vorgestellt.

5.1 Ganze Zahlen in C und C++

ANSI C definiert vier Datentypen für vorzeichenbehaftete Ganzzahlen (Integer): `signed char`, `short int`, `int int`, `long int` und (auf das zweite `int` wird im Weiteren verzichtet)[21] Für jeden dieser Datentypen gibt es ein vorzeichenloses Pendant, das mit dem Schlüsselwort `unsigned` markiert wird [40].

Der Standard schreibt die genaue Größe der Datentypen nicht vor. Es werden allerdings Regeln definiert. Sei dazu $S(i)$ die Menge an Speicher, die von einem Typen i beansprucht wird.

- Der Typ `char` ist mindestens 1 Byte, `short` und `int` mindestens 2 Byte, ein `long` mindestens 4 Byte groß.
- Für den Typen `int` ist weiterhin definiert: ‚A "plain" int object has the natural size suggested by the architecture of the execution environment.‘ (Übersetzt etwa: Ein einfaches Ganzzahl-Objekt hat die natürliche Größe, die durch die Architektur der Ausführungsumgebung naheliegt.) [40]
- $S(\texttt{signed char}) \leq S(\texttt{short}) \leq S(\texttt{int}) \leq S(\texttt{long})$
- Vorzeichenbehaftete und vorzeichenlose Typen haben die gleiche Größe.

[21] Es gibt noch einen weiteren Ganzzahl-Datentypen `long long`. Dieser wird aber erst mit C99 definiert und hier daher vernachlässigt.

Außerdem existiert `size_t` als vorzeichenloser Ganzzahltyp, dessen Größe dadurch definiert ist, alle Ergebnisse des `sizeof`-Operators fassen zu können. Eine weitere Größe ist die Speicherbreite von Zeigern (`sizeof(void*)`), die optional explizit mit `uintptr_t` vorliegen kann.

In der Praxis unterscheiden sich diese Größen von Architektur zu Architektur. Tabelle 5.1 listet die verschiedenen Speicherbreiten für 32 und 64 Bit.

	32 Bit		64 Bit		
Datentyp	LP32[22]	ILP32[23]	LP64[24]	ILP64	LLP64[25]
`char`	8	8	8	8	8
`short`	16	16	16	16	16
`int`	16	32	32	64	32
`long`	32	32	64	64	32
`pointer`	32	32	64	64	64
`size_t`	32	32	64	64	64

Tabelle 5.1 Speicherbreiten der C-Datentypen für 32 und 64 Bit [41]

C++ übernimmt die Datentypen von C, sodass dort die gleichen Aussagen gelten.

5.2 Integer-Schwachstellen

Der limitierte Wertebereich von Ganzzahlen kann in der Praxis zu Fehlern führen, die zu Schwachstellen werden können. Diese Fehler entstehen entweder, wenn eine bestehende Zahl auf einen anderen Wertebereich konvertiert wird (Vorzeichenfehler und Truncation) oder das Ergebnis einer Operation nicht in den dafür vorgesehenen Wertebereich passt (Unter- und Überlauf).

Im Folgenden werden die einzelnen Schwachstellen im Detail beschrieben.

5.2.1 Vorzeichenfehler

Ein Vorzeichenfehler tritt dann auf, wenn eine Zahl, die eigentlich vorzeichenlos sein sollte, mit Vorzeichen interpretiert wird oder umgekehrt. Dies resultiert in einer Verschiebung des Wertebereichs.

[22] verwendet u. a. von Apple Macintosh.
[23] verwendet u. a. von Microsoft Windows und Linux.
[24] verwendet u. a. von macOS und Linux.
[25] verwendet u. a. von Microsoft Windows.

Codeblock 5.1 zeigt ein Beispiel. `malloc` erwartet eine Größe vom Typ `size_t`, was zu einer Uminterpretation der Integer `buf_size` als vorzeichenlose Zahl führt. Ein negativer Wert von `buf_size` wird von dem Vergleich nicht abgefangen, resultiert aber in großen Mengen Speicher, die alloziert werden.

```
1  int buf_size = attacker_controlled();
2  if (buf_size > 50)
3      return;  // buffer too big
4  char* buf = malloc(buf_size);
```

Codeblock 5.1 Beispiel für einen Vorzeichenfehler

5.2.2 Truncation

Eine Verkleinerung der Speicherbreite und damit des Wertebereichs einer ganzen Zahl durch Abschneiden der vorderen Bits nennt man *Truncation*[26].

In Codeblock 5.2 ist ein Beispiel für eine Schwachstelle auf Basis einer Truncation. Unter der Annahme, dass `size_t`, der Rückgabewert von `strlen()`, größer als `short` ist, gibt es in Zeile 2 eine Truncation. Diese führt dann in Zeile 3 dazu, dass für die Operation in Zeile 4 zu wenig Speicher alloziert wird.

```
1  char* a = attacker_controlled();
2  size_t len = strlen(a);
3  unsigned short other_len = len;
4  char* other = malloc(other_len);
5  memcpy(other, a, len);
```

Codeblock 5.2 Beispiel für eine Truncation

5.2.3 Unter- und Überlauf

Ein Unter- oder Überlauf tritt auf, wenn eine Operation für Ganzzahlen ein Ergebnis liefert, das größer als die maximal im Wertebereich der Ganzzahl darstellbare Zahl ist (Überlauf) oder kleiner als die minimal darstellbare Zahl (Unterlauf).

Codeblock 5.3 zeigt ein Beispiel. Abhängig vom Wert von `CONST` resultiert die Addition in einer Zahl, die größer ist, als das von `malloc` verlangte `size_t`.

[26] Sei b die größte darstellbare Zahl des kleineren Wertebereichs und % der Modulo-Operator. Dann ist die Operation Truncation T_b definiert als: $T_b(x) = x \,\%\, (b+1)$.

```
1 size_t a = attacker_controlled();
2 char* buf = malloc(a + CONST);
```

Codeblock 5.3 Beispiel für einen Überlauf

5.3 Probleme des Architekturwechsels

Bei einem Wechsel von 32 auf 64 Bit[27] können durch die Änderung der Speicherbreiten mehrere der oben genannten Probleme auf subtile Art und Weise auftreten. Der Hauptgrund dafür ist die beliebte Verwendung von `int` als Typ für Größen und Zeigeradressen. Allerdings sind bei 64 Bit im Gegensatz zu 32 Bit die Breite von `uintptr_t`, `size_t` und `int` verschieden.

Man kann daher Stellen identifizieren, die speziell durch einen Wechsel von 32 auf 64 Bit auftreten.

- Truncations treten vor allem zusätzlich auf, wenn `size_t` oder `long` dem Typ `int` zugewiesen wird. Die Codezeile „`unsigned int a = strlen(...);`“ ist ein typisches Beispiel.
- Überläufe können zusätzlich auftreten, wenn zwei bislang gleich breite Variablen vom Typ `int` und `long` zusammenhängend verarbeitet werden, z. B. als Zählvariablen in einer Schleife.

Eine detaillierte Beschreibung an neu auftretenden Problemen ist in [42] zu finden.

In dieser Arbeit wird speziell nach solchen Stellen, aber auch nach dem allgemeinen Auftreten von Integer-Schwachstellen gesucht.

5.4 Suchmuster

Im Folgenden sollen die Suchmuster vorgestellt werden, die auf die oben genannten Probleme zugeschnitten sind. Die Anfragen wurden alle für ein 64 Bit Datenmodell entworfen. In den nachfolgenden Abschnitten werden zuerst alle Anfragen inhaltlich vorgestellt und anschließend ihre Implementierung gezeigt.

5.4.1 Vorzeichenfehler

Durch die Suche auf der IR ist die Information über das Vorzeichen nicht mehr an den Wert gekoppelt. Aus diesem Grund wird in den Anfragen nach Operationen gesucht, die an das Vorzeichen gekoppelt sind.

[27] Damit ist das LP64 oder LLP64 Datenmodell gemeint.

Daraus ergeben sich die Anfragen:

Q.S.1 Suche nach allen Anweisungen „`sext`“, bei denen das Resultat in einer Funktion verwendet wird, die eine vorzeichenlose Eingabe erwartet.

Q.S.2 Suche nach allen Vergleichen, die vorzeichenbehaftet vergleichen, bei denen das Resultat aber in einer Funktion verwendet wird, die eine vorzeichenlose Eingabe erwartet.

Diese Anfragen sind vor allem darauf zugeschnitten, fehlerhafte Verwendungen von `int` zu erkennen, in denen `size_t` erwartet wird.

Ein weitere potentielle Schwachstelle ist die fehlerhafte Verwendung der gleichen Zahl in vorzeichenbehafteten und vorzeichenlosen Kontexten. Darum gibt es eine weitere Anfrage:

Q.S.3 Suche nach allen Werten, die sowohl vorzeichenbehaftet, als auch vorzeichenlos erweitert werden.

Dieser Fall tritt z. B. bei einer Variable des Datentyps `int` auf, die erst explizit gecastet wird (vorzeichenlose Erweiterung), danach aber direkt in einem Aufruf verwendet wird, der `size_t` erwartet (vorzeichenbehaftete Erweiterung).

5.4.2 Truncation

Eine Verkleinerung des Wertebereichs ist eine explizite Instruktion in der IR („`trunc`“). Somit ist die Suche nach Truncations trivial, resultiert aber auch in einer hohen Anzahl an falschen Treffern.

Darum sind die Anfragen eingeschränkt worden, um die oben beschriebenen Fälle abzudecken. Um Stellen zu finden, die speziell bei 64 Bit zu einem Problem werden, wurde folgende Anfrage formuliert:

Q.T.1 Finde alle Truncations, bei denen das Ergebnis in einem vorzeichenbehafteten Kontext verwendet wird.

Diese Anfrage findet u. a. alle Stellen, bei denen eine `size_t` fälschlicherweise einer `int` zugewiesen wird.

Stellen, die auch allgemein zu Problemen führen, wie in Codeblock 5.2 beschrieben, werden durch die zweite Anfrage gefunden:

Q.T.2 Finde alle Truncations, bei denen sowohl die Eingabe als auch das Resultat der Truncation zusammen in einem Funktionsaufruf verwendet werden.

5.4.3 Unter- und Überlauf

Unter- und Überläufe sind bei der statischen Analyse im Allgemeinen schwer zu finden, da sie oft stark von den Eingabeparametern abhängen. Trotzdem gibt es spezielle Muster, die oft zu Unter- und Überläufen führen.

Die erste Stelle, die modelliert wurde, sind direkte Additionen (bzw. Subtraktionen) in einer Speicherallokation. Die Gremlin-Anfrage **Q.O.1** findet genau diese Stellen.

Die zweite Stelle sind Schleifen, deren Zählvariable über einen größeren Bereich zählt als Variablen im Rumpf der Schleife verarbeiten können, wie bei den Migrationsproblemen erwähnt. Anfrage **Q.O.2** findet solche Stellen. Die Suche nach der Zählvariable ist dabei nicht trivial, da die IR zwar Schleifen erkennt, diese aber nur mit einfachen Blöcken repräsentiert. Die Anfrage geht daher zusätzlich von der Annahme aus, dass die Zählvariable in einem Vergleich im Schleifenkopf verwendet wird.

Tabelle 5.2 zeigt eine Übersicht über alle Anfragen und deren Kategorien. Q.T.1 gehört dabei zusätzlich zu den Vorzeichenfehlern, da die Anfrage Stellen findet, bei denen der Wertebereich nicht nur verkleinert, sondern auch verschoben wird.

Anfrage	Vorzeichenfehler	Truncation	Unter-/Überlauf
Q.S.1	✓		
Q.S.2	✓		
Q.S.3	✓		
Q.T.1	✓	✓	
Q.T.2		✓	
Q.O.1			✓
Q.O.2			✓

Tabelle 5.2 Übersicht über alle Anfragen und deren Fehlerklassen

```
1  g.V(module_id).repeat(out(llvmgraph::HIERARCHY_EDGE)).times(3)
2      .has("op_code", "trunc").get();
```

Codeblock 5.4 Erste Teilanfrage Q.T.1 und Q.T.2

```
1  g.V(index)  // Knoten mit Truncation laden
2      .repeat(_union(in(llvmgraph::USE_EDGE)
3                         ._not(has("op_code", "call")),
4                     out(llvmgraph::USED_AS_ARGUMENT_EDGE)
5                         .in(llvmgraph::USE_EDGE)
6              )
7              .choose(has("op_code", "store"),
8                      out(llvmgraph::USE_EDGE)
9                          .in(llvmgraph::USE_EDGE).simplePath()
10             )
11             .dedup().simplePath()
12     )
13     .until(has("signedness"))
14     .has("signedness", "signed").limit(1).get();
```

Codeblock 5.5 Zweite Teilanfrage Q.T.1

5.4.4 Implementierung

Im Folgenden sollen die Anfragen im Detail vorgestellt werden.

Truncation

Q.T.1 und Q.T.2 setzen sich beide jeweils aus zwei Teilen zusammen, von denen sie sich den ersten teilen. Dieser findet alle Stellen, an denen eine Truncation auftritt. Der zweite Teil spezialisiert diesen dann mit den oben vorgestellten Mustern.

Im ersten Teil (Codeblock 5.4) werden alle Instruktionen gesucht, die den Opcode „trunc“ haben. Durch das dynamische Erzeugen des Graphen, wird dabei beim Modulknoten angefangen.

Q.T.1 und Q.T.2 bekommen anschließend jeweils die Knoten der Ergebnismenge um diese weiterzuverarbeiten. Diese sind in den folgenden Anfragen mit dem Index „`index`“ versehen.

Q.T.1 sucht auf den Knoten weiter, bis eine Anweisung gefunden wird, die das Ergebnis der Truncation vorzeichenbehaftet verwendet. Die Anfrage findet sich in Codeblock 5.5. Dazu wird simultan innerhalb der Funktion die Kette der Benutzungen entlanggegangen oder bei Funktionsaufrufen dem Aufruf gefolgt (Zeile 2-6).

```
1  g.V(index)  // Knoten mit Truncation laden
2      .repeat(_union(in(llvmgraph::USE_EDGE),
3                     out(llvmgraph::USED_AS_ARGUMENT_EDGE)
4                         .in(llvmgraph::USE_EDGE)
5              )
6              .choose(has("op_code", "store"),
7                      out(llvmgraph::USE_EDGE)
8                          .in(llvmgraph::USE_EDGE).simplePath()
9              )
10             .dedup().simplePath()
11     ).emit()  // alle Benutzungen finden
12     .has("op_code", "call").dedup().aggregate("x")  // in x speichern
13     .count().V(index).dedup()  // Pfad löschen und Indexknoten bekommen
14     .out(llvmgraph::USE_EDGE)
15     .choose(has("op_code", "load"),
16             out(llvmgraph::USE_EDGE).in(llvmgraph::USE_EDGE).simplePath()
17     )
18     .in(llvmgraph::USE_EDGE)
19     ._not(has("op_code", "trunc"))  // andere Benutzer finden
20     .emit().repeat(_union(in(llvmgraph::USE_EDGE),
21                           out(llvmgraph::USED_AS_ARGUMENT_EDGE)
22                               .in(llvmgraph::USE_EDGE)
23                    )
24                    .choose(has("op_code", "store"),
25                            out(llvmgraph::USE_EDGE)
26                                .in(llvmgraph::USE_EDGE).simplePath()
27                    )
28                    .dedup().simplePath()
29     )  // alle Benutzungen der anderen Benutzer finden
30     .where(within("x"))  // vergleichen
31     .limit(1).get();  // beschränken
```

Codeblock 5.6 Zweite Teilanfrage Q.T.2

`store` und `load` Anweisungen werden übersprungen (Zeile 7-9). Diese Schritte werden solange wiederholt, bis eine Instruktion gefunden wurde, die das Vorzeichen des Operanden festlegt (Zeile 13). Falls dieser vorzeichenbehaftet ist, wird das Ergebnis ausgegeben (Zeile 14).

Im zweiten Teil von Q.T.2 wird von dem Truncation-Knoten in zwei Richtungen gesucht und anschließend verglichen. Codeblock 5.6 zeigt die vollständige Anfrage.

Die Teilanfrage lässt sich wiederum unterteilen (in Klammern ist jeweils die entsprechende Zeile der Anfrage angegeben):

1. Zeile 1-12. Dieser Teil geht von der Anweisung mit der Truncation aus (1) und verfolgt über die `USE_EDGE`s und die `USED_AS_ARGUMENT_EDGE`s (parallele Auswertung über `union()`, Zeile 2-4) die weitere Benutzung dieses Wertes. Übergangen werden hierbei `store`- und `load`-Anweisungen (6-8), da LLVM ohne Optimierung jede Variable als `volatile` betrachtet[28]. Diese Schritte werden solange wie möglich wiederholt (`repeat()`), dabei aber fortwährend ausgegeben (11). Abschließend werden alle Funktionsaufrufe in den Seiteninformationen unter dem Namen `"x"` abgespeichert.

2. Zeile 13-29. Dieser Teil sucht zuerst alle anderen Benutzungen der Eingabe der Truncation-Anweisung (13-19). Von diesen ausgehend wird wieder die vollständige Kette an Benutzungen (ähnlich zu 2-10) gesucht (20-28). Die andere Platzierung des `emit()`-Schrittes bewirkt die zusätzliche Ausgabe aller Elemente aus Zeile 19. Beachtenswert ist auch der `count()`-Schritt, dessen Eigenschaft, den Pfad aller Traverser zu zerstören und gleichzeitig genau ein Ergebnis zu liefern, missbraucht wird.

3. Zeile 30 und 31. Abschließend werden die im zweiten Teil gefundenen Instruktionen mit denen aus der Menge `"x"` verglichen und der erste Treffer ausgegeben.

Vorzeichenfehler

Vorzeichenfehler werden durch die Anfragen Q.S.1 - Q.S.3 aufgedeckt.

```
1  g.V(module_id).repeat(out(llvmgraph::HIERARCHY_EDGE)).times(3)
2      .has("op_code", "sext").as("a")
3      .in(llvmgraph::USE_EDGE)
4      .choose(has("op_code", "store"),
5              out(llvmgraph::USE_EDGE).in(llvmgraph::USE_EDGE).simplePath()
6      )
7      .values("function_name").filter(FilterWithRegex(r)).as("b")
8      .select("a", "b").get();
```

Codeblock 5.7 Anfrage Q.S.1

Q.S.1 ist in Codeblock 5.7 definiert. Es werden erneut alle Anweisungen abgerufen (Zeile 1) und danach nach vorzeichenbehafteten Erweiterungen durchsucht (Zeile 2). Anschließend werden die Benutzer der Erweiterung gesucht (mit Übergehung von `store` und `load`, Zeile 3-6). Falls diese ein Funktionsaufruf sind, wird der Name

[28] `volatile` heißt, dass kein Wert in einem Register gehalten wird, sondern jedes Mal explizit in den Speicher wegspeichert und neu geladen wird.

mit dem regulären Ausdruck „`.*(memset|vsnprintf|alloc|memcpy).*`“ gefiltert (Zeile 7). Die Funktionen, die von dem regulären Ausdruck erkannt werden, haben alle ein Argument vom Typ `size_t`. Somit erkennt die Anfrage u. a. die falsche Verwendung von Variablen des Typs `int`.

```
1  g.V(module_id).repeat(out(llvmgraph::HIERARCHY_EDGE)).times(3)
2      .has("comparison").has("signedness", "signed").as("a")
3      .flatMap(out(llvmgraph::USE_EDGE)
4               .choose(has("op_code", "load"),
5                       out(llvmgraph::USE_EDGE)
6               )
7               .in(llvmgraph::USE_EDGE).simplePath()
8               .choose(has("op_code", "sext"),
9                       in(llvmgraph::USE_EDGE)
10              )
11              .in(llvmgraph::USE_EDGE)
12              .values("function_name").filter(FilterWithRegex(r)).limit(1)
13     ).as("b")
14     .select("a", "b").get();
```

Codeblock 5.8 Anfrage Q.S.2

Der Code von Q.S.2 findet sich in Codeblock 5.8. Die Anfrage sucht zuerst alle Vergleiche, die vorzeichenbehaftet vergleichen (Zeile 1-2), und führt anschließend in einem gesonderten Geltungsbereich (erreicht durch den `flatMap`-Schritt) die weitere Suche aus. Das führt vor allem dazu, dass der `limit`-Schritt (Zeile 12) nicht global limitiert. In dem gesonderten Geltungsbereich werden zuerst die Operanden des Vergleichs angefordert (Zeile 3-6, `load` wird wieder übergangen). Danach werden ihre Benutzungen in einer Funktion gesucht (Zeile 7-12, vorzeichenbehaftete Erweiterungen werden übergangen), die vorzeichenlose Werte erwartet. Dafür wird mit dem oben schon vorgestellten regulären Ausdruck gefiltert (Zeile 12).

In Codeblock 5.9 findet sich der Code für Q.S.3. Diese Anfrage sucht zuerst alle Instruktionen, die vorzeichenlos erweitern (Zeile 1-2) und ruft anschließend die Instruktionen ab, in denen die Operanden zusätzlich benutzt werden (Zeile 3-10). Anschließend wird geprüft, ob es sich bei der Instruktion um eine vorzeichenbehaftete Erweiterung handelt (Zeile 11). Durch das Limitieren auf ein Ergebnis muss die Anfrage erneut in einem gesonderten Geltungsbereich ablaufen (`flatMap`, Zeile 3-12).

Unter- und Überläufe

Anfrage Q.O.1 (Codeblock 5.10) sucht nach Stellen, bei denen direkt in einer Funktion, die durch den oben vorgestellten regulären Ausdruck definiert wird, addiert

```
1  g.V(module_id).repeat(out(llvmgraph::HIERARCHY_EDGE)).times(3)
2      .has("op_code", "zext").as("a")
3      .flatMap(out(llvmgraph::USE_EDGE)
4                .choose(has("op_code", "load"),
5                        out(llvmgraph::USE_EDGE)
6                ).as("b")
7                .in(llvmgraph::USE_EDGE)
8                .choose(has("op_code", "load"),
9                        in(llvmgraph::USE_EDGE)
10               )
11               .has("op_code", "sext").limit(1).as("c")
12     )
13     .select({"a", "b", "c"}).get();
```

Codeblock 5.9 Anfrage Q.S.3

```
1  g.V(module_id).repeat(out(llvmgraph::HIERARCHY_EDGE)).times(3).as("a")
2      .values("function_name").filter(FilterWithRegex(r)).select("a")
3      .out(llvmgraph::USE_EDGE)
4      ._or(has("op_code", "add"),
5           has("op_code", "sub"))
6      .select("a").dedup().get();
```

Codeblock 5.10 Anfrage Q.O.1

oder subtrahiert wird. Dabei wird ausgenutzt, dass bei einer direkten Addition bzw. Subtraktion das Ergebnis nicht im Arbeitsspeicher zwischengespeichert wird und in der IR auch in unoptimierter Form kein `load` und `store` vorkommt. In der Anfrage werden dazu erst sämtliche eben definierten Funktionsaufrufe angefordert (Zeile 1-2) und anschließend die Operanden geprüft (Zeile 3-5).

Die Suche nach Variablen in Schleifen, die inkrementiert werden, ist relativ komplex. Die dazu notwendigen Schritte der Anfrage Q.O.2 finden sich Codeblock 5.11. Sie lassen sich in drei Teile unterteilen:

1. Zeile 1-5: Den Vergleich im Kopf der Schleife anfordern und die Bitbreite speichern. Dazu wird zuerst auf die Ebene der einfachen Blöcke gegangen (im Gegensatz zu den anderen Anfragen, Zeile 1-2). Die Breite wird mit dem Etikett `"b"` versehen (Zeile 5).

2. Zeile 6-12: Alle Variablen innerhalb der Schleife abrufen, die in einer Addition vorkommen und eine kleinere Bitbreite haben. Die Prüfung, ob der Operand

```
1  g.V(module_id).repeat(out(llvmgraph::HIERARCHY_EDGE)).times(2)
2      .has("is_loop_header").as("a")  // Kopf der Schleife
3      .out(llvmgraph::HIERARCHY_EDGE).has("op_code", "icmp")  // Vergleich
4      .out(llvmgraph::USE_EDGE).has("op_code", "load")
5      .values("bit_width").as("b")  // Bitbreite der Variablen des Vergleichs
6      .select("a").dedup()
7      .in(llvmgraph::LOOP_EDGE).out(llvmgraph::LOOP_EDGE)
8      .out(llvmgraph::HIERARCHY_EDGE)  // alle Instruktionen der Schleife
9      .has("op_code", "add").as("c")                    // Additionen mit ge-
10     .values("bit_width").where(lt("b")).select("c")  // ringerer Bitbreite
11     .out(llvmgraph::USE_EDGE)
12     .has("op_code", "load")  // auf Variablen der Hochsprache beschränken
13     .out(llvmgraph::USE_EDGE).in(llvmgraph::USE_EDGE)
14     .has("op_code", "store").out(llvmgraph::USE_EDGE).aggregate("d")
15     .select("c").dedup().where(within("d")) // Operand der Addition wird
16     .get();                                 // mit Ergebnis überschrieben
```

Codeblock 5.11 Anfrage Q.O.2

der Addition eine `load`-Instruktion ist, garantiert, dass es sich um eine Variable handelt, die im Speicher liegt.

3. Zeile 13-15: Alle Vorkommen suchen, in denen diese Variable gespeichert wird (Zeile 13-14) und deren Benutzungen mit der davor abgespeicherten Addition übereinstimmen (Zeile 14-15).

Kapitel 6
Auswertung

Um den Gremlin-Pass zu evaluieren, wurden sowohl Messungen auf einer großen Menge Software durchgeführt, als auch einige Fallstudien vorgenommen. Im Folgenden werden erst die quantitativen Messungen vorgestellt und anschließend die Fallstudien.

6.1 Messungen

Um Einschätzungen hinsichtlich der Trefferhäufigkeit der Anfragen und des damit verbundenen zusätzlichen Rechenaufwands zu bekommen, wurde der Gremlin-Pass mit einer großen Anzahl Software getestet.

Dazu wurden verschiedene Testsets benutzt, die im Folgenden zuerst vorgestellt werden. Anschließend werden die Ergebnisse präsentiert und diskutiert.

6.1.1 Testset

Ziele bei der Auswahl des Testset waren zum einen das massenhafte Testen von Software, die im Alltag benutzt wird, und zum anderen das Testen auf bekannten Benchmarks. Der technische Vorgang besteht dabei aus dem Kompilieren der Software mit Clang bei gleichzeitigem Setzen von zusätzlichen Optionen des Compilers („Compiler-Flags“)[29].

Aus diesem Grund war eine Umgebung notwendig, die diese Eigenschaften auf möglichst generischem Weg bereitstellt. Dies trifft auf die Linux-Distribution Gentoo zu [43].

Getestet wurden darum 395 Pakete der Distribution[30] (im Folgenden T_{All}) zusammen mit einem Subset der CPUINT2006 Testsuite[31] (im Folgenden T_{SPEC}).

CPUINT2006 ist ein Benchmark der *Standard Performance Evaluation Corporation*[32] (SPEC), der die Geschwindigkeit von CPUs und damit implizit auch die

[29] Im Anhang A.1 wird die Benutzung genauer beschrieben.
[30] Im Speziellen waren dies Pakete der Basisdistribution zusammen mit einigen ausgewählten zusätzlichen Paketen, wie den verschiedenen Teilen des X-Servers, der Grafiksoftware Gimp und dem Texteditor Kate.
[31] Es handelt sich um die Tests 401-bzip2, 403-gcc, 429-mcf, 445-gobmk, 456-hmmer, 458-sjeng, 462-libquantum, 464-h264ref, 471-omnetpp und 473-astar.
[32] Eine Organisation, die sich speziell mit dem Erstellen von Benchmarks beschäftigt.

Güte des kompilierten Binärcodes misst [44]. Letztere Eigenschaft hängt bei gleichem Quellcode von den Optimierungen des Compilers ab, sodass der Benchmark auch dazu dient. Der Benchmark ist eine Untermenge des CPU2006 Benchmarks und befasst sich speziell mit dem Testen von Ganzzahloperationen. Die LLVM-Testumgebung bringt bereits Unterstützung für die Benchmarks der SPEC mit, die für diese Arbeit verwendet wurde.

Die Zeitmessungen wurden auf einer ausgewählten Untermenge von 50 Elementen von T_{All} ausgeführt (im Folgenden T_{Time})[33].

6.1.2 Testumgebung

Getestet wurde auf einem separaten Computer mit einem Intel Core i7-7700 mit einem Basistakt von 3.6 GHz. Sämtliche Software wurde vollständig im Arbeitsspeicher gehalten und kompiliert.

6.1.3 Quantitative Messungen

Ziel der quantitativen Messungen war es, Ergebnisse darüber zu bekommen, welche Anfragen in handelsüblicher Software welche Menge an Ergebnissen liefern. Dazu wurden alle Anfragen auf T_{All} und T_{SPEC} angewandt und ausgewertet. In Tabelle 6.1 ist die Auflistung der Ergebnisse zu sehen.

Dort ist zum einen die absolute Anzahl und die Anzahl pro kompilierter Datei erkennbar.

Gut zu sehen sind die verschiedenen Häufigkeiten der Ergebnisse für die Anfragen. Anfrage Q.O.2, die am meisten spezialisiert ist, liefert die wenigstens Resultate. Anfrage Q.S.2 und Q.T.1, die eher allgemein gehalten sind, liefern am häufigsten Resultate. Die relative Häufigkeit der Anfragen ist bei allen Anfragen unter zwei Ergebnissen für die Anfrage pro Datei und daher in einem akzeptablen Rahmen.

Vergleicht man verschiedene Programme, ist eine stark unterschiedliche Anzahl der Treffer zu erkennen. So haben z. B. die beiden gut getesteten und etablierten Programme GTK+ und x264 eine sehr verschiedene relative Gesamtanzahl von 1.52 Treffer pro Datei im Vergleich zu 17.83 Treffern pro Datei. Der Grund könnte die durchaus beabsichtigte sehr häufige Typkonvertierung in hoch optimierter Software wie x264 sein, die in diesem Fall zu einer hohen Anzahl von falschen Treffern führt. Die Software x264 arbeitet außerdem häufig mit Präprozessormakros, die durch die Codeverdopplung an vielen Stellen im Code zu immer dem gleichem Treffer führen.

[33] Eine Liste der Pakete findet sich im Anhang A.2.

	T_{All}		gtk+		gimp		less		gzip		python		ffmpeg		dolphin		x264	
Anfrage	abs.	rel.	abs.	rel.	abs.	rel.	abs.	rel.	abs.	rel.	abs.	rel.	abs.	rel.	abs.	rel.	abs.	rel.
Q.O.1	7381	0.26	41	0.05	34	0.02	1	0.03	6	0.10	84	0.39	113	0.07	23	0.17	4	0.13
Q.O.2	570	0.02	0	0.00	9	0.01	0	0.00	21	0.33	10	0.05	18	0.01	0	0.00	0	0.00
Q.S.1	14408	0.50	298	0.38	1341	0.88	7	0.20	3	0.05	28	0.13	2361	1.41	18	0.13	21	0.70
Q.S.2	57664	1.99	461	0.59	367	0.24	0	0.00	25	0.40	1743	8.07	6445	3.85	149	1.07	276	9.20
Q.S.3	5850	0.20	74	0.10	108	0.07	20	0.57	2	0.03	32	0.15	92	0.06	0	0.00	11	0.37
Q.T.1	47564	1.65	288	0.37	587	0.39	41	1.17	26	0.41	301	1.39	6659	3.98	47	0.34	213	7.10
Q.T.2	5562	0.19	18	0.02	61	0.04	4	0.11	10	0.16	38	0.18	1113	0.67	0	0.00	10	0.33
Gesamt	138999	4.81	1180	1.52	2507	1.65	73	2.09	93	1.48	2236	10.35	16801	10.05	237	1.71	535	17.83

	401		403		429		445		456		458		462		464		471		473	
Anfrage	abs.	rel.	abs.	rel.	abs.	rel.	abs.	rel.	abs.	rel.	abs.	rel.	abs.	rel.	abs.	rel.	abs.	rel.	abs.	rel.
Q.O.1	0	0.00	36	0.23	1	0.08	0	0.00	34	0.59	0	0.00	0	0.00	2	0.05	0	0.00	0	0.00
Q.O.2	0	0.00	3	0.02	0	0.00	3	0.04	0	0.00	0	0.00	0	0.00	0	0.00	0	0.00	0	0.00
Q.S.1	9	0.90	141	0.90	0	0.00	8	0.12	72	1.24	3	0.15	11	0.65	59	1.37	6	0.07	11	0.92
Q.S.2	4	0.40	717	4.60	0	0.00	159	2.34	222	3.83	14	0.70	2	0.12	11	0.26	0	0.00	1	0.08
Q.S.3	0	0.00	262	1.68	0	0.00	0	0.00	12	0.21	3	0.15	3	0.18	1	0.02	0	0.00	0	0.00
Q.T.1	21	2.10	329	2.11	1	0.08	17	0.25	67	1.16	4	0.20	10	0.59	162	3.77	47	0.55	1	0.08
Q.T.2	0	0.00	46	0.29	0	0.00	0	0.00	1	0.02	0	0.00	0	0.00	2	0.05	0	0.00	0	0.00
Gesamt	34	3.40	1534	9.83	2	0.17	187	2.75	408	7.03	24	1.20	26	1.53	237	5.51	53	0.62	13	1.08

Tabelle 6.1 Absolute und relative Treffer (Treffer pro kompilierter Datei) der Anfragen auf T_{All}, einigen ausgewählten bekannteren Programmen und T_{SPEC}

Auf den SPEC Benchmarks ist zu erkennen, dass besonders bei 403-gcc eine Menge von Ergebnissen gefunden wurde, die über dem Durchschnitt liegt. Die Werte zeigen allgemein eine ähnliche Verteilung wie bei T_{All}. Der Test 429-mcf hat von allen Tests die wenigsten Resultate. Betrachtet man die 403-gcc im Detail, ist zu erkennen, dass sehr häufig Treffer gefunden werden, die durch große „switch-case“ oder „if-else“ Blöcke mit sehr ähnlichem Code innerhalb der Blöcke hervorgerufen werden.

6.1.4 Messung des Aufwands

Gemessen wurde bei jedem Paket der Zeitstempel zu Beginn und zum Ende des Kompilierungsprozesses[34], sowie die Zeiten[35] zum Ausführen des Passes für jede kompilierte Datei. Alle Zeitmessungen wurden auf nur einem Kern ausgeführt. Die zugrunde liegenden Testsets aller Zeitmessungen waren T_{Time} und T_{SPEC}.

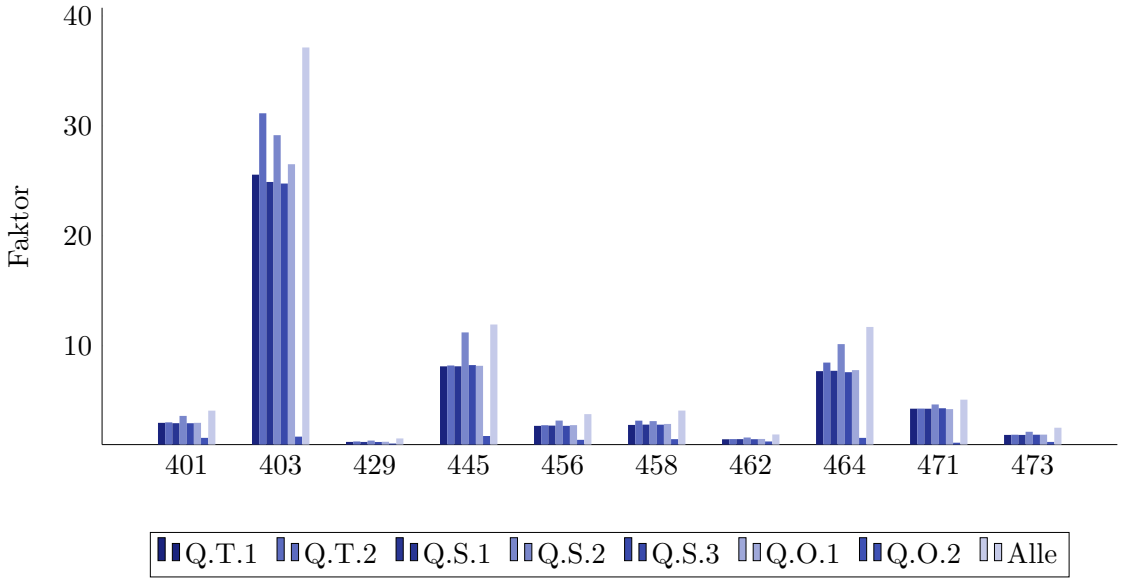


Abbildung 6.1 Faktor, um den sich die Kompilierzeit auf T_{SPEC} bei Aktivierung des Gremlin-Pass im Vergleich zum Clang alleine verlängert. Die Skala beginnt bei 1.

In Abbildung 6.1 ist dargestellt, um welchen Faktor sich der Kompiliervorgang auf T_{SPEC} beim Aktivieren des Passes im Vergleich zum direkten Kompilieren mit Clang verlängert. Angegeben sind dabei die Verlängerungen bei der Ausführung

[34] Das Entpacken und Konfigurieren der Software ist von der Messung ausgenommen.
[35] Es wurde nur die Zeit gemessen, die der Prozessor mit dem Berechnen des Prozesses verbringt.

aller Anfragen in einem Durchlauf, als auch bei der Ausführung jeder Anfrage einzeln.

Man erkennt vor allem bei 403-gcc eine deutliche Zunahme der Kompilierzeit. Die Gründe liegen nicht in der höheren Anzahl von Ergebnissen, da sonst z. B. 445-gobmk und 456-hmmer anders verteilt wären, sondern in deutlich größeren bzw. komplexeren Quellcodedateien im Vergleich zu anderen Programmen. Dies wird später genauer diskutiert. 445-gobmk und 464-h264ref liegen bei einem Faktor zwischen 12 und 13. Alle anderen Teile besitzen einen Faktor unter 6 und liegen damit in einem akzeptablen Rahmen.

Auffällig ist die sehr geringe Ausführungszeit von Q.O.2. Dies lässt sich damit erklären, dass diese Anfrage als einzige schon auf der Basis von einfachen Blöcken einen Großteil der Knoten aussortiert. Alle anderen Anfragen fordern als erstes alle Instruktionen an, um dann auszusortieren.

Gut zu erkennen ist zudem der nur sehr gering gestiegene Faktor der Ausführung aller Abfragen in einem Durchlauf im Vergleich zur Summierung der Faktoren der einzelnen Anfragen. Dies ist damit zu erklären, dass der LLVM-Graph zu Zwecken der Eindeutigkeit bereits erstellte Knoten und Kanten zwischenspeichert und bei einer gemeinsamen Ausführung der Anfragen der gleiche Graph verwendet wird. Da außerdem jede Anfrage (außer Q.O.2) als ersten Schritt die Menge der Instruktionen aufbaut, ist der initiale Schritt, den Graphen zu erzeugen, bereits getan.

Aufgrund der hohen Diversität der Faktoren der einzelnen Anfragen wurde in Abbildung 6.2 die Verteilung des Faktors als Boxplot dargestellt. Es lassen sich die gleichen Aussagen wie auf den SPEC Benchmarks treffen. Gut zu erkennen ist die sehr hohe Spanne zwischen minimalen und maximalen Faktoren für jeweils eine Anfrage. 75% der Programme liegen allerdings bei einem Faktor unter 7 und liegen somit in einem akzeptablen Rahmen.

Die starken Ausreißer nach oben lassen sich mit der sehr verschiedenen Komplexität und Anzahl der Quellcodezeilen pro Datei und Programm erklären. Der Gremlin-Pass arbeitet immer auf dem vollständigen Modul. Eine Datei mit vielen Zeilen zu durchsuchen, benötigt daher deutlich mehr Zeit als bei einer Datei mit wenigen Zeilen. Ähnliches gilt für die Komplexität bzw. die Anzahl der Verschachtelungen des Graphen, was zu einer Erhöhung der Kantenzahl führt.

Dazu wurde eine weitere Messung durchgeführt, die die Zeit abhängig von der IR-Instruktionen-Anzahl[36] misst. Es war wichtig, als Maß die Zahl der IR-Instruktionen und nicht der Quellcodezeilen der Hochsprache zu wählen, da beide Maße nur teilweise zusammenhängen.

[36] Die Anzahl wurde ebenfalls mit einer Gremlin-Anfrage gemessen.

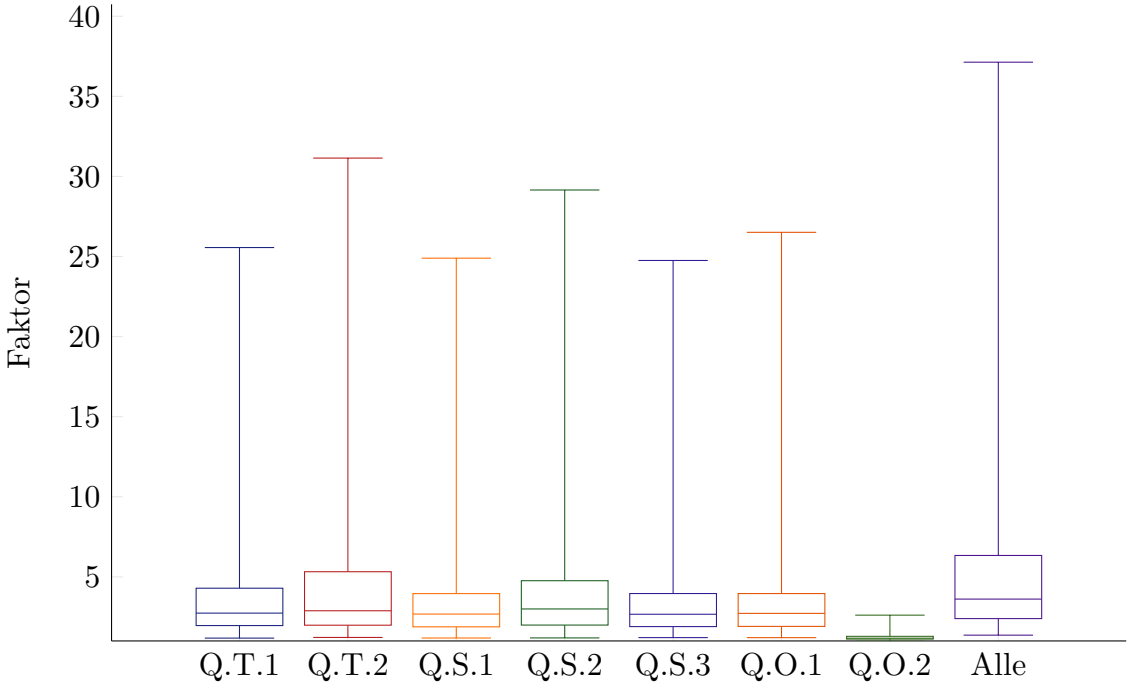


Abbildung 6.2 Faktor auf T_{Time}, um den sich die Kompilierzeit bei Aktivierung des Gremlin-Pass im Vergleich zu Clang allein verlängert. Die Skala beginnt bei 1. Markiert sind jeweils das Minimum, das Maximum, der Median und die unteren 25% und 75%.

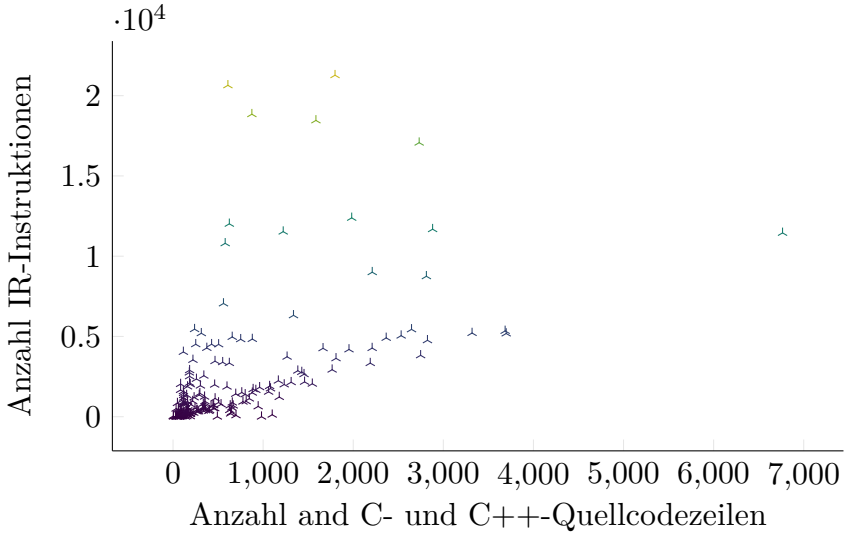


Abbildung 6.3 Anzahl IR-Instruktionen abhängig von der Anzahl an C- bzw. C++-Quellcodezeilen

In Abbildung 6.3 ist die Verteilung der beiden Größen grafisch dargestellt (es wurden dazu 200 zufällige Messwerte gewählt). Man erkennt eine lineare Grundrichtung, die aber deutlich viele „Ausreißer“ zeigt, bei denen eine C/C++-Datei mit wenig Quellcodezeilen sehr viele IR-Instruktionen erzeugt. Das lässt sich mit dem Präprozessor erklären, der bei beiden Sprachen zum Einsatz kommt. Er kann zum einen mit der Anweisung `#include` das Einfügen quasi beliebig vieler neuer Zeilen bewirken und zum anderen mit der Makroexpansion die Anzahl der resultierenden Quellcode-Zeilen um Größenordnungen erhöhen.

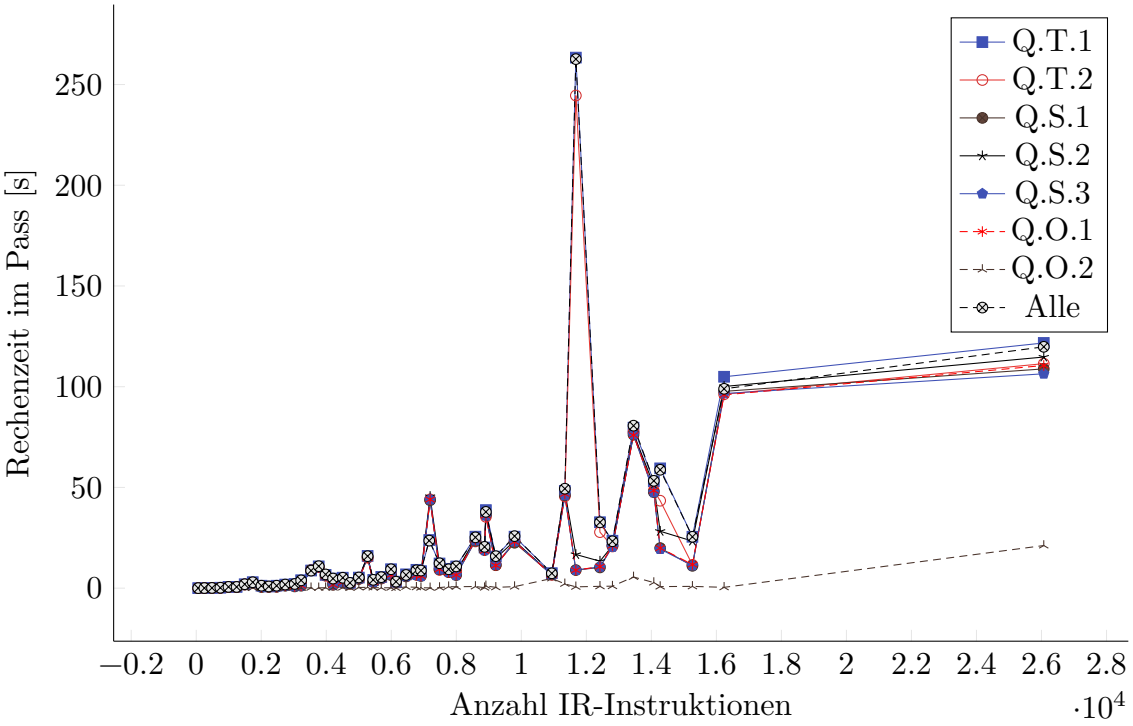


Abbildung 6.4 Zeitverbrauch des Gremlin-Pass in s, abhängig von der Anzahl der IR-Instruktionen

Abbildung 6.4 zeigt die Zeit, die Clang mit dem Berechnen des Passes verbringt, abhängig von der Anzahl der IR-Instruktionen. Man erkennt einen linearen Anstieg. Der Anstieg macht sich besonders bei Programmen wie Clang selbst oder Cython bemerkbar, bei denen IR-Module mit über 100.000 Instruktionen die Regel sind. Der Ausreißer bei etwa 12000 Instruktionen gehört einer Datei des ICU-Projekts, die sich durch eine hohe Schachtelungstiefe auszeichnet und so den Graphen überproportional wachsen lässt.

Für die meisten Programme ist die Größe der Eingangsdatei jedoch gering genug, sodass der Gremlin-Pass im normalen Gebrauch benutzbar ist.

6.2 Fallstudien

Die Entwicklung des Gremlin-Passes hat sich stark an [42] orientiert. Damit verbunden waren auch einige Fallstudien, die ebenfalls getestet wurden. Außerdem wurde eine weitere Schwachstelle aufgedeckt, die zusätzlich vorgestellt worden soll.

6.2.1 Übersicht

Der Gremlin-Pass wurde an einigen bekannten Schwachstellen getestet, dazu gehören CVE-2007-1884 (PHP) und CVE-2013-0211 (libarchive). CVE-2007-1884 konnte nachgestellt werden, CVE-2013-0211 hängt von Daten ab, die über die Modulgrenze hinausgehen und ist damit nicht zu finden. Außerdem wurden die Schwachstellen, die in [42] zusätzlich entdeckt wurden, ebenfalls gesucht. Die dortigen Ergebnisse konnten bis auf einige Ausnahmen nachgestellt werden[37].

Auf die Schwachstellen CVE-2007-1884, CVE-2013-0211 und eine weitere Schwachstelle in zlib aus [42] soll im Folgenden genauer eingegangen werden, anschließend wird die neu gefundene Schwachstelle präsentiert.

6.2.2 CVE-2007-1884

Bei der Schwachstelle handelt es sich um eine Truncation, die entsteht, indem das Ergebnis der Funktion `long php_sprintf_getnumber(...)` insgesamt dreimal einer ganzen Zahl vom Typ `int` zugewiesen wird [45]. Auf einer 32-Bit-Architektur ist dies unproblematisch, führt aber bei 64 Bit dazu, dass eine größere Zahl zurückgegeben werden kann, als in `int` passt und somit zu einem negativen Wert führt.

Zu Demonstrationszwecken soll die Anwendung des Gremlin-Passes hier genauer thematisiert werden. Auf die Datei angewandt, die für den Fehler sorgt, liefert der Pass insgesamt 15 Treffer, neun davon vom Typ Q.T.1 und die restlichen sechs vom Typ Q.S.1. CVE-2007-1884 wird durch die Anfrage Q.T.1 an genau den drei genannten Stellen gefunden. Von den übrigen sechs sind drei explizite Casts und können ignoriert werden. Die letzten Stellen sind die Codezeilen:

1. `*p1 += 5;`
2. `i = (endptr - &buffer[*pos]);`
3. `cvt_len = strlen(cvt);`

[37] Der Linux Kern ist ohne Anpassungen mit Clang nicht kompilierbar und wurde darum nicht berücksichtigt. Außerdem wurden die Lücken in Boost, libstdc++ und Chromium außer Acht gelassen, da der in den entsprechenden Dateien vorhandene GNU-Inline-Assembler-Code ohne weitere Konfiguration nicht zu übersetzen war.

`p1` ist ein Zeiger auf `char`, sodass die Zuweisung der Konstante vom Typ `int` eine Truncation verursacht.

Im zweiten Fall sind „`endptr`“ und „`buffer`“ beide Zeiger (also 64 Bit breit), die Differenz wird aber einer Zahl vom Typ `int` (32 Bit) zugewiesen. Die Differenz bzw. der Wert von „`i`“ ist dabei maximal so groß wie die Länge einer dezimal notierten Zahl. Da genau diese Zahl aber davor schon in einer Variable des Typs `long` gespeichert wird und die Länge somit maximal 63 betragen kann, ist die Stelle unkritisch.

Der dritte Fall weist die zurückgegebene Zahl des Typs `size_t` einer Variablen des Typs `int` zu und kann damit prinzipiell zu Problemen führen. Die Eingabe „`cvt`“ ist allerdings vom Programm erzeugt und kann damit nicht überlaufen.

Hier ist gut zu erkennen, dass der Gremlin-Pass zwar falsche Ergebnisse liefert, diese aber teilweise nur durch den Kontext identifizierbar sind und somit schwer automatisch auszusortieren sind.

6.2.3 CVE-2013-0211

Die Schwachstelle in libarchive basiert auf einem Vorzeichenfehler, der aufgrund der anderen Wertebereiche erst bei 64 Bit auftritt. Verantwortlich sind die beiden Zeilen aus Codeblock 6.1 [46].

```
1     if ((int64_t)s > zip->remaining_data_bytes)
2         s = (size_t)zip->remaining_data_bytes;
```

Codeblock 6.1 Kritische Stelle von CVE-2013-0211

Die Variable „`s`“ hat den Typ `size_t` und die Variable „`remaining_data_bytes`“ den Typ `int64_t` (synonym zu `long` auf 64-Bit-Systemen). Wenn die Variable „`s`“ zu groß wird, wird sie fälschlicherweise zu einer negativen Zahl umgewandelt und der Vergleich liefert „unwahr“ zurück. Das führt im weiteren Verlauf zu einem Buffer Overflow.

Der entsprechende IR-Code ist in Codeblock 6.2 abgebildet. Man erkennt den vorzeichenbehafteten Vergleich zwischen den Variablen in Zeile 4 und weiterhin die Zuweisung zu „`s`“ in Zeile 11. Da es sich nur um 64 Bit große Datentypen handelt, kommen keine Truncations oder Erweiterungen vor.

Ein Vorzeichenfehler ist in diesem Teil des Codes nicht sichtbar, da Vorzeichen in der IR nicht mit Datentypen abgebildet werden. Der Buffer Overflow entsteht jedoch erst bei der weiteren Benutzung von `s`. Genau danach könnte man mit einer Anfrage suchen. In diesem Fall wird `s` aber erst wieder in einer weiteren Funktion vorzeichenbehaftet verwendet, der die Variable als Parameter übergeben

```
1    %5 = load i64, i64* %s.addr, align 8
2    ...
3    %7 = load i64, i64* %remaining_data_bytes, align 8
4    %cmp = icmp sgt i64 %5, %7
5    br i1 %cmp, label %if.then, label %if.end
6
7  if.then:                                          ; preds = %entry
8    %8 = load %struct.zip*, %struct.zip** %zip, align 8
9    %remaining_data_bytes1 = ...
10   %9 = load i64, i64* %remaining_data_bytes1, align 8
11   store i64 %9, i64* %s.addr, align 8
12   br label %if.end
```

Codeblock 6.2 IR-Code der kritischen Stelle von CVE-2013-0211

wird. Diese Funktion liegt außerhalb der Datei und somit dem LLVM-Modul und ist damit nicht erreichbar. Die momentanen Möglichkeiten des Passes können also diesen Fehler nicht finden.

6.2.4 Überlauf in zlib

In [42] wurde ein weiterer Überlauf in zlib gefunden, der mit dem Gremlin-Pass ebenfalls gefunden wird.

Der kritische Code ist in Codeblock 6.3 zu sehen.

```
1    len = vsnprintf((char *)(state->in), size, format, va);
```

Codeblock 6.3 Überlauf in zlib

`vsnprintf` ist eine Funktion der Standardbibliothek und erwartet als zweiten Parameter eine Variable vom Typ `size_t`. Die übergebene Variable „`size`“ ist jedoch vom Typ `int`, sodass der Compiler automatisch eine vorzeichenbehaftete Erweiterung einfügt. Beinhaltet „`size`“ eine negative Zahl, führt dies zu einem Overflow. Die Anfrage, die bei diesem Fehler anschlägt, ist Q.S.1.

Der Gremlin-Pass liefert auf dieser Datei außer diesem noch fünf Ergebnisse. Diese werden alle durch die Anfrage Q.T.1 ausgelöst. Drei davon sind explizite Casts und können ignoriert werden[38]. Die beiden übrigen Ergebnisse verweisen auf folgende Zeilen:

1. `got = write(state->fd, strm->next_in, strm->avail_in);`

[38] Dies geschieht in der Annahme, dass der Programmierer den Cast aus guten Gründen eingefügt hat.

2. `if (have && ((got = write(state->fd, state->x.next, have)) < 0 ||`

Im ersten Fall liefert der Funktionsaufruf eine `size_t` zurück, die in die Variable „`got`“ vom Typ `int` geschrieben wird, und somit eine Truncation auslöst. Diese wird aber direkt in der nächsten Zeile abgefangen, die „`got`“ u. a. auf negative Werte überprüft.

Der zweite Fall ist zum ersten äquivalent. Hier wird allerdings „`got`“ anschließend jedes Mal explizit in den Typen `unsigned int` umgewandelt, sodass der Aufruf unkritisch ist.

6.2.5 Überlauf in Audacity

Audacity ist eine in C++ geschriebene Software, um Tondateien zu verarbeiten. Die Software besitzt aus diesem Grund verschiedene Parser.

Bei der Anwendung des Gremlin-Passes wird von Anfrage Q.T.1 u. a. Zeile 5 des Codestücks in Codeblock 6.4 ausgegeben. Es kommt aus dem Parser für Allegro-Musikdateien.

```
1  int Alg_reader::find_real_in(string &field, int n)
2  {
3      // scans from offset n to the end of a real constant
4      bool decimal = false;
5      int len = field.length();
6      ...
7      return len;
8  }
```

Codeblock 6.4 Truncation in Audacity

Die Funktion „`field.length()`“ liefert eine `size_t` zurück, sodass es möglich ist, in die Variable „`len`“ negative Werte zu schreiben, wenn „`field.length()`“ manipulierbar ist. Die Variable „`len`“ wird außerdem von der Funktion zurückgegeben. Sucht man folglich die Stellen, in denen „`find_real_in`“ verwendet wird, stößt man auf die Funktion „`Alg_reader::parse()`“, die die vollständige Datei parst. Der Wert von „`field.length()`“ entspricht dabei der Länge einer Zeile der Eingabedateien.

Aus diesem Grund wurde Audacity eine Allegro-Datei gegeben, die eine Zeile mit einer Länge größer als 2^{31} besaß. Audacity ist bei dieser Datei mit einem Speicherzugriffsfehler abgestürzt. Die abschließende Analyse hat aber ergeben, dass nicht die beschrieben Zeilen dafür verantwortlich sind, sondern der Fehler schon davor durch einen Überlauf einer Variablen vom Typ `int` auftritt.

Um die Datei zu parsen, wird sie zeilenweise in eine Variable vom Typ `std::string` geladen, die dann zeichenweise über eine Variable „`pos`“ vom Typ `int` verarbeitet wird. Ist eine Zeile bzw. die resultierende Zeichenkette länger als $2^{31} - 1$ bzw. der maximale Wert, den die Zahl auf einer 64-Bit-Architektur annehmen kann, dann kommt es zu einem Überlauf der entsprechenden Variable und das Programm stürzt beim nächsten Zugriff auf die Zeichenkette an der jetzt falschen Position ab. Die entsprechende Codestelle ist in Codeblock 6.5 zu sehen.

```
1  void String_parse::get_nonspace_quoted(string &field)
2  {
3      ...
4      while ((*str)[pos] && (quoted || !isspace((*str)[pos]))) {
5          ...
6          pos = pos + 1;
7          ...
8      }
9  }
```

Codeblock 6.5 Überlauf in Audacity

Sowohl die Variable „`str`“ (Zeiger auf `std::string`) als auch „`pos`“ sind Datenfelder der Klasse. Die zeichenweise Verarbeitung ist in Zeile 4 zu erkennen, das Inkrementieren in Zeile 6.

Dieser Fehler tritt nur bei großen Eingabedateien auf, die zu einem vielfach höheren Arbeitsspeicherverbrauch führen, sodass ein Auftreten des Fehlers in der Praxis insgesamt unwahrscheinlich ist. Der Überlauf wurde den Entwicklern von Audacity gemeldet.

Kapitel 7
Zusammenfassung

Diese Arbeit besteht aus drei Teilen, die aufeinander aufbauen. Der erste Teil besteht in der Untersuchung und dem Nachbau der Sprache Gremlin. Daraus ist die C++-Bibliothek „libgremlin“ hervorgegangen, die in der Lage ist, Gremlin-Anfragen auszuführen. Diese sind dabei direkt in die Sprache C++ eingebettet und ermöglichen so eine intuitive Programmierung. Die Anfragen werden auf Basis eines Graphen ausgeführt, der aber nicht Bestandteil der Bibliothek ist, sondern als Schnittstelle definiert wird und folglich auf den jeweiligen Anwendungszweck optimiert bzw. dynamisch erzeugt werden kann.

Auf Basis dieser Bibliothek wurde dann für LLVM mit dem Gremlin-Pass ein Aufsatz geschaffen, der einen solchen Graphen bereitstellt und es ermöglicht, das Programm direkt zur Übersetzungszeit nach bestimmten Mustern zu durchsuchen. Die Ausgaben folgen dabei dem Muster von Compiler-Warnungen, können aber durch die allgemeine Struktur der Anfragen im Gegensatz zu diesen sehr einfach erweitert und angepasst werden.

Anschließend wurden Sicherheitslücken untersucht, die hauptsächlich durch die verschiedene Handhabung von ganzen Zahlen bei unterschiedlichen Architekturen auftreten. Diese wurden in Muster in der IR übertragen und Anfragen dafür geschaffen.

Die Effektivität der Implementierung und der Anfragen konnte anhand bestehender Sicherheitslücken nachgewiesen und überdies gezeigt werden, dass der Pass sehr einfach auf einer große Menge Software anwendbar ist. Durch die Beschränkung auf einzelne Kompiliereinheiten sind allerdings auch die Grenzen der Suche deutlich geworden. Abschließend wurde bei der Analyse ein weiterer Programmfehler entdeckt, der durch den größeren Addressraum von 64-Bit-Umgebungen entsteht.

Ziel der Arbeit war es, ein Werkzeug zu schaffen, das Softwareentwickler bereits beim Programmieren dabei unterstützt, bestimmte Fehler gar nicht erst zu begehen. Dieses Werkzeug sollte dabei möglichst wenig zusätzlichen Aufwand erfordern. Dieses Ziel wurde mit dem Gremlin-Pass erfüllt, der über das einfache Setzen von Zusatzoptionen in den Compiler eingehängt werden kann und so ohne gesonderten Mehraufwand ausgeführt wird. Durch die Möglichkeit, beliebige Muster mit wenig Code suchen zu lassen, ist zudem eine einfache Erweiterung auf die jeweiligen Wünsche des Anwenders möglich.

Durch die Trennung der Bibliothek „libgremlin“ vom Gremlin-Pass wurde zudem ein weiteres Werkzeug geschaffen, das zwar vom Gremlin-Pass benutzt wird, aber auch in einem völlig anderen Zusammenhang eingesetzt werden kann.

7.1 Ausblick

Wie bei der Zusammenfassung bereits angesprochen, liegt sowohl der Umfang als auch die Grenze der Suche bei der jeweiligen Kompiliereinheit. Für die Zukunft ist darum eine feinere Abstufung des Umfangs denkbar, bei der anhand des gesuchten Musters ausgewählt werden kann, in welchem Rahmen die Suche abläuft. Dies würde einerseits die Suche über Kompiliereinheiten hinaus, andererseits aber auch eine Beschränkung auf z. B. Schleifen und so eine deutliche Beschleunigung ermöglichen.

In der Bibliothek „libgremlin“ ist bereits ein Modul enthalten, dass in der Lage ist, Gremlin-Anfragen, die als Zeichenketten übergeben werden, auszuführen. Dieses Modul kann ausgebaut und für den Gremlin-Pass verfügbar gemacht werden, sodass nicht nur der Gremlin-Pass, sondern auch die Anfragen selbst dem Compiler als Argumente übergeben werden könnten. Aus Gründen des Umfangs wurde in dieser Arbeit darauf verzichtet.

Um das Problem der langen Laufzeiten bei der Kompilierung einiger Programme anzugehen, kann überdies die Auswertung der Gremlin-Anfrage parallelisiert werden.

Die Suche nach Mustern wird weiterhin erheblich einfacher, wenn diese zentral gesammelt werden. Eine Schaffung einer solchen Struktur ist darum wünschenswert.

Bei der Evaluierung der Anfragen wurde bislang nur mit C und C++ getestet. Durch die hohe Popularität der Programmiersprache Rust, die zudem Speichersicherheit als Ziel hat, wäre eine Untersuchung von Programmen dieser Sprache ansprechend.

Anhang

A.1 Benutzung von libgremlin und dem Gremlin-Pass

Der Gremlin-Pass kann als „Shared Library“ direkt von Clang geladen werden.

Dazu muss zuerst libgremlin kompiliert und installiert werden. Anschließend kann der Gremlin-Pass kompiliert werden. Die Anweisungen dazu können jeweils den beigefügten Dateien „README.md“ entnommen werden.

Nach der Übersetzung liegt im Build-Verzeichnis die Datei „libGremlinPass.so“. Diese kann dann direkt mit Clang verwendet und liefert Ausgaben wie in Codeblock A.1.

```
$ clang -Xclang -load -Xclang /path/to/libGremlinPass.so -g -Rpass=gremlin-
pass my_file.c
my_file.c:6:14: remark: Truncation[2] found. Both the input and output of
      the truncation are used in line 8. [-Rpass=gremlin-pass]
        int size2 = size;
                    ^

my_file.c:7:21: remark: Signedness[1]: Found signed extension, that is used
      in unsigned context (function malloc). [-Rpass=gremlin-pass]
        char* buf = malloc(size2);
                           ^
```

Codeblock A.1 Benutzung und Ausgabe des Gremlin-Pass

Um ein vollständiges Programm mit dem Gremlin-Pass zu kompilieren, reicht es zumeist, die Variablen „CC“ und „CFLAGS“ anzupassen, die von allen gängigen Build-Systemen verstanden werden[39].

Will man die Bibliothek libgremlin direkt verwenden, zeigt Codeblock A.2 ein Minimalbeispiel. Die Klasse „`Gremlin`“, von der geerbt wird, dient dazu, einen Satz vordefinierte Datenfelder und alle Gremlin-Schritte bereitzustellen.

[39] Falls das Programm GNU Make benutzt, muss außerdem noch das generierte Shellskript „libtool“ nach dem Konfigurieren gepatcht werden, da sonst das Linken fehlschlägt. Die Ursache liegt in dem Shellskript, das mit den zusätzlichen Parametern von Clang nicht umgehen kann.

```cpp
#include <iostream>
#include <vector>

#include "libgremlin/gremlin.h"
#include "libgremlin/graph_modern.h"
#include "libgremlin/path_node.h"

class MyGremlin : public gremlin::Gremlin {
public:
    MyGremlin(gremlin::Graph& graph) : gremlin::Gremlin(graph) {}

    void myMethod() {
        vector<gremlin::FlatNode> result = g.V().values("age").max().get();
        std::cout << result[0] << std::endl;
    }
};

int main() {
    gremlin::ModernGraph graph;
    MyGremlin mg = MyGremlin(graph);
    mg.myMethod();
}
```

Codeblock A.2 Minimalbeispiel für die Benutzung der libgremlin

A.2 Testsets im Detail

Nachfolgend ist das Testset T_{All} gelistet. Die Namen entsprechen dabei den Paketnamen der Linuxdistribution Gentoo. Alle mit einem Stern (*) markierten Pakete sind überdies Teil des Testsets T_{TIME}.

app-accessibility/at-spi2-core-2.22.1
app-accessibility/at-spi2-atk-2.22.0
app-arch/bzip2-1.0.6-r8
app-arch/cpio-2.12-r1*
app-arch/gnome-autoar-0.2.2
app-arch/gzip-1.8
app-arch/libarchive-3.3.1
app-arch/lz4-1.7.5-r1
app-arch/rpm2targz-9.0.0.5g
app-arch/snappy-1.1.3-r1
app-arch/tar-1.29-r1
app-arch/unzip-6.0_p20
app-arch/xz-utils-5.2.3*
app-arch/zip-3.0-r3*
app-crypt/gcr-3.20.0
app-crypt/gnupg-2.1.20-r1
app-crypt/p11-kit-0.23.2*
app-crypt/pinentry-0.9.7-r1
app-editors/gedit-3.22.0
app-editors/vim-8.0.0386
app-editors/vim-core-8.0.0386
app-misc/geoclue-2.4.4
app-misc/pax-utils-1.1.7
app-misc/tmux-2.3-r1
app-portage/eix-0.32.4
app-portage/portage-utils-0.62
app-shells/bash-4.3_p48-r1
app-shells/quoter-3.0_p2*
app-text/evince-3.22.1
app-text/ghostscript-gpl-9.21
app-text/hunspell-1.6.1-r1*
app-text/libpaper-1.1.24-r2
app-text/libspectre-0.2.7
app-text/manpager-1*
app-text/poppler-0.45.0
app-text/xmlto-0.0.26-r1
dev-lang/orc-0.4.26-r1
dev-lang/perl-5.24.1-r2
dev-lang/python-3.4.5
dev-lang/python-exec-2.4.4
dev-lang/ruby-2.2.6
dev-lang/swig-3.0.12
dev-libs/atk-2.22.0
dev-libs/boehm-gc-7.4.2
dev-libs/dbus-glib-0.108*
dev-libs/gjs-1.46.0
dev-libs/expat-2.2.1*
dev-libs/glib-2.50.3-r1
dev-libs/gmp-6.1.0
dev-libs/gobject-introspection-1.50.0
dev-libs/hyphen-2.8.8
dev-libs/icu-58.2-r1*
dev-libs/iniparser-3.1-r1*
dev-libs/json-c-0.12
dev-libs/json-glib-1.2.8
dev-libs/jsoncpp-1.8.1
dev-libs/libaio-0.3.110
dev-libs/libassuan-2.4.3-r1
dev-libs/libatasmart-0.19-r1
dev-libs/libatomic_ops-7.4.2
dev-libs/libbsd-0.8.3*
dev-libs/libcroco-0.6.12-r1*
dev-libs/libdaemon-0.14-r2
dev-libs/libdbusmenu-qt-0.9.3_pre20160218
dev-libs/libevdev-1.5.7
dev-libs/libevent-2.1.8
dev-libs/libffi-3.2.1
dev-libs/libgcrypt-1.7.7
dev-libs/libgit2-0.24.6
dev-libs/libgpg-error-1.27-r1
dev-libs/libgudev-230-r1*
dev-libs/libinput-1.7.2
dev-libs/libksba-1.3.5-r1
dev-libs/libltdl-2.4.6*
dev-libs/libnl-3.2.28
dev-libs/libpcre-8.40-r1
dev-libs/libpeas-1.20.0-r1
dev-libs/libpipeline-1.4.0
dev-libs/libtasn1-4.10-r2
dev-libs/libuv-1.10.2
dev-libs/libxml2-2.9.4-r1
dev-libs/libxslt-1.1.29-r1
dev-libs/mpc-1.0.2-r1
dev-libs/lzo-2.09
dev-libs/libyaml-0.1.7
dev-libs/mpfr-3.1.3_p4
dev-libs/nettle-3.3-r1
dev-libs/newt-0.52.15
dev-libs/npth-1.3
dev-libs/nspr-4.13.1
dev-libs/openssl-1.0.2k
dev-libs/popt-1.16-r2
dev-libs/wayland-1.13.0
dev-perl/HTML-Parser-3.710.0-r1
dev-perl/Locale-gettext-1.50.0-r1
dev-perl/Net-SSLeay-1.810.0
dev-perl/TermReadKey-2.330.0
dev-perl/Text-CharWidth-0.40.0-r1
dev-perl/XML-Parser-2.440.0
dev-perl/libintl-perl-1.240.0-r2
dev-python/cffi-1.9.1
dev-python/markupsafe-0.23
media-video/ffmpeg-3.2.4
dev-python/pygobject-3.22.0
dev-python/pyxattr-0.5.5
dev-qt/linguist-tools-5.6.2
dev-qt/qtchooser-0_p20151008*
dev-qt/qtconcurrent-5.6.2*
dev-qt/qtcore-5.6.2-r1
dev-qt/qtdbus-5.6.2
dev-qt/qtimageformats-5.6.2
dev-qt/qtnetwork-5.6.2*
dev-qt/qtopengl-5.6.2
dev-qt/qtprintsupport-5.6.2
dev-qt/qtquickcontrols-5.6.2
dev-qt/qtscript-5.6.2
dev-qt/qtsql-5.6.2
dev-qt/qtsvg-5.6.2
dev-qt/qttest-5.6.2
dev-qt/qtwayland-5.6.2
dev-qt/qtwebchannel-5.6.2
dev-qt/qtx11extras-5.6.2*
dev-qt/qtxml-5.6.2
dev-ruby/json-1.8.3
dev-ruby/racc-1.4.11
dev-util/boost-build-1.62.0-r1*
dev-util/ctags-20161028
dev-util/desktop-file-utils-0.23
dev-util/gperf-3.0.4*
dev-util/gtk-update-icon-cache-3.22.2*
dev-util/ninja-1.7.2
dev-util/pkgconfig-0.28-r2
dev-util/ragel-6.7-r1
dev-util/re2c-0.16
dev-vcs/git-2.13.0
gnome-base/dconf-0.26.0-r1
gnome-base/gnome-desktop-3.22.2
gnome-base/gvfs-1.30.4
gnome-base/libglade-2.6.4-r2
gnome-base/librsvg-2.40.17
gnome-base/nautilus-3.22.3
gnome-extra/polkit-gnome-0.105-r1*
gnome-extra/sushi-3.24.0
kde-apps/dolphin-16.12.3
kde-apps/gwenview-16.12.3
kde-apps/kate-16.12.3
kde-apps/kio-extras-16.12.3-r1
kde-frameworks/attica-5.34.0
kde-frameworks/frameworkintegration-5.34.0
kde-frameworks/kactivities-5.34.0
kde-frameworks/karchive-5.34.0
kde-frameworks/kauth-5.34.0
kde-frameworks/kbookmarks-5.34.0
kde-frameworks/kcmutils-5.34.0
kde-frameworks/kcodecs-5.34.0*
kde-frameworks/kcompletion-5.34.0
kde-frameworks/kconfig-5.34.0
kde-frameworks/kconfigwidgets-5.34.0
kde-frameworks/kcoreaddons-5.34.0-r1
kde-frameworks/kcrash-5.34.0
kde-frameworks/kdbusaddons-5.34.0
kde-frameworks/kdeclarative-5.34.0
kde-frameworks/kded-5.34.0
kde-frameworks/kdelibs4support-5.34.0*
kde-frameworks/kdesignerplugin-5.34.0*
kde-frameworks/kdesu-5.34.0
kde-frameworks/kdnssd-5.34.0
kde-frameworks/kdoctools-5.34.0
kde-frameworks/kemoticons-5.34.0
kde-frameworks/kglobalaccel-5.34.0
kde-frameworks/kguiaddons-5.34.0
kde-frameworks/ki18n-5.34.0
kde-frameworks/kiconthemes-5.34.0
kde-frameworks/kidletime-5.34.0
kde-frameworks/kimageformats-5.34.0
kde-frameworks/kinit-5.34.0
kde-frameworks/kitemmodels-5.34.0
kde-frameworks/kitemviews-5.34.0
kde-frameworks/kjobwidgets-5.34.0
kde-frameworks/kjs-5.34.0
kde-frameworks/knewstuff-5.34.0
kde-frameworks/knotifications-5.34.0
kde-frameworks/kpackage-5.34.0
kde-frameworks/kparts-5.34.0
kde-frameworks/kpty-5.34.0
kde-frameworks/kservice-5.34.0
kde-frameworks/ktextwidgets-5.34.0
kde-frameworks/kunitconversion-5.34.0
kde-frameworks/kwallet-5.34.0-r1
kde-frameworks/kwayland-5.34.0
kde-frameworks/kwidgetsaddons-5.34.0*
kde-frameworks/kwindowsystem-5.34.0
kde-frameworks/kxmlgui-5.34.0
kde-frameworks/plasma-5.34.0
kde-frameworks/solid-5.34.0
kde-frameworks/sonnet-5.34.0
kde-frameworks/syntax-highlighting-5.34.0
kde-frameworks/threadweaver-5.34.0
kde-plasma/breeze-5.8.6
kde-plasma/kactivitymanagerd-5.8.6
kde-plasma/kde-cli-tools-5.8.6-r1
kde-plasma/kdecoration-5.8.6
kde-plasma/kscreenlocker-5.8.6
kde-plasma/polkit-kde-agent-5.8.6

Anhang

media-fonts/font-util-1.3.1
media-gfx/exiv2-0.25-r2
media-gfx/gimp-2.9.4-r3
media-gfx/graphite2-1.3.10*
media-libs/alsa-lib-1.1.2
media-libs/babl-0.1.18
media-libs/celt-0.11.1-r1
media-libs/clutter-1.26.2
media-libs/clutter-gst-3.0.24
media-libs/clutter-gtk-1.8.2
media-libs/cogl-1.22.2*
media-libs/flac-1.3.2-r1
media-libs/fontconfig-2.11.1-r2
media-libs/freeglut-3.0.0
media-libs/freetype-2.8
media-libs/gegl-0.3.8*
media-libs/glu-9.0.0-r1
media-libs/gst-plugins-bad-1.10.3
media-libs/giflib-5.1.4
media-libs/gst-plugins-base-1.10.3
media-libs/gst-plugins-good-1.10.3
media-libs/gstreamer-1.10.3
media-libs/harfbuzz-1.4.5
media-libs/jasper-2.0.12
media-libs/jbig2dec-0.13-r4
media-libs/lcms-2.8-r1
media-libs/libdvbpsi-1.3.0-r1
media-libs/libepoxy-1.4.2
media-libs/libjpeg-turbo-1.5.1
media-libs/libmng-2.0.2-r1
media-libs/libmypaint-1.3.0
media-libs/libogg-1.3.2
media-libs/libpng-1.6.27
media-libs/libsamplerate-0.1.9
media-libs/libtheora-1.1.1-r1
media-libs/libv4l-1.10.1
media-libs/libvorbis-1.3.5
media-libs/libvpx-1.5.0
media-libs/libwebp-0.5.2
media-libs/musicbrainz-5.1.0
media-libs/openjpeg-2.1.1_p20160922
media-libs/opus-1.1.3-r1
media-libs/phonon-4.9.1-r1
media-libs/phonon-vlc-0.9.1-r1*
media-libs/speex-1.2_rc1-r2
media-libs/tiff-4.0.8
media-libs/x264-0.0.20160712
media-plugins/gst-plugins-v4l2-1.10.3
media-sound/lame-3.99.5-r1
media-video/ffmpeg-3.2.4
net-dns/libidn-1.33
net-dns/libidn2-0.16-r1
net-firewall/iptables-1.4.21-r1*
net-libs/glib-networking-2.50.0
net-libs/gnutls-3.5.13
net-libs/http-parser-2.6.2*
net-libs/libmbim-1.14.0
net-libs/libmnl-1.0.4*
net-libs/libndp-1.6-r1*
net-libs/libproxy-0.4.13-r2
net-libs/libqmi-1.16.2
net-libs/libsoup-2.56.0
net-libs/libsrtp-1.4.4_p20121108-r1
net-libs/libssh-0.7.4
net-libs/neon-0.30.2*
net-misc/curl-7.54.1
net-misc/dhcp-4.3.4
net-misc/networkmanager-1.4.4-r1
net-misc/modemmanager-1.6.4
net-misc/openssh-7.5_p1-r1*
net-misc/rsync-3.1.2*
net-misc/wget-1.19.1-r1
net-wireless/b43-fwcutter-019
sys-apps/acl-2.2.52-r1
sys-apps/attr-2.4.47-r2
sys-apps/busybox-1.25.1
sys-apps/coreutils-8.25
sys-apps/dbus-1.10.18
sys-apps/debianutils-4.7
sys-apps/diffutils-3.5
sys-apps/file-5.30
sys-apps/findutils-4.6.0-r1
sys-apps/gawk-4.1.3
sys-apps/gentoo-functions-0.10
sys-apps/gptfdisk-1.0.1
sys-apps/grep-2.27-r1
sys-apps/groff-1.22.2
sys-apps/help2man-1.46.6
sys-apps/install-xattr-0.5
sys-apps/iproute2-4.4.0
sys-apps/kbd-2.0.3
sys-apps/kmod-23
sys-apps/less-487
sys-apps/man-db-2.7.6.1-r2
sys-apps/net-tools-1.60_p20160215155418
sys-apps/openrc-0.26.3
sys-apps/paxctl-0.9
sys-apps/pciutils-3.4.1
sys-apps/portage-2.3.6
sys-apps/sandbox-2.10-r3
sys-apps/sed-4.2.2
sys-apps/shadow-4.4-r2
sys-apps/sysvinit-2.88-r9
sys-apps/texinfo-6.3
sys-apps/util-linux-2.28.2
sys-apps/which-2.21
sys-auth/nss-mdns-0.10-r3
sys-auth/polkit-0.113
sys-auth/polkit-qt-0.112.0-r1
sys-block/parted-3.2
sys-boot/grub-2.02
sys-boot/plymouth-0.9.2
sys-devel/bc-1.06.95-r1
sys-devel/bison-3.0.4-r1*
sys-devel/flex-2.6.1
sys-devel/libtool-2.4.6-r3
sys-devel/patch-2.7.5
sys-devel/make-4.2.1
sys-devel/m4-1.4.17
sys-fs/btrfs-progs-4.10.2
sys-fs/e2fsprogs-1.43.3-r1
sys-fs/lvm2-2.02.145-r2
sys-fs/udisks-2.1.8
sys-libs/compiler-rt-sanitizers-4.0.1
sys-libs/cracklib-2.9.6-r1
sys-libs/e2fsprogs-libs-1.43.3
sys-libs/gdbm-1.11*
sys-libs/libomp-4.0.1
sys-libs/libcap-2.24-r2*
sys-libs/gpm-1.20.7-r2
sys-libs/libseccomp-2.3.2
sys-libs/libutempter-1.1.6-r2
sys-libs/mtdev-1.1.5
sys-libs/ncurses-6.0-r1
sys-libs/pam-1.2.1
sys-libs/readline-6.3_p8-r3*
sys-libs/slang-2.3.0
sys-libs/timezone-data-2017a
sys-libs/zlib-1.2.11
sys-power/cpupower-4.9.0
sys-process/cronie-1.5.0-r1
sys-process/htop-2.0.2
sys-process/procps-3.3.12
sys-process/psmisc-22.21-r3
www-client/links-2.14
x11-apps/mkfontscale-1.1.2
x11-apps/xauth-1.0.10
x11-apps/xprop-1.2.2*
x11-apps/xset-1.2.3*
x11-libs/cairo-1.14.8
x11-libs/gdk-pixbuf-2.36.6
x11-libs/gtk+-3.22.15
x11-libs/gtksourceview-3.22.2
x11-libs/libICE-1.0.9-r1
x11-libs/libSM-1.2.2-r1
x11-libs/libX11-1.6.5
x11-libs/libXScrnSaver-1.2.2-r1
x11-libs/libXau-1.0.8
x11-libs/libXaw-1.0.13
x11-libs/libXcomposite-0.4.4-r1
x11-libs/libXcursor-1.1.14
x11-libs/libXdamage-1.1.4-r1*
x11-libs/libXdmcp-1.1.2-r1
x11-libs/libXext-1.3.3
x11-libs/libXfixes-5.0.3*
x11-libs/libXft-2.3.2
x11-libs/libXi-1.7.9
x11-libs/libXmu-1.1.2
x11-libs/libXpm-3.5.12
x11-libs/libXrandr-1.5.1*
x11-libs/libXrender-0.9.10*
x11-libs/libXt-1.1.5*
x11-libs/libXtst-1.2.3
x11-libs/libXv-1.0.11
x11-libs/libXxf86vm-1.1.4
x11-libs/libdrm-2.4.80
x11-libs/libfontenc-1.1.3*
x11-libs/libpciaccess-0.13.4
x11-libs/libxcb-1.12-r2
x11-libs/libxkbcommon-0.7.1
x11-libs/libxshmfence-1.2
x11-libs/pango-1.40.5
x11-libs/pixman-0.34.0
x11-libs/xcb-util-0.4.0
x11-libs/xcb-util-cursor-0.1.3-r1
x11-libs/xcb-util-image-0.4.0
x11-libs/xcb-util-keysyms-0.4.0
x11-libs/xcb-util-renderutil-0.3.9-r1
x11-libs/xcb-util-wm-0.4.1-r1
x11-misc/shared-mime-info-1.8
x11-themes/gtk-engines-adwaita-3.22.3

Literatur

[1] Thomas Pichler, *Malware-Jubiläum: 20 Jahre Internet-Würmer – Heutige Bedrohungen erreichen kaum mehr große Bekanntheit*, (2008). (Online; aufgerufen am 24-Juli-2017) (https://www.pressetext.com/news/20081101001)

[2] Martin Holland und Axel Kannenberg, *WannaCry: Angriff mit Ransomware legt weltweit Zehntausende Rechner lahm*, (2017). (Online; aufgerufen am 24-Juli-2017) (https://heise.de/-3713235)

[3] The University of Maryland, *FindBugs™ - Find Bugs in Java Programs*, (2017). (Online; aufgerufen am 25-Juli-2017) (http://findbugs.sourceforge.net/index.html)

[4] S. C. Johnson, Lint, a C program checker, *Computing Science TR* (1978).

[5] Diverse, *Available Checkers*, (2017). (Online; aufgerufen am 25-Juli-2017) (https://clang-analyzer.llvm.org/available_checks.html)

[6] Cristian Cadar, Daniel Dunbar, und Dawson Engler, KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs., In OSDI (2008).

[7] Guodong Li, Indradeep Ghosh, und Sreeranga Rajan, KLOVER: A symbolic execution and automatic test generation tool for C++ programs, In Computer Aided Verification, Springer (2011).

[8] Milena Vujosevic Janicic, *LAV*, (2017). (Online; aufgerufen am 25-Juli-2017) (http://argo.matf.bg.ac.rs/?content=lav)

[9] Carsten Sinz, Stephan Falke, und Florian Merz, A Precise Memory Model for Low-Level Bounded Model Checking., In Ralf Huuck, Gerwin Klein, und Bastian Schlich (Herausgeber) SSV (USENIX Association, 2010).

[10] Diverse, *Coccinelle: A Program Matching and Transformation Tool for Systems Code*, (2017). (Online; aufgerufen am 25-Juli-2017) (http://coccinelle.lip6.fr/)

[11] Simon Goldsmith, Robert O'Callahan, und Alex Aiken, *Light-weight instrumentation from relational queries over program traces*, Technischer Bericht (CALIFORNIA UNIV BERKELEY DEPT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCES, 2004).

[12] Seth Hallem, Benjamin Chelf, Yichen Xie, und Dawson Engler, A System and Language for Building System-specific, Static Analyses, In Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (ACM, New York, NY, USA, 2002).

[13] Monica S. Lam, John Whaley, V. Benjamin Livshits, Michael C. Martin, Dzintars Avots, Michael Carbin, und Christopher Unkel, Context-sensitive Program Analysis As Database Queries, In Proceedings of the Twenty-fourth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (ACM, 2005).

[14] Michael Martin, Benjamin Livshits, und Monica S. Lam, Finding Application Errors and Security Flaws Using PQL: A Program Query Language, In Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (ACM, 2005).

[15] Fabian Yamaguchi, *Joern - A Robust Code Analysis Platform for C/C++*, (2017). (Online; aufgerufen am 16-Januar-2017) (http://mlsec.org/joern/)

[16] Fabian Yamaguchi, Nico Golde, Daniel Arp, und Konrad Rieck, Modeling and Discovering Vulnerabilities with Code Property Graphs, In 2014 IEEE Symposium on Security and Privacy (IEEE, 2014).

[17] Michał Zalewski, *American fuzzy lop (2.49b)*, (2017). (Online; aufgerufen am 25-Juli-2017) (http://lcamtuf.coredump.cx/afl/)

[18] Chris Lattner, *LLVM: An Infrastructure for Multi-Stage Optimization*, (Masterarbeit). Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL (2002). (*See* `http://llvm.org/pubs/2002-12-LattnerMSThesis.html`.)

[19] Chris Lattner und Vikram Adve, LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation, In Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04) (Palo Alto, California, 2004). (*See* `http://llvm.org/pubs/2004-01-30-CGO-LLVM.html`.)

[20] Chris Lattner, *The LLVM Compiler Infrastructure Project*, (2017). (Online; aufgerufen am 01-Juli-2017) (http://llvm.org/)

[21] Chris Lattner, *Chris Lattner*, (2017). (Online; aufgerufen am 01-Juli-2017) (http://nondot.org/sabre/Resume.html)

[22] Chris Lattner, *Clang: a C language family frontend for LLVM*, (2017). (Online; aufgerufen am 01-Juli-2017) (http://clang.llvm.org/)

[23] Apple Inc., *About Objective-C*, (2014). (Online; aufgerufen am 01-Juli-2017) (https://developer.apple.com/library/content/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/Introduction/Introduction.html)

[24] Roman Divacky, *[ANNOUNCE]: clang/llvm can compile booting FreeBSD kernel on i386/amd64*, (2009). (Online; aufgerufen am 01-Juli-2017) (https://lists.freebsd.org/pipermail/freebsd-current/2009-February/003743.html)

[25] Sylvestre Ledru, *Build of the Debian archive with clang*, (2016). (Online; aufgerufen am 01-Juli-2017) (http://clang.debian.net/)

[26] Chris Lattner, *Clang vs Other Open Source Compilers*, (2017). (Online; aufgerufen am 01-Juli-2017) (http://clang.llvm.org/comparison.html)

[27] The Linux Foundation, *LLVMLinux Project Overview*, (2017). (Online; aufgerufen am 01-Juli-2017) (http://llvm.linuxfoundation.org/index.php/Main_Page)

[28] Apple Inc., *Compiler and Standard Library*, (2017). (Online; aufgerufen am 01-Juli-2017) (https://swift.org/compiler-stdlib/#compiler-architecture)

[29] Rust Documentation Team, *Frequently Asked Questions*, (2017). (Online; aufgerufen am 01-Juli-2017) (https://www.rust-lang.org/en-US/faq.html)

[30] David A. Terei und Manuel M.T. Chakravarty, An LLVM Backend for GHC, In Proceedings of the third ACM Haskell symposium on Haskell (ACM, New York, NY, USA, 2010).

[31] Chris Lattner, *Projects built with LLVM*, (2017). (Online; aufgerufen am 01-Juli-2017) (http://llvm.org/ProjectsWithLLVM/)

[32] Cling Team, *Cling – Interactive Compiler Interface*, (2017). (Online; aufgerufen am 01-Juli-2017) (https://rawgit.com/vgvassilev/cling/master/www/index.html)

[33] Chris Lattner, *LLVM Type System Changes*, (2004). (Online; aufgerufen am 27-Juli-2017) (http://nondot.org/~sabre/LLVMNotes/TypeSystemChanges.txt)

[34] LLVM Project, *Source Level Debugging with LLVM*, (2017). (Online; aufgerufen am 01-Juli-2017) (http://llvm.org/docs/SourceLevelDebugging.html)

[35] Diverse, *DWARF FAQ*, (2017). (Online; aufgerufen am 01-Juli-2017) (http://wiki.dwarfstd.org/index.php?title=DWARF_FAQ)

[36] Marko A. Rodriguez, *TinkerPop as of Spring 2011*, (2011). (Online; aufgerufen am 01-Juli-2017) (https://markorodriguez.com/2011/03/03/tinkerpop-as-of-spring-2011/)

[37] The Apache Software Foundation, *TinkerPop Project Incubation Status*, (2017). (Online; aufgerufen am 01-Juli-2017) (http://incubator.apache.org/projects/tinkerpop.html)

[38] Marko A. Rodriguez, The Gremlin Graph Traversal Machine and Language, *CoRR* **abs/1508.03843** (2015).

[39] Marko A. Rodriguez, *Interface GraphTraversal<S,E>*, (2017). (Online; aufgerufen am 27-Juli-2017) (http://tinkerpop.apache.org/javadocs/3.2.5/core/org/apache/tinkerpop/gremlin/process/traversal/dsl/graph/GraphTraversal.html)

[40] American National Standards Institute, International Organization for Standardization, *American National Standard for Programming Languages – C*, Standard (American National Standards Institute, International Organization for Standardization, Geneva, CH, 1990).

[41] Aspen Data Model Committee, *64-Bit Programming Models: Why LP64?*, (1998). (Online; aufgerufen am 01-Juli-2017) (http://www.unix.org/version2/whatsnew/lp64_wp.html)

[42] Christian Wressnegger, Fabian Yamaguchi, Alwin Maier, und Konrad Rieck, Twice the Bits, Twice the Trouble: Vulnerabilities Induced by Migrating to 64-Bit Platforms., In ACM Conference on Computer and Communications Security (ACM, 2016).

[43] Diverse, *Clang*, (2017). (Online; aufgerufen am 25-Juli-2017) (https://wiki.gentoo.org/wiki/Clang)

[44] Standard Performance Evaluation Corporation, *Standard Performance Evaluation Corporation*, (2017). (Online; aufgerufen am 25-Juli-2017) (http://spec.org/)

[45] Stefan Esser, *MOPB-38-2007:PHP printf() Family 64 Bit Casting Vulnerabilities*, (2007). (Online; aufgerufen am 29-Juli-2017) (http://www.php-security.org/MOPB/MOPB-38-2007.html)

[46] Vincent Danen, *Bug 902998 - (CVE-2013-0211) CVE-2013-0211 libarchive: read buffer overflow on 64-bit systems*, (2013). (Online; aufgerufen am 29-Juli-2017) (https://bugzilla.redhat.com/show_bug.cgi?id=902998)